Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML

David Tarditi
Carnegie Mellon University
December, 1996
CMU–CS–97–108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Peter Lee, Chair
Steven Lucco
Nevin Heintze
Simon Peyton-Jones, Glasgow University

Copyright ©1996 David Tarditi

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Keywords: Standard ML, optimization, compilation, functional programming, garbage collection, automatic storage management

Abstract

The trends in software development are towards larger programs, more complex programs, and more use of programs as "component software." These trends mean that the features of modern programming languages are becoming more important than ever before. Programming languages need to have features such as strong typing, a module system, polymorphism, automatic storage management, and higher-order functions. In short, modern programming languages are becoming more important than ever before.

Even though modern programming languages are becoming more important than ever before, programmers have traditionally faced a dilemma: programs written in these languages traditionally have had lower performance than programs written in more conventional, but error-prone languages.

In this thesis, I study this problem in the context of one particular modern programming language, Standard ML. Standard ML contains all the language features mentioned previously and more. I use an empirical approach to understand where Standard ML programs spend their time and how to improve the performance of Standard ML programs though better optimization.

The thesis contains two main results. First, I find that a "pay-as-you-go" compilation strategy, where programmers pay for advanced language features only when they use them, is a practical strategy for compiling Standard ML. In fact, this strategy produces better code overall than a strategy that makes advanced language features run fast at the expense of slowing down programs that do not use those language features. Second, I find that compilers for Standard ML should focus on generating good code for the frequently-executed parts of programs. Specifically, just as compilers for conventional languages such as C focus on generating good code for loops, compilers for languages such as Standard ML should focus on generating good code for recursive functions.

These results suggest that compilation of modern programming languages such as Standard ML should have a great deal in common with compilation of more conventional languages such as C. First, Standard ML programs that do not use higher-order functions or polymorphism should run just as fast as comparable C programs. Second, Standard ML compilers should apply the same sets of optimizations to recursive functions that more conventional compilers apply to loops.

These results also suggest that programmers should be able to avoid the dilemma mentioned earlier: they should be able to write their programs in modern languages such as Standard ML, confident that they can rewrite parts of the programs in a subset of Standard ML if necessary for efficiency.

To my wife Liz

Contents

List of Figures

List of Tables

Preface

Writing this thesis was a long but exciting process. I was privileged to be able to build a sophisticated compiler from scratch — not many people have the luxury of being given the time and resources needed to do this.

There are many people that I would like to thank for their help. The TIL compiler was primarily joint work with Greg Morrisett. I would like to thank Greg for his hard work and enthusiasm. Without him, the compiler would not exist. Perry Cheng, Chris Stone, Robert Harper, and Peter Lee also worked on TIL. I would like particularly like to thank Perry for his work on the run-time system. The LMLI language that I describe in Chapter 4 of this thesis was designed by Greg Morrisett and Robert Harper.

The memory-system performance study of SML programs in Chapter 2 is joint work with Amer Diwan and Eliot Moss. The study of the cost of automatic storage management in Chapter 3 is also joint work with Amer Diwan. I would like to thank Amer for his work on these studies.

I would like to thank my advisor, Peter Lee, for his many years of encouragement and support of this work. Peter is a dedicated teacher and scholar, and I was privileged to be his student. Peter strongly encouraged me to work with Greg on the TIL compiler, on the belief that the whole would be more than the sum of the parts. The TIL compiler would not have come into existence without Peter's encouragement.

I would also like to thank the other members of my thesis committee: Steve Lucco, Nevin Heintze, and Simon Peyton-Jones. They made many excellent suggestions that improved the content of the thesis, and helped me to keep the larger picture in mind while conducting this work. I would especially like to thank them for promptly reading the thesis on relatively short notice and making many excellent comments.

I would like to thank AT&T and the Advanced Research Projects Agency (and thus the U.S. taxpapers) for their financial support of this work.

I would like to thank my parents and family for their many years of support and encouragement while I was in graduate school.

Last, and most importantly, I would like to think my wife Liz for her loving support, encouragement, and great patience while I worked on this thesis. This thesis took me longer to write than I expected: it was originally supposed to be finished while we were engaged, and we have now been married for over six months.

David Tarditi
Kirkland, Washington
December, 1996

Chapter 1

Introduction

1.1 The problem

The trends in software development are towards larger programs, more complex programs, and more use of programs as "component software." These trends have some important practical implications: programs are more likely to fail somewhere because bigger programs mean more chances for programmers to make mistakes, software is being developed by large teams, and software development and maintenance costs are increasing.

These implications mean that the features of modern programming languages are more important than ever before. Programming languages need to have features such as strong typing, a module system, polymorphism, automatic storage management, and higher-order functions:

- Strong typing reduces the chance of program failures.
- A module system is crucial for developing large software and reusable libraries of software.
- Polymorphism helps build reusable software.
- Automatic storage management reduces the chance of program failures. It eliminates a whole group of memory-management related programmer errors, such as "dangling pointers."
- Higher-order functions support code reuse and provide a natural way to implement callbacks.

In short, modern programming languages are becoming more important than ever before.

Even though modern programming languages are becoming more important than ever before, programmers have traditionally faced a dilemma: programs written in these languages traditionally have had lower performance than programs written in more conventional, but error-prone languages.

In this thesis, I study this problem in the context of one particular modern programming language, Standard ML [61] (SML). SML contains all the language features mentioned previously and more. I use an empirical approach to understand where SML programs spend their time and how to improve the performance of SML programs through better optimization.

1.2 Summary of results

This thesis contains two main results:

- First, I find that a "pay-as-you-go" compilation strategy, where programmers pay for advanced language features only when they use them, is a practical strategy for compiling SML. In fact, this strategy produces better code overall than a strategy that makes advanced language features run fast at the expense of slowing down programs that do not use those language features.

- Second, I find that compilers for SML should focus on generating good code for the frequently-executed parts of programs. Specifically, just as compilers for conventional languages such as C focus on generating good code for loops, compilers for languages such as SML should focus on generating good code for recursive functions.

These results suggest that compilation of modern programming languages such as SML should have a great deal in common with compilation of more conventional languages such as C:

- SML programs that do not use higher-order functions or polymorphism should run just as fast as comparable C programs.

- SML compilers should apply the same sets of optimizations to recursive functions that more conventional compilers apply to loops.

In short, these results suggest that programmers should be able to avoid the dilemma mentioned earlier: they should be able to write their programs in modern languages such as SML, confident that they can rewrite parts of the programs in a subset of SML if necessary for efficiency.

This thesis is divided into three parts. In the first part of the thesis, I study the performance of SML programs. I focus on two widely-held conjectures about where SML programs spend their time. First, I study the memory-system performance of SML programs, which is believed to be poor [7]. I also study the cost of automatic storage management. I find that the problem is neither memory-system performance nor the cost of automatic storage management. On the basis of these studies, and in particular the memory-system performance study, I conclude that SML programs are executing too many instructions and that we need better optimization of SML programs.

Before I began the work that I describe in the second part of the thesis, I profiled SML benchmark programs and identified the most frequently-executed parts of those programs. I then examined the machine code generated for the most frequently-executed parts of the programs, so that I could understand why SML programs are executing too many instructions. I noticed two things:

- Even though these parts of the programs were almost always monomorphic because of optimization, these parts still incurred many costs to support polymorphism. This suggested that instead of making polymorphic code fast at the expense of monomorphic code, we should make monomorphic code fast at the expense of polymorphic code. That is, this suggested that I should pursue a "pay-as-you-go" compilation strategy.

- There are many optimizations that are known to improve loops in conventional languages. Applying these optimizations to recursive functions in SML would improve the code generated for the most-frequently executed parts of the programs.

In the second part of the thesis, I describe a new approach to compiling SML programs based on these observations. First, I describe new algorithms that show how to apply several "loop" optimizations to SML programs. I focus on two sets of optimizations: code motion optimizations, such as common-subexpression elimination and invariant removal, and array-bounds checking optimizations.

It is important to emphasize that my algorithms are new, even though most of the optimizations are well-known. I designed new algorithms because I use a λ-calculus based intermediate language for optimization. This makes it impossible to apply the textbook versions of optimizations: the languages used in the textbooks differ too much from the λ-calculus. Those languages have only first-order functions, have a flat name space, are imperative and have looping constructs. In contrast, the intermediate language that I use has higher-order functions, has lexical scoping, emphasizes variable binding instead of assignment, and uses recursion instead of looping.

Because my algorithms work on programs in a λ-calculus based intermediate language, the techniques that I use in my algorithms are quite different from conventional optimization techniques [3]. Conventional techniques, for example, use control-flow graphs. In contrast, because there is only recursion in SML I use call graphs instead of control-flow graphs. Conventional techniques also use dataflow analysis. Because SML emphasizes binding instead of assignment, I use a simple side-effects analysis instead of dataflow analysis. Conventional techniques ignore lexical scoping, whereas I must spend a great deal of effort to preserve it.

In addition to describing new optimization algorithms in the second part of the thesis, I describe the TIL compiler, which is joint work primarily with Greg Morrisett. Perry Cheng, Chris Stone, Robert Harper, and Peter Lee also helped in the construction of TIL. The TIL compiler is organized around the idea of typed intermediate languages: we propagate type information through all phases of the compiler to machine code generation, and if necessary pass type information around at run time. The TIL framework, which is the subject of Morrisett's PhD thesis [63], allows us to implement two key parts of a "pay-as-you-go" compilation strategy:

- tag-free garbage collection,

- and compiling polymorphic functions to support native machine representations for data.

Conventional implementation approachs for languages such as SML have used a *universal representation* for data. All data is represented as machine word where part of the word is dedicated to a tag that is used by the garbage collector. Data that is naturally smaller than a machine word, such as characters, are padded to be the size as a machine word. Data that is naturally the same size as a machine word or larger than a machine word, such as a floating point numbers, are represented as pointers to the actual data. In other words, monomorphic programs that do not use garbage collection pay a price for polymorphism and garbage collection. This is not the case with TIL. TIL uses a *specialized* representation, where SML integers are represented as machine integers and SML floating point numbers are represented as machine floating point numbers, even when they are stored in arrays that are manipulated using polymorphic array operations.

In the TIL compiler, I take an approach to compiling languages with higher-order functions such as SML that is quite different from the approach suggested in the literature. Many researchers who have implemented languages with higher-order functions such as SML have stated the importance of compiling functions well [52, 7, 81] — they have focused on strategies for representing environments for first-class functions. I believe that this is not the central problem for compiling languages such as SML, because optimization is so effective at eliminating higher-order functions. I show the effectiveness of optimization at eliminating higher-order functions when SML programs are compiled as a whole in Chapter 10.1. In contrast to the approaches suggested in the literature, I take conventional compiler technology and adapt it to compile SML programs to machine code. Thus, I use a simple closure conversion strategy, but combine it with a conventional graph-coloring register allocator that uses callee and caller-save registers. To support the register allocator, I use a sophisticated garbage collector. I make certain that TIL converts constant expressions, such as constant records, to data at compile time. I use the standard system assembler to do instruction scheduling. The result is that TIL's translation to machine code is similar to the approach used in more conventional compilers.

In the third part of the thesis, I provide evidence to support my claims about how to compile SML. First, I establish that a "pay-as-you-go" compilation strategy is a practical strategy for compiling SML by comparing TIL to the SML/NJ compiler [7], a widely used reference compiler for SML. I find that TIL produces good code compared to the SML/NJ compiler. Indeed, TIL often produces code that is much better than that produced by the SML/NJ compiler. On DEC ALPHA workstations, programs compiled by TIL are roughly three times faster, do one-fifth the total heap allocation, and use one-half the physical memory of programs compiled by SML/NJ.

Second, I show that compilers for SML should focus on generating good code for recursive functions. First, even though I only implemented several of the many optimizations known to improve loops, I find that for my benchmarks these optimizations reduce execution time 8 to 87%, with a geometric mean reduction of 51%. I also measure the effect of the optimizations on the total amount of data heap allocated, the total amount of data copied by the garbage collector, maximum physical memory used by a program while it runs, and on program executable size. Second, I measure which of the new optimizations are most important to improving performance. Finally, I measure how specific these results are to the TIL framework.

I conclude this part of the thesis by measuring some other important aspects of TIL. First, I demonstrate that when SML programs are compiled as a whole, optimization is often highly effective at eliminating higher-order and polymorphic functions. Second, I study the effect of types on intermediate program size. The concern is that types could make intermediate programs much larger than their untyped versions, which could in turn slow down compilation and increase memory usage of a compiler. I find that with proper management of type information, typed intermediate programs are on average two times bigger than their untyped counterparts before optimization and only 15% bigger after optimization.

1.3 Examples

In this section, I present several examples taken from the body of the thesis that illustrate the main results of the thesis. The first example is about memory system performance. The second example illustrates the "pay-as-you-go" compilation strategy. The third example compares the performance of programs compiled by TIL to programs compiled by SML/NJ.

1.3.1 Example I: Memory system performance

In the first part of the thesis, one of the things that I study is the memory-system performance of SML programs. To study memory-system performance, I use trace-driven simulation of programs compiled by the SML/NJ compiler [7]. I instrument programs to produce a trace of the memory references made by programs, and use those traces to simulate various memory systems. This allows me to study memory-system performance across a variety of memory system architectures.

Figure 1.1 shows a breakdown of the memory system performance of one particular benchmark program, a program that does scheduling for a VLIW machine. The graph plots cycles per useful instruction versus cache size for memory systems with a particular set of characteristics (details are given in Chapter 2). Cycles per useful instruction is a measure of how well the memory system is being utilized. A lower number is better.

The memory system for the 64K point corresponds closely to the memory system for an actual machine, the DECStation 5000/200 [28]. At this point, the memory system performance is quite good (the cycles per useful instruction is about 1.4). Most cache misses are due to the instruction cache. This is not surprising, because VLIW is substantially larger than the cache (its executable size is about 500K). The cost of cache read misses is small. With this particular memory system, there is no cost for cache write misses.

This example illustrates an important finding of the memory system performance study: the memory-system performance of SML programs is good for some memory system configurations corresponding to actual machines. Specifically, for the memory-system configuration corresponding to the DECStation 5000/200, the memory-system performance of the SML benchmarks that I measure is comparable to that of C and Fortran programs [19]: programs run only 3 to 13% slower due to data-cache misses than they would run with a zero-latency memory. This has an important implication: SML programs are executing too many instruction.

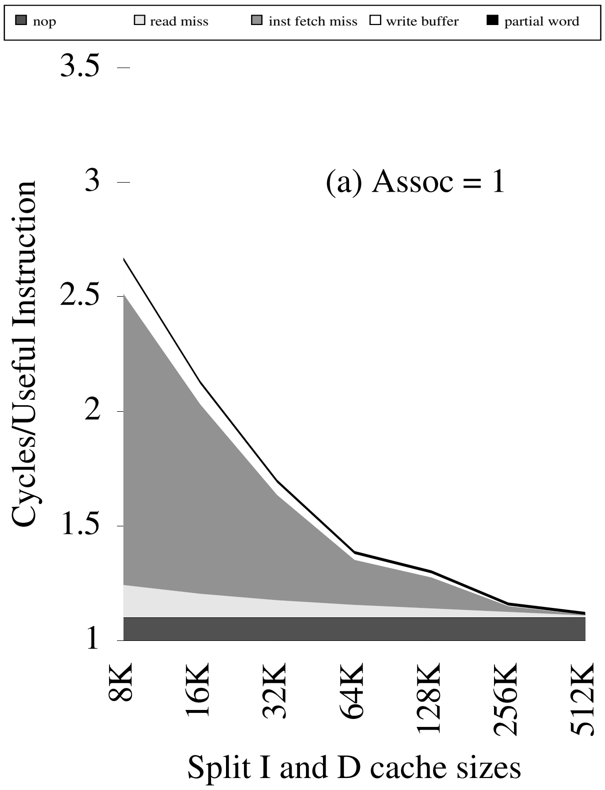

Figure 1.1: Breakdown of memory system performance for VLIW benchmark, with a memory-system with write allocation, subblock placement, and a cache block of 16

1.3.2 Example II: "Pay-as-you-go" compilation

In this section, to illustrate the "pay-as-you-go" compilation strategy, I show a dot product function as it passes through some of the stages of TIL. The dot product function is the inner loop of an integer matrix multiply benchmark. I give the complete version of this example, showing code as it passes through all of the stages of TIL, in Chapter 4 .

Figure 1.2 shows the original SML code for the dot product function, which uses `sub2`, a built-in 2-d array subscript operation. The front-end expands `sub2` to the code shown in Figure 1.3.

```
val sub2 : 'a array2 * int * int -> 'a

fun dot(cnt,sum) =
  if cnt<bound then
   let val sum'=sum+sub2(A,i,cnt)*sub2(B,cnt,j)
   in dot(cnt+1,sum')
   end
 else sum
```

Figure 1.2: Original SML code

```
fun sub2 ({columns,rows,v}, s :int, t:int) =
  if s <0 orelse s>=rows orelse t<0 orelse
  t>=columns then raise Subscript
  else unsafe_sub1(v,s * columns + t)
```

Figure 1.3: Built-in 2-d array subscript

Figure 1.4 shows the functions after they have been converted to a typed intermediate language and before TIL has done any optimization. Readers should feel free to skip this particular figure because it is difficult to read; I explain the details of the intermediate language in Chapter 4. The point of this figure is that the quality of code at this stage in compilation is poor: executing the body of `dot` results in eight function applications, four record constructions, and numerous checks for array bounds.

Figure 1.5 shows the code for the `dot` function after it has been optimized. The body of the loop consists of 9 expressions: the optimizer has eliminated all of the function applications and also safely eliminated the comparisons for array bounds checking. `sub_ai(av,a)` is an application of the (unsafe) integer array subscript primitive. This function could be improved even further by applying two more "loop" optimizations (strength reduction and induction variable elimination).

Figure 1.6 shows the actual DEC ALPHA assembly language that TIL generates for the `dot` function. The code between `L1` and `L3` corresponds to the body of `dot`. The other code

```
sub2 =
   let fix f = Λty.
        let fix g = λarg.
            let a = (#0 arg)
                s = (#1 arg)
                t = (#2 arg)
                columns = (#0 a)
                rows = (#1 a)
                v = (#2 a)
                check =
                  let test1 = lti(s,0)
                  in Switch_enum test of
                       1 => λ.enum(1)
                     | 0 => λ.
                        let test2 = gti(s,rows)
                        in Switch_enum test2 of
                             1 => λ.enum(1)
                           | 0 => λ.
                             let test3 = lti(t,0)
                             in  Switch_enum test3 of
                                 1 => λ.enum(1)
                               | 0 => λ.gti(t,columns)
                             end
                        end
                  end
            in Switch_enum check of
                  1 => λ.raise Subscript
                | 0 => λ.unsafe_sub1 [ty] {v,t + s * columns}
            end
        in g
   in f
   end

fix dot=
    λi.let cnt = (#0 i)
            sum = (#1 i)
            d = lti(cnt,bound)
        in  Switch_enum d
            of 1 => λ.let sum' = sum +
                               ((sub2 [Int]) {A,i,cnt}) *
                              ((sub2 [Int]) {B,cnt,j})
                           in  dot{cnt+1,sum'}
                           end
             |  0 => λ.sum
        end
```

Figure 1.4: After conversion to a typed intermediate language

```
fix dot =
    λcnt,sum.
     let test = lti(cnt,bound)
         r = Switch_enum test of
                1 =>
                λ.
                 let a = t1 + cnt
                     b = sub_ai(av,a)
                     c = columns * cnt
                     d = j + c
                     e = sub_ai(bv,d)
                     f = b*e
                     g = sum+f
                     h = 1+cnt
                     i = dot(h,g)
                 in  i
                 end
             |  0 => λ.sum
     in  r
     end
```

Figure 1.5: After optimization

is epilogue and prologue code for entering and exiting the function. Note that no tagging operations or garbage-collection related operations occur anywhere in this code.

1.3.3 Example III: Comparison to SML/NJ

Figure 1.7 compares the execution time of programs compiled by TIL to the execution time of programs compiled by SML/NJ on a DEC ALPHA AXP 3000/250 workstation. This workstation uses an Alpha processor with a clock speed of 250 Mhz that issues up to 4 instructions per clock cycle. In this measurement, whole programs were compiled. I describe the benchmark programs and details of the measurements in Chapter 8.

The 100% mark on the graph represents the execution time of programs compiled by SML/NJ. The bars show the relative execution time of programs compiled by TIL.

1.4 Related work

Kranz *et al.* [52, 51] show that Scheme versions of Pascal programs can be compiled to be as efficient as the original Pascal programs. I extend this result by showing that it is often possible to compile programs that use higher-order functions and polymorphism, language features not found in Pascal, to machine code that is similar to that produced for Pascal or C programs.

```
        .ent Lv2851_dot_205955
 # arguments : [$bound,$0] [$columns,$1] [$bv,$2]
 #             [$av,$3] [$t1,$4] [$j,$5]
 #             [$cnt,$6] [$sum,$7]
 # results    : [$result,$0]
 # return addr  : [$retreg,$26]
 # destroys   : $0 $1 $2 $3 $4 $5 $6 $7 $27
Lv2851_dot_205955:
        .mask (1 << 26), -32
        .frame $sp, 32, $26
        .prologue 1
        ldgp       $gp, ($27)
        lda       $sp, -32($sp)
        stq       $26, ($sp)
        stq       $8, 8($sp)
        stq       $9, 16($sp)
        mov       $26, $27
L1:
        cmplt       $6, $0, $8
        bne       $8, L2
        mov       $7, $1
        br       $31, L3
L2:
        addl       $4, $6, $8
        s4addl       $8, $3, $8
        ldl       $8, ($8)
        mull       $1, $6, $9
        addl       $5, $9, $9
        s4addl       $9, $2, $9
        ldl       $9, ($9)
        mullv       $8, $9, $8
        addlv       $7, $8, $7
        addlv       $6, 1, $6
        trapb
        br       $31, L1
L3:
        mov       $1, $0
        mov       $27, $26
        ldq       $8, 8($sp)
        ldq       $9, 16($sp)
        lda       $sp, 32($sp)
        ret       $31, ($26), 1
        .end Lv2851_dot_205955
```

Figure 1.6: Actual DEC ALPHA assembly language

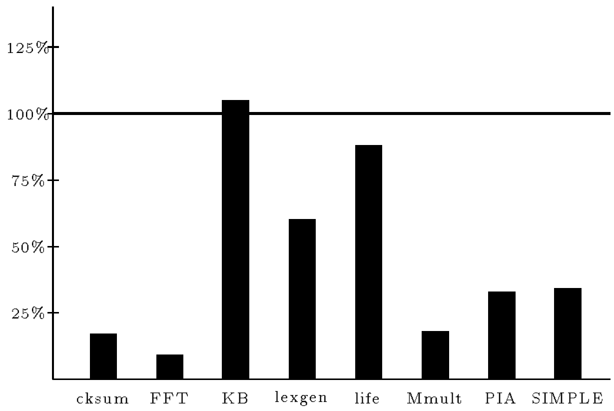

Figure 1.7: TIL Execution Time Relative to SML/NJ

Shivers [82, 83] argues for the importance of loop optimizations when compiling languages with higher-order functions, but proposes an approach based on control-flow analysis of higher-order functions. I show that control-flow analysis of higher-order functions is not needed to apply these optimizations to λ-calculus based programs. Inlining suffices to eliminate many higher-order functions and make most of the control-flow graph known at compile time. Furthermore, I show that simple local transformations suffice to produce good code for recursive functions.

Appel [7] shows that optimizations, especially inlining and uncurrying, are important to compiling SML programs. However, he does not argue for the importance of applying optimizations known to improve loops in conventional language to recursive functions in SML.

Leroy [56] shows that using untagged integers significantly increases performance. However, in his approach integers must still be boxed when stored in arrays or recursive data structures. With the TIL approach, integers are always unboxed and untagged. His approach also slows down programs that use polymorphic functions. I show that inlining polymorphic functions avoids this problem for a range of SML programs.

1.5 Organization

The rest of the thesis is organized into the following chapters:

- Chapter 2 studies the memory-system performance of SML programs.
- Chapter 3 studies the cost of automatic storage management.
- Chapter 4 describes the TIL framework.
- Chapter 5 describes the TIL optimizer.
- Chapter 6 describes new algorithms that show how to apply several "loop" optimizations to SML programs.

- Chapter 7 discusses how TIL compiles the intermediate programs produced by optimization to machine code.
- Chapter 8 compares the performance of TIL and the SML/NJ compiler to establish that TIL produces good code.
- Chapter 9 explores the effect on performance of the several "loop" optimizations that I describe in Chapter 6.
- Chapter 10 measures the effect of optimization on the number of higher-order and polymorphic functions when programs are compiled as a whole. It also measures the effect of types on intermediate program size.
- Chapter 11 discusses future directions.
- Chapter 12 concludes.

Part I

Measurement

In this part of the thesis, I examine two conjectures about where SML programs spend their time. The first conjecture is that processors executing SML programs spend much of their time waiting for the memory system. There are two reasons to believe that this may be true. First, Appel conjectures in his book on SML compilation [7] that SML programs spend up to two-thirds of their time waiting for the memory system. Second, Appel also shows that SML programs make intensive use of heap allocation — allocating roughly one word every five instructions. Many researchers have claimed that heap allocation leads to poor memory-system performance [50, 68, 96, 97, 99]. Together, these results indicate that memory-system performance of SML programs may be problem.

The second conjecture is that SML programs spend a lot of time doing automatic storage management. The cost of automatic storage management is important because automatic storage management is crucial for SML; it is needed for type safety.

These conjectures are difficult to examine carefully because they are *indirect* conjectures about program performance. Running SML programs and measuring execution times gives us little insight into whether these conjectures are true. It is, however, important to check them, because they could account for well over half the execution time of SML programs.

To examine these conjectures, I use trace-driven simulation of SML programs compiled by the SML/NJ compiler. In trace-driven simulation, you instrument programs to produce a trace of the instruction and data memory references made by the programs. You can then use these traces to simulate various memory systems. You can also analyze the traces on an instruction-by-instruction basis to determine precisely how many instructions programs execute on behalf of automatic storage management. Essentially, trace-driven simulation provides us with an idealized world in which we can control and measure all aspects of program execution.

I chose to use the SML/NJ compiler because it was the only full implementation of SML whose source code was available to researchers. I needed a full implementation of SML so that I could use reasonably large SML programs as benchmarks. I needed access to the compiler source code so that I could modify the compiler to produce traces of instruction and data references.

I have organized this part of the thesis in the following manner. In Chapter 2, I examine the memory-system performance of SML programs. I examine memory-system performance first because it appears to account for a larger part of execution time than automatic storage management. In Chapter 3, I examine the cost of automatic storage management for SML programs.

Chapter 2

Memory-system performance of SML programs

In this chapter, I study the conjecture that processors executing SML programs spend much of their time waiting for the memory system. This is joint work with Amer Diwan and Eliot Moss: Amer, Eliot, and I designed the experiment. I conducted many of the measurements and Amer modified the SML/NJ compiler [7], QPT [12, 53, 54], and Tycho [42] so that we could instrument SML programs and simulate memory-system performance. Amer also conducted some of the measurements.

As I mentioned earlier, I use trace-driven simulation to study performance: I instrument SML programs to produce traces of all memory references and feed the references into a memory-system simulator that calculates a performance penalty due to the memory system. To understand the memory-system performance of SML programs in general, I fix the architecture to be a prototypical RISC — the MIPS R3000 [47] — and vary the memory-system configurations to cover the design space typical of workstations of the late 1980's, such as DECStations, SPARCStations, and HP 9000 series 700. I study eight substantial programs.

I vary the following memory-system parameters: cache size (8K to 512K), cache-block size (16 or 32 bytes), write-miss policy (write allocate or write no-allocate), subblock placement (with and without), associativity (1 and 2 way), TLB sizes (1 to 64 entries), write-buffer depth (1 to 6 deep), and page-mode writes (with and without). I simulate only split instruction and data caches, *i.e.*, no unified caches. I report data only for write-through caches but it is easy to to extend the results to write-back caches.

I find two main results about the memory-system performance of SML programs. First, surprisingly, the memory-system performance of SML programs is good for some memory system configurations corresponding to actual machines. For example, for the memory-system configuration corresponding to the DECStation 5000/200, the memory-system performance of SML programs is comparable to that of C and Fortran programs [19]: programs run only 3 to 13% slower due to data-cache misses than they would run with a zero-latency memory. Second, heap allocation has a dominant effect on the memory-system performance of SML programs. For memory-system configurations that do not support heap allocation well, the slowdown due to data-cache misses is often higher than 50%. I discuss the implications of these results for compiling SML programs in the conclusion of this chapter.

I organize this chapter in the following manner: Section 2.1 gives background information, Section 2.4 describes related work, Section 2.2 describes the simulation methods, the benchmarks, and the memory-system performance metrics that I used, Section 2.3 presents the simulation results, analyses them, validates them, and gives an analytical model that extends them to programs that do less heap allocation, Section 2.4 describes related work, and Section 2.5 concludes.

2.1 Background

The following sections describe memory systems, garbage collection in SML/NJ, SML, and the SML/NJ compiler.

2.1.1 Memory systems

This section describes cache organization for a single level of caching. A cache is divided into *blocks* that are grouped into *sets*. Main memory is also divided into blocks, which typically have the same size as cache blocks. A main memory block may reside in the cache in exactly one set, but may reside in any block within the set. A cache with sets of size n is said to be *n-way associative*. If $n=1$, the cache is called *direct-mapped*. Some caches have valid bits that indicate what sections of a block hold valid data. Each section that has a valid bit associated with it is called a *subblock*. In this thesis, *subblock placement* implies a subblock of one word, *i.e.*, each word has a valid bit. Moreover, on a read miss, the whole block is brought into the cache, not just the missing word. Przybylski [71] notes that this is a good choice.

A memory access to a location that is resident in the cache is a *hit*. Otherwise, the memory access is a *miss*. A miss is a *compulsory miss* if it is due to a memory block being accessed for the first time. A miss is a *capacity miss* if it results from the cache not being large enough to hold all the memory blocks used by a program. It is a *conflict miss* if it results from two memory blocks mapping to the same set [41].

A read miss is handled by copying the missing block from main memory to the cache. A write hit is always written to the cache. There are several policies for handling a write miss, which differ in their performance penalties. For each of the policies, the actions taken on a write miss are:

1. write-no-allocate:
   - Do not allocate a block in the cache
   - Send the write to main memory without putting the write in the cache.
2. write-allocate, no-subblock placement:
   - Allocate a block in the cache.
   - Fetch the corresponding memory block from main memory.
   - Write the word to the cache (and to memory if write through).

3. write-allocate, subblock placement:
   If the tag matches and the block is in the cache, but the valid bit for the particular word in the block is off:

   - Write the word to the cache (and to memory if write through).

   If the tag does not match:

   - Allocate a block in the cache.
   - Write the word to the cache (and to memory if write through).
   - Invalidate the remaining words in the block.

Write allocate/subblock placement has a lower write-miss penalty than *write-allocate/no subblock placement* because it avoids fetching a memory block from main memory. In addition, it has a lower penalty than *write no-allocate* if the written word is read before being evicted from the cache. See Jouppi [45] for more information on write-miss policies.

A *write buffer* reduces the cost of writes to main memory. A write buffer is a queue containing writes that are to be sent to main memory. When the CPU does a write, it places the write in the write buffer and it continues without waiting for the write to finish. The write buffer retires entries to main memory using free memory cycles. The write buffer cannot always prevent stalls on writes to main memory. First, if the CPU writes to a full write buffer, the CPU must wait for an entry to become available in the write buffer. Second, if the CPU reads a location that is queued in the write buffer, the CPU may need to wait until the write buffer is empty. Third, if the CPU issues a read to main memory while a write is in progress, the CPU must wait for the write to finish.

Main memory is divided into DRAM pages. *Page-mode writes* reduce the latency of writes to the same DRAM page when there are no intervening memory accesses to another DRAM page [66]. For example, on a DECStation 5000/200, a non-page-mode write takes 5 cycles and a page-mode write takes 1 cycle. Page-mode writes are especially effective at handling writes with high spatial locality, such as writes that initialize consecutive memory locations.

2.1.2 Memory system performance

This section describes two metrics for measuring the performance of memory systems. One metric is the *cache miss ratio*. The cache miss ratio is the number of memory accesses that miss divided by the total number of memory accesses. Because different kinds of memory accesses usually have different miss costs, it is useful to have miss ratios for each kind of access.

Cache miss ratios alone do not measure the effect of the memory system on overall system performance. A metric that measures this better is the contribution of the memory system to cycles per useful instruction (CPI); all instructions besides nops (software-controlled pipeline stalls) are considered useful. CPI is calculated for a program as *number of CPU cycles to complete the program / total number of useful instructions executed*. It measures how efficiently the CPU is being utilized. The contribution of the memory system to CPI is

```
cmp alloc+12,top          ; Check for heap overflow
branch-if-gt call-gc
store tag,(alloc)         ; Store tag
store ra,4(alloc)         ; Store value
store rd,8(alloc)         ; Store pointer to next cell
move alloc+4,result       ; Save pointer to cell
add alloc,12              ; Increment allocation pointer
```

Figure 2.1: Pseudo-assembly code for allocating a list cell

calculated as *number of CPU cycles spent waiting for the memory system / total number of useful instructions executed.* As an example, on a DECStation 5000/200, the lowest CPI possible is 1, completing one instruction per cycle. If the CPI for a program is 1.50, and the memory contribution to CPI is 0.3, 20% (0.3/1.5) of the CPU cycles are spent waiting for the memory system (the rest may be due to other causes such as nops, multi-cycle instructions such as integer division, etc.). CPI is machine dependent because it is calculated using actual penalties.

2.1.3 Copying garbage collection

A copying garbage collector [32, 20] reclaims an area of memory by copying all the live (non-garbage) data to another area of memory. The area can then be re-used. Because copying garbage collection reclaims memory in large contiguous areas, programs can allocate objects sequentially from such areas in a few instructions. Figure 2.1 gives an example of pseudo-assembly code for allocating a list cell. `ra` contains the value to be stored in the list cell, `rd` contains the pointer to the next list cell, `alloc` is the address of the next free word in the allocation area, and `top` contains the end of the allocation area.

2.1.4 Garbage collection in SML/NJ

I use version 0.91 of the SML/NJ compiler, which uses a simple generational copying garbage collector [5]. Memory is divided into an old generation and an allocation area. New objects are created in the allocation area; garbage collection copies the live objects in the allocation area to the old generation, freeing the allocation area. Generational garbage collection relies on the fact that most allocated objects die young. Thus, most objects (about 99% [7, p. 206]) are not copied from the allocation area. This makes the garbage collector efficient because it works mostly on an area of memory where it is very effective at reclaiming space.

The most important property of a copying collector with respect to memory system behavior is that allocation sequentially initializes memory that has not been touched in a long time and is unlikely to be in the cache. This is especially true if the allocation area is large relative to the size of the cache because allocation cycles through the cache and eventually knocks everything out of the cache. This means that there will be a large number of write misses when the allocation area is larger than the cache.

For example, consider the code in Figure 2.1 for allocating a list cell. Assume that a cache write miss costs 16 CPU cycles and that the block size is 4 words. On average, every fourth write causes a write miss. Thus, the average memory system cost of allocating a word on the heap is 4 cycles. The average cost for allocating a list cell is seven cycles (at one cycle per instruction) plus 12 cycles for the memory system overhead. Thus, although allocation is cheap in terms of instruction counts, it may be expensive in terms of machine cycle counts.

2.1.5 Standard ML

Standard ML (SML) [61] is a call-by-value, lexically scoped language with higher-order functions. SML encourages a non-imperative programming style. Variables cannot be altered once they are bound and by default data structures cannot be altered once they are created. The only kinds of assignable data structures are `ref` cells and arrays, which must be declared explicitly. The implications of this non-imperative programming style for compilation are clear: SML programs tend to do more allocation and copying than programs written in imperative languages.

2.1.6 SML/NJ compiler

The SML/NJ compiler [7] is a publicly available compiler for SML. As I mention earlier, I use version 0.91 of the compiler. The compilation strategy focuses on making memory allocation inexpensive and making function calls fast. Allocation is done in-line, except for the allocation of arrays. Optimizations done by the compiler include inlining, passing function arguments in registers, register targeting, constant folding, code hoisting, uncurrying, and instruction scheduling.

The most controversial design decision in the compiler is to allocate procedure activation records on the heap instead of the stack [4, 10]. In principle, the presence of higher-order functions means that procedure activation records must be allocated on the heap. With a suitable analysis, a stack can be used to store most activation records [52]. However, using only a heap simplifies the compiler, the run-time system [6], and the implementation of first-class continuations [40]. The decision to use only a heap is controversial because it greatly increases the amount of heap allocation, which is believed to cause poor memory system performance.

2.2 Methodology

I use trace-driven simulation to evaluate the memory-system performance of programs. For this technique to be useful, there must be an accurate simulation model and a good selection of benchmarks. Simulations that make simplifying assumptions about important aspects of the system being modeled can yield misleading results. Toy or unrepresentative benchmarks can be equally misleading. In this work, I devote much effort to addressing these issues.

Section 2.2.1 describes the trace generation and simulation tools. Section 2.2.2 states my assumptions and argues that they are reasonable. Section 2.2.3 describes and characterizes the benchmark programs that I use. Section 2.2.4 describes the metrics that I use to measure memory-system performance.

2.2.1 Tools

Amer Diwan extended QPT (Quick Program Profiler and Tracer) [54, 12, 53] to produce memory traces for SML/NJ programs. QPT rewrites an executable program to produce compressed trace information; QPT also produces a program-specific regeneration program that expands the compressed trace into a full trace. Because QPT operates on the executable program, it can trace the SML code and the garbage collector (which is written in C).

Amer Diwan also extended Tycho [42] for the memory system simulations. His extensions to Tycho include a write-buffer simulator.

I obtain allocation statistics by using an allocation profiler built into SML/NJ. The profiler instruments intermediate code to increment appropriate elements of a *count array* on every allocation. I extended this profiler to count the number of assignments.

2.2.2 Simplifications and Assumptions

I try to minimize assumptions that might reduce the validity of my simulations. This section describes the important assumptions that I make.

1. *Simulating write allocate/subblock placement with write allocate/no subblock placement.* Tycho does not simulate subblock placement so I approximate it by simulating *write allocate/no subblock* and ignoring the reads from memory that occur on a write miss. This can cause a small inaccuracy in the CPI numbers, as the following example illustrates.

   Suppose the cache-block size is 2 words, the subblock size is 1 word, that a program writes the first word in a memory block, and that the write misses. In subblock placement, the word will be written to the cache and the second word in the cache block will be invalidated. However, the simplified model will mark both words as valid after the write. If the program subsequently reads the second word, the read will incorrectly hit. Thus the CPI reported for caches with subblock placement can be less than the actual CPI. These incorrect hits, however, occur rarely because SML programs tend to do few assignments (see Section 2.2.3) and most writes are to sequential memory locations.

2. *Ignoring the effect of context switches and the effect of system calls*

   Context switches and system calls cause instructions and data from the SML programs to be evicted from the cache. I do not attempt to measure this effect.

3. *The simulations are driven by virtual addresses.* Some machines such as the SPARC-Station II have physically indexed caches, and will have different conflict misses than those reported here.

4. *Placing code in the text segment instead of the heap.* This improves performance over the unmodified SML/NJ system. It reduces garbage-collection costs by never copying code and by avoiding the need for instruction-cache flushes after garbage collections.

5. *Used default compilation settings for SML/NJ.* Default compilation settings enable extensive optimization (Section 2.1.6). Evaluating the impact of these optimizations on cache behavior is beyond the scope of this thesis.

6. *Used default garbage-collection settings.*

   The preferred ratio of heap size to live data is set to 5 [5]. The softmax, which is the desired upper limit on the heap size, is set to 20MB; the benchmark programs never reach this limit. The initial heap size is 1MB.

   I do not investigate the interaction of the sizing strategy and cache size. Understanding these tradeoffs is beyond the scope of this thesis.

7. *All the traces are for the DECStation 5000/200, which uses a MIPS R3000 CPU [47].*

8. *All instructions take one cycle with a perfect memory system.*

   This affects only write buffer costs because multi-cycle instructions give the write buffer more time to retire writes. The effect of this assumption is negligible; Section 2.3.4 shows that write-buffer costs are already small.

9. *CPU cycle time does not vary with memory organization.* This may not be true because the CPU cycle time depends on the cache access time, which may differ across cache organizations. For example, a 128K cache may take longer to access than an 8K cache.

2.2.3 Benchmarks

Table 2.1 describes the benchmark programs. *Knuth-Bendix, Lexgen, Life, Simple, VLIW*, and *YACC* are identical to the benchmarks that Appel [7] measures. The description of these benchmarks is taken from [7]. Table 2.2 gives the following for each benchmark: lines of SML code excluding comments and empty lines, maximum heap size, compiled code size, and user-mode CPU time on a DECStation 5000/200. The code size includes 207 Kbytes for standard libraries, but does not include the garbage collector and other run-time support code, which is about 60 Kbytes. The run times are the minimum of five runs.

Table 2.3 characterizes the memory references of each program. For each program, it reports all numbers as a percentage of the total number of instructions executed by the program. The *Reads*, *Writes*, and *Partial writes* columns list the reads, full-word writes, and partial-word writes done by the program *and* the garbage collector; the *assignments* column lists the non-initializing writes done by the program only. The *Nops* column lists the nops executed by the program and the garbage collector. All the benchmarks have long traces; most related works (described in Section 2.4) use traces that are an order of magnitude smaller. Also, the benchmark programs do few assignments; the majority of the writes are initializing writes.

Program	Description
CW	The Concurrency Workbench [22] is a tool for analyzing networks of finite state processes expressed in Milner's Calculus of Communicating Systems. The input is the sample session from Section 7.5 of [22].
Knuth-Bendix	An implementation of the Knuth-Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.
Lexgen	A lexical-analyzer generator, implemented by James S. Mattson and David R. Tarditi [11], processing the lexical description of Standard ML.
Life	The game of Life, written by Chris Reade [72], running 50 generations of a glider gun. It is implemented using lists.
PIA	The Perspective Inversion Algorithm [94] decides the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [23], translated into ID [31], and then translated into Standard ML by Lal George.
VLIW	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
YACC	A LALR(1) parser generator, implemented by David R. Tarditi [89], processing the grammar of Standard ML.

Table 2.1: Benchmark Programs

	Size			Run time	
Program	Lines	Heap size (Kbytes)	Code size (Kbytes)	Non-gc (sec)	Gc (sec)
CW	5728	1107	894	22.74	3.09
Knuth-Bendix	491	2768	251	13.47	1.48
Lexgen	1224	2162	305	15.07	1.06
Life	111	1026	221	16.97	0.19
PIA	1454	1025	291	6.07	0.34
Simple	999	11571	314	25.58	4.23
VLIW	3207	1088	486	23.70	1.91
YACC	5751	1632	580	4.60	1.98

Table 2.2: Sizes of Benchmark Programs

Program	Inst Fetches	Reads (%)	Writes (%)	Partial Writes (%)	Assignments (%)	Nops (%)
CW	523,245,987	17.61	11.61	0.01	0.41	13.24
Knuth-Bendix	312,086,438	19.66	22.31	0.00	0.00	5.92
Lexgen	328,422,283	16.08	10.44	0.20	0.21	12.33
Life	413,536,662	12.18	9.26	0.00	0.00	15.45
PIA	122,215,151	25.27	16.50	0.00	0.00	8.39
Simple	604,611,016	23.86	14.06	0.00	0.05	7.58
VLIW	399,812,033	17.89	15.99	0.10	0.77	9.04
YACC	133,043,324	18.49	14.66	0.32	0.38	11.14

Table 2.3: Characteristics of benchmark programs

	Allocation	Escaping		Known		Callee Saved		Records		Other	
Program	(words)	%	Size	%	Size	%	Size	%	Size	%	Size
CW	56,467,440	4.0	4.12	3.3	15.39	67.2	6.20	19.5	3.01	6.0	4.00
Knuth-Bendix	67,733,930	37.6	6.60	0.1	15.22	49.5	4.90	12.7	3.00	0.1	15.05
Lexgen	33,046,349	3.4	6.20	5.4	12.96	72.7	6.40	15.1	3.00	3.7	6.97
Life	37,840,681	0.2	3.45	0.0	15.00	77.8	5.52	22.2	3.00	0.0	10.29
PIA	18,841,256	0.4	5.56	28.0	11.99	25.0	4.69	12.7	3.41	33.9	3.22
Simple	80,761,644	4.0	5.70	1.1	15.33	68.1	6.43	8.3	3.00	18.5	3.41
VLIW	59,497,132	9.9	5.22	6.0	26.62	61.8	7.67	20.3	3.01	2.1	2.60
YACC	17,015,250	2.3	4.83	15.3	15.35	54.8	7.44	23.7	3.04	4.0	10.22

Table 2.4: Allocation characteristics of benchmark programs

Table 2.4 gives the allocation statistics for each benchmark program. All allocation and sizes are in words. The *Allocation* column lists the total allocation done by the benchmark. The remaining columns break down the allocation by kind: closures for escaping functions, closures for known functions, closures for callee-save continuations, records, and others (includes spill records, arrays, strings, vectors, ref cells, store list records, and floating-point numbers). Closures for callee-save continuations correspond roughly to procedure-activation records. They can be trivially allocated on a stack in the absence of first-class continuations. A function is a known function if all of its call sites are known. Otherwise, it is an escaping function. For each kind of allocated object, the *%* column gives the total words allocated for objects of that kind as a percentage of total allocation and the *Size* column gives the average size in words, including the 1 word tag, of an object of that kind.

2.2.4 Metrics

I state cache performance numbers in *cycles per useful instruction* (*CPI*). I consider all instructions except for null operations to be useful. Null operations provide software-controlled pipeline stalls on the MIPS R3000.

Table 2.5 lists the timings for memory operations that I use in the simulations. I derive these numbers from the penalties for the DECStation 5000/200; these penalties are similar

Task	Timings (cycles)
Non-page-mode write	5
Page-mode write	1
Partial-word write	11
Page-mode flush	4
Read 16 bytes from memory	15
Read 32 bytes from memory	19
Refresh period	195
Refresh time	5
Write hit or miss (subblocks)	0
Write hit (16 bytes, no subblocks)	0
Write hit (32 bytes, no subblocks)	0
Write miss (16 bytes, no subblocks)	15
Write miss (32 bytes, no subblocks)	19
TLB miss	28

Table 2.5: Timings of memory operations

to those of other machines of the same class. In addition to the times in Table 2.5, all reads and writes may also incur write-buffer penalties. In an actual implementation, there may be a one cycle penalty for write misses in caches with subblock placement. This is because a tag needs to be written to the cache after the miss is detected. This does not change my results, because it adds at most 0.02–0.05 to the CPI that I report for caches with subblock placement.

I use a DRAM page size of 4K in the simulation of page-mode writes. Page-mode flush is the number of cycles needed to flush the write pipeline after a series of page-mode writes.

I report TLB data as the TLB miss contribution to the CPI. I use a virtual memory page size of 4K in my simulations.

2.3 Results and Analysis

Section 2.3.1 presents a qualitative analysis of the memory behavior of programs compiled with SML/NJ. Section 2.3.2 lists the cache and TLB configurations that I simulate and explains why I selected them. Sections 2.3.3, 2.3.4, and 2.3.5 present data for memory-system performance, write-buffer performance, and TLB performance. Section 2.3.6 validates the simulations. Section 2.3.7 presents an analytical model that extends these results to programs with different allocation behavior.

Write Policy	Write Miss Policy	Subblocks	Assoc	Block Size	Cache Sizes	Write Buffer	Page mode
through	allocate	yes	1, 2	16, 32 bytes	8K–512K	1–6 deep	yes
through	allocate	no	1, 2	16, 32 bytes	8K–512K	6 deep	no
through	no allocate	no	1, 2	16, 32 bytes	8K–512K	6 deep	no

Table 2.6: Memory-system organizations studied

2.3.1 Qualitative Analysis

Recall from Section 2.1 that SML/NJ uses a copying collector. The most important property of a copying collector with respect to memory-system behavior is that allocation initializes memory in an area that has not been touched since the last garbage collection. This means that for caches that are not large enough to contain the allocation area there will be many write misses. The slowdown that these write misses translate to depends on the memory-system organization.

Recall from Section 2.2.3 that SML/NJ programs have the following important properties. First, they do few assignments: the majority of the writes are initializing writes. Second, programs do heap allocation at a furious rate: 0.1 to 0.22 words per instruction. Third, writes come in bunches because they correspond to initialization of a newly-allocated area.

The burstiness of writes combined with the property of copying collectors mentioned above suggests that an aggressive write policy is needed to achieve good performance. In particular, writes should not stall the CPU. Memory-system organizations where the CPU has to wait for a write to be written through (or back) to memory will perform poorly. Even memory systems where the CPU does not need to wait for writes if they are issued far apart (e.g., 2 cycles apart in the HP 9000 series 700) may perform poorly due to the bunching of writes. The means that the memory system needs two features. First, a write buffer or fast page-mode writes are essential to avoid waiting for writes to memory. Second, on a write miss, the memory system must avoid reading a cache block from memory if it will be written before being read. Of course, this requirement only holds for caches with a *write-allocate* policy. Subblock placement [50], a block size of 1 word, and the ALLOCATE instruction [68] can all achieve this. Because the effects on cache performance of these features are similar, I discuss only subblock placement. Note that large caches will reduce the benefit of subblock placement: the allocation area will fit in the cache and thus there will be few write misses.

2.3.2 Cache and TLB configurations simulated

The design space for memory systems is enormous. There are many variables involved and the dependencies between the variables are complex. Therefore I can study only a subset of the memory-system design space. I restrict this study to features found in RISC workstations of the late 1980's [29, 28, 84, 24]. Table 2.6 summarizes the cache organizations that I simulated. Table 2.7 lists the memory-system organizations of some popular machines.

I simulate only separate instruction and data caches (*i.e.,* no unified caches). Many machines have separate caches (e.g., DECStations, HP 700 series), but there are some exceptions

Architecture	Write Policy	Write Miss Policy	Write Buffer	Subblocks	Assoc	Block Size	Cache Size
DS3100	through	allocate	4 deep	—	1	4 bytes	64K
DS5000/200	through	allocate	6 deep	yes	1	16 bytes	64K
HP 9000	back	allocate	none	no	1	32 bytes	64K–2M
SPARCStation II	through	no allocate	4 deep	no	1	32 bytes	64K

Note:

- SPARCStations have unified caches.
- Most HP 9000 series 700 caches are much smaller than 2M: 128K instruction cache and 256K data cache for models 720 and 730, and 256K instruction cache and 256K data cache for model 750.
- The DS5000/200 actually has a block size of four bytes with a fetch size of sixteen bytes. This is stronger than subblock placement because every "subblock" has a full tag.

Table 2.7: Memory-system organization of some popular machines

(notably SPARCStations).

I report data only for write-through caches. It is easy to approximate the CPI for write-back caches from that data. Write-through and write-back caches have identical misses, but have different contributions to CPI for two reasons. First, their write-buffer costs differ because they write to main memory with different frequencies and at different points during program execution. Second, they have different costs for write hits and write misses. Write hits and write misses for a write-back cache may cost one cycle more than they do for a write-through cache. A write-back cache must probe the tag *before* writing to the cache [45], unlike a write-through cache.

The different write-buffer costs can be ignored. The write-buffer costs for a write-through cache are usually an upper bound on the costs for a comparable write-back cache because write-through caches write to main memory more often than write-back caches do. The write-buffer costs for the write-through caches that I measure are already small, so the difference between the write-buffer costs for comparable write-back and write-through caches is likely to be negligible.

Thus, to obtain the CPI for write-back caches from the CPI for write-through caches, we need to account only for the extra cycle required to probe the tag. This is easy to do: if a program does w writes and n useful instructions, the extra cycle adds w/n to the CPI. For the `VLIW` program, for example, w/n is 0.18.

I simulate fully associative, unified TLBs from 1 to 64 entries with an LRU replacement policy. Some machines, such as the HP 9000 series, have separate instruction and data TLBs. From Section 2.3.5 it is clear that for the benchmarks even small unified TLBs perform well.

Two of the most important cache parameters are *write allocate* versus *write no allocate* and *subblock placement* versus *no subblock placement*. Of these, the combination *write no allocate/subblock placement* placement offer no improvement over *write no allocate/no subblock placement* for cache performance. Thus, I present data for the *write no allocate/subblock placement* configuration.

2.3.3 Memory-System Performance

I present memory-system performance in summary graphs and breakdown graphs. Each summary graph shows the performance of one benchmark program for a range of cache sizes (8K to 512K), write-miss policies (write allocate or write no allocate), subblock placement (with or without), and associativity (1 or 2). Each curve in a summary graph corresponds to a different memory-system organization. There are two summary graphs for each program, one for a block size of 16 bytes and another for a block size of 32 bytes. Each breakdown graph breaks down the memory-system overhead into its components for one configuration in a summary graph. The write-buffer depth in these graphs is fixed at 6 entries.

In this section I present only the summary graphs for `VLIW` (Figure 2.2). The data for other programs is similar and I give it in Appendix A. Figures 2.3, 2.4, and 2.5 are the breakdown graphs for `VLIW` for the 16 byte block size configurations; the remaining breakdown graphs for `VLIW` are similar and I omit them for conciseness. The breakdown graphs for the other benchmarks are similar (and predictable from the summary graphs) and I omit them for the same reason.

In the summary graphs, the *nops* curve is the base CPI: the total number of instructions executed divided by the number of useful (not nop) instructions executed. This corresponds to the CPI for a perfect memory system. For the breakdown graphs, the *nop* area is the CPI contribution of nops; *read miss* is the CPI contribution of read misses; *write miss* is the CPI contribution of write misses (if any), *inst fetch miss* is the CPI contribution of instruction fetch misses; *write buffer* is the CPI contribution of the write buffer; *partial word* is the CPI contribution of partial-word writes.

The 64K point on the *write alloc, subblock, assoc=1* curves corresponds closely to the DECStation 5000/200 memory system.

In the following subsections I describe the effect of write-miss policy and subblock placement, associativity, block size, cache size, write buffer depth, and partial-word writes on the memory system performance of the benchmark programs. Even though CPI is a ratio, I summarize CPI improvements across the benchmarks using arithmetic means. The geometric means are similar (and in some cases identical) to the arithmetic means.

Write Miss Policy and Subblock Placement

From the summary graphs, it is clear that the best cache organization of the ones that I study is *write allocate/subblock placement*; it outperforms all other configurations substantially. For a memory system with a 64K direct-mapped write-allocate cache and 4 word blocks, subblock placement reduces the CPI by 0.35 to 0.88, with an arithmetic mean improvement of 0.55. Surprisingly, for sufficiently large caches with the *write allocate/subblock placement* organization, the memory system performance of SML/NJ programs is acceptable; the overhead due to data-cache misses ranges from 3% to 13% (arithmetic mean 9%) for 64K direct-mapped caches and 1% to 13% (arithmetic mean 9%) for 32K two-way associative caches. Recall that the 64K direct-mapped configuration corresponds to the DECStation 5000/200 memory system. The memory system overhead of SML/NJ programs on the DECStation 5000/200 is similar to that of C and Fortran programs [19]. It is worth emphasizing that the memory system performance of SML/NJ programs is *good* on some current machines *despite the very* *high miss rates*; for a 64K cache with a block size of 16 bytes, the write miss and read miss ratios for VLIW are 0.23 and 0.02 respectively.

Recall that in Section 2.3.1 I argue that the benefit of subblock placement will be substantial, but that the benefit will decrease for larger caches. The summary graphs indicate that the reduction in benefit is not substantial even for 128K cache sizes. The benefit of subblock placement, however, decreases sharply for larger caches for six of the benchmark programs. This suggests that the allocation area size of six of the benchmark programs is between 256K and 512K.

The performance of *write allocate/no subblock* is almost identical to that of *write no allocate/no subblock* (Knuth-Bendix is an exception). The difference between *write allocate/no subblock* and *write no allocate/no subblock* is so small in most graphs that the two curves overlap. This suggests that memory addresses are being read soon after being written. Even in an 8K cache, a memory address is read after being written before it is evicted from the cache (if it was evicted from the cache before being read, then *write allocate/no subblock* would have inferior performance). The only difference between these two policies is *when* a cache block is read from main memory. In one case, it is brought in on a write miss; in the other, it is brought in on a read miss. Because SML/NJ programs allocate sequentially and do few assignments, a newly allocated object remains in the cache until the program has allocated another C bytes, where C is the size of the cache. Because the programs allocate 0.4–0.9 bytes per instruction, my results suggest that a cache block is read within 9,000–20,000 instructions after it is written.

The benefit of subblock placement is not limited to functional languages such as Standard ML. Jouppi [45] reports that subblock placement combined with an 8K data cache and a 16 byte cache block size eliminates 31% of the cost of cache misses for C programs. Reinhold [73] finds that the memory performance of Scheme is good with subblock placement.

Changing Associativity

From Figure 2.2 we see that increasing associativity improves all organizations. The improvement in going from one-way to two-way set associativity is much smaller than the improvement obtained from subblock placement. For a memory system with a 64K write-allocate cache and 4 word blocks, increasing associativity reduces the CPI by 0.01 to 0.09, with an arithmetic mean improvement of 0.06. The maximum benefit from higher associativity is obtained for small cache sizes less than 16K. However, increasing associativity may increase CPU cycle time and thus the improvements may not be realized in practice [41].

From Figures 2.3, 2.4, and 2.5 we see that higher associativity improves the instruction-cache performance but has little or no effect on data-cache performance. Surprisingly, for direct-mapped caches (Figures 2.3 (a), 2.4 (a), and 2.5 (a)) the instruction-cache penalty is substantial for 128K or smaller caches. For caches with subblock placement, the instruction-cache penalty can dominate the penalty for the memory system. The improvement observed in going to a two-way associative cache suggests that a lot of the penalty from the instruction cache is due to conflict misses and that from the data cache is due to capacity misses. The data cache is simply not large enough to hold the working set. The performance of the instruction cache is not surprising given the benchmark programs. The programs use small functions and make frequent function calls, which lowers spatial locality. Thus, the chances of conflicts are greater than if the instructions had strong spatial locality.

Changing Block Size

From Figure 2.2 we see that increasing the block size from 16 to 32 bytes also improves performance. For a memory system with a 64K direct-mapped write-allocate cache, increasing the block size reduces the CPI by 0.14 to 0.35, with an arithmetic mean improvement of 0.22. For the *write allocate* organizations, doubling the block size can halve the write-miss rate. Thus, larger block sizes improve performance when there is a penalty for a write miss [50]. In contrast, for caches with *write allocate/subblock placement*, where there is no write miss penalty, increasing the block size improves performance only a little.

From Figure 2.2 we see that caches with *write no allocate* benefit just as much from larger block size as caches with *write allocate/no subblock placement*. This suggests that the spatial locality of reads is comparable to that of writes.

Note that subblock placement improves performance more than even two-way associativity and 32 byte blocks combined.

Changing Cache Size

There are three distinct regions of performance as the cache size varies. The first region corresponds to the range of cache sizes where the allocation area does not fit in the cache (i.e., allocation happens in an area of memory that is not cache resident). For most of the benchmarks, this region corresponds to cache sizes of less than 256K (for `Simple` and `Knuth-Bendix` this region extends beyond 512K). In this region, increasing the cache size uniformly improves performance for all configurations. However, the performance improvement from doubling the cache size is small.

From the breakdown graphs in Figures 2.3 through 2.4 we see that in the first region the cache size has little effect on the data-cache miss contribution to CPI. Most of the improvement in CPI that comes from increasing the cache size is due to improved performance of the instruction cache. As with associativity, cache sizes have interactions with the cycle time of the CPU: larger caches can take longer to access. Thus, small improvements due to increasing the cache size may not be achieved in practice.

The second region ranges from when the allocation area begins to fit in the cache until the allocation area fits in the cache. For most of the benchmarks (once again excepting `Simple` and `Knuth-Bendix`), this region corresponds to cache sizes in the range 256K to 512K[1]. In this region, increasing the cache size sharply improves the data-cache performance for memory organizations without subblock placement. However, increasing the cache size in this region has little effect on instruction-cache performance because the instruction-cache miss costs are already low at this point.

The third region corresponds to cache sizes where the allocation area fits in the cache. For five of the benchmarks, this region corresponds to caches larger than 512K (for `Lexgen`, `Knuth-Bendix`, and `Simple` this region starts at larger cache sizes). In this range, increasing

[1] For `Lexgen` this region extends a little beyond 512K.

the cache size has little or no effect on memory-system performance because everything remains cache resident and thus there are no capacity misses to eliminate.

Write Buffer and Partial-Word Write Overheads

From the breakdown graphs we see that the write buffer and partial-word write contributions to CPI are negligible. A six-deep write buffer coupled with page-mode writes is sufficient to absorb the bursty writes. As expected, memory-system features that reduce the number of misses (such as higher associativity and larger cache sizes) also reduce the write buffer overhead.

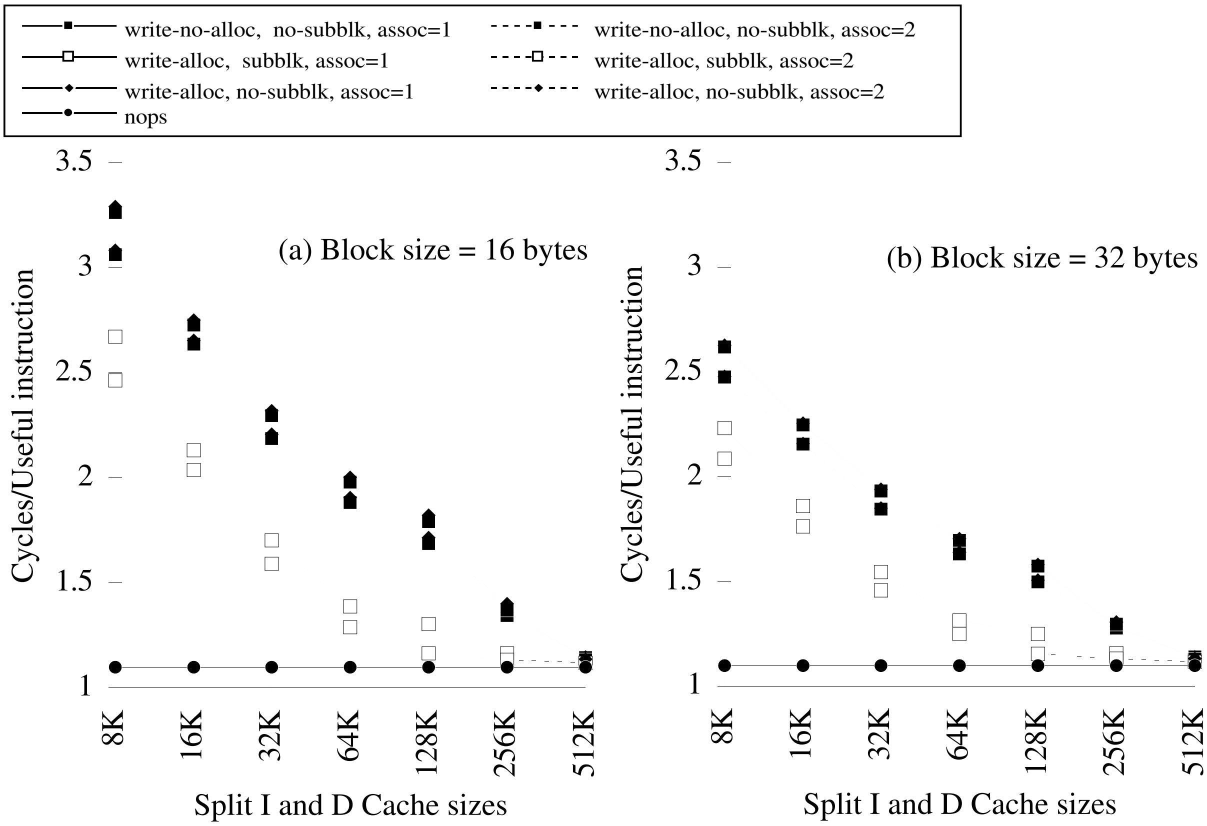

Figure 2.2: VLIW summary

2.3.4 Write-buffer depth

In Section 2.3.3 I showed that a six-deep write buffer coupled with page-mode writes is able to absorb the bursty writes of SML/NJ programs. In this section I explore the effect of write-buffer depth on the write-buffer stall contribution to CPI. Because the speed at which the write buffer can retire writes depends on whether or not the memory system has page-mode writes, I conduct two sets of experiments, one with and the other without page-mode writes. I vary the write-buffer depth from 1 to 6. I conduct this study for two of the larger

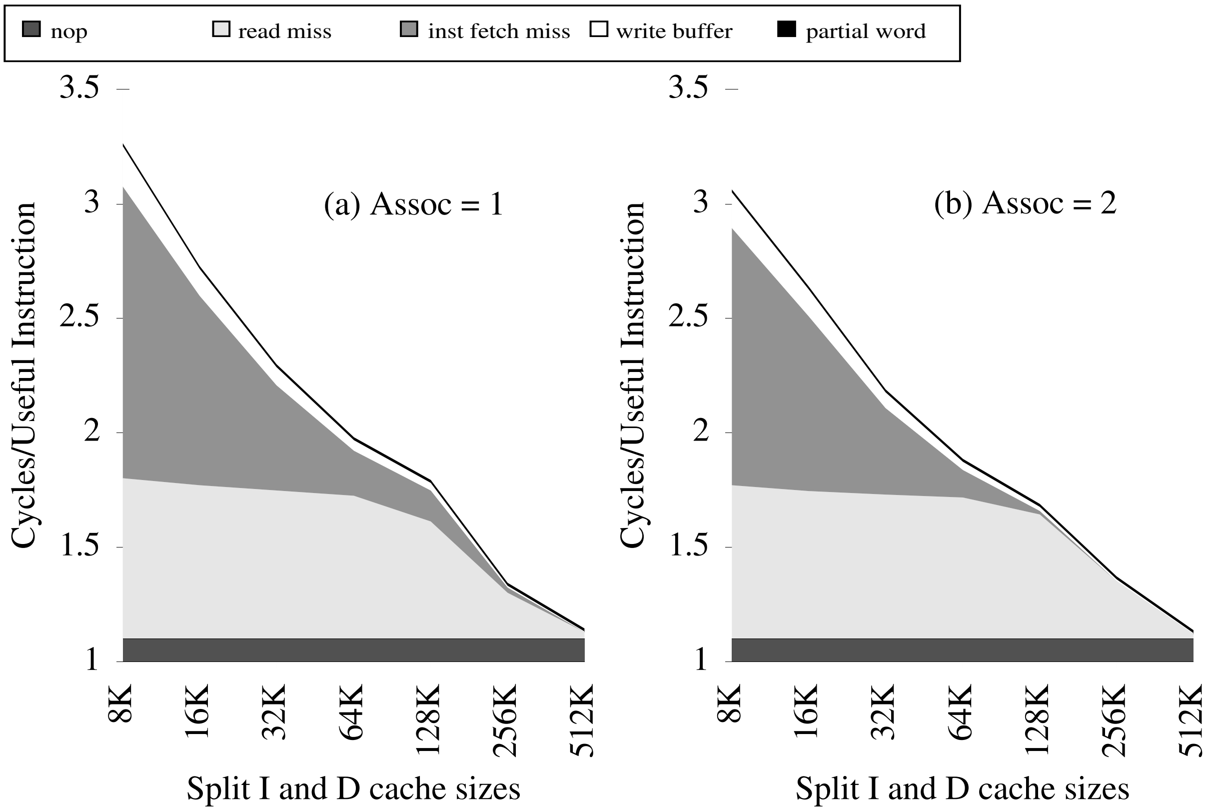

Figure 2.3: VLIW breakdown, write no alloc, no subblk, block size=16

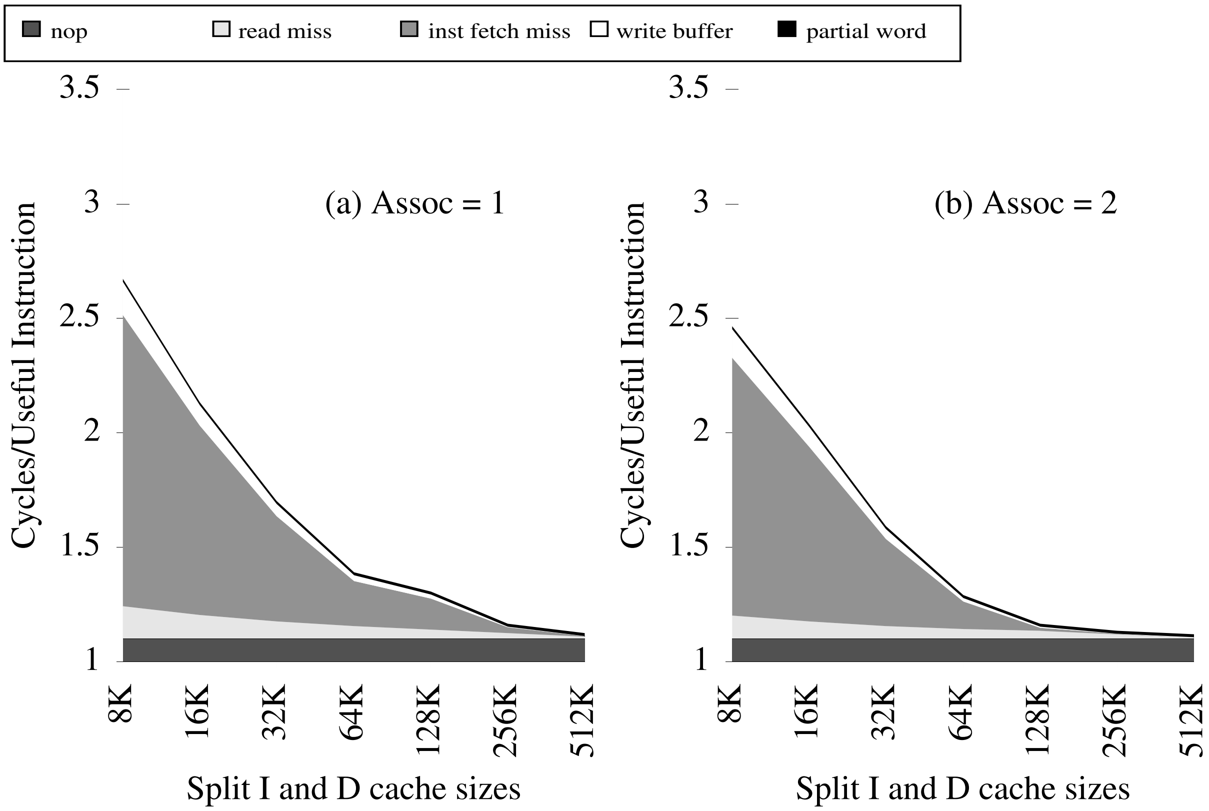

Figure 2.4: VLIW breakdown, write alloc, subblk, block size=16

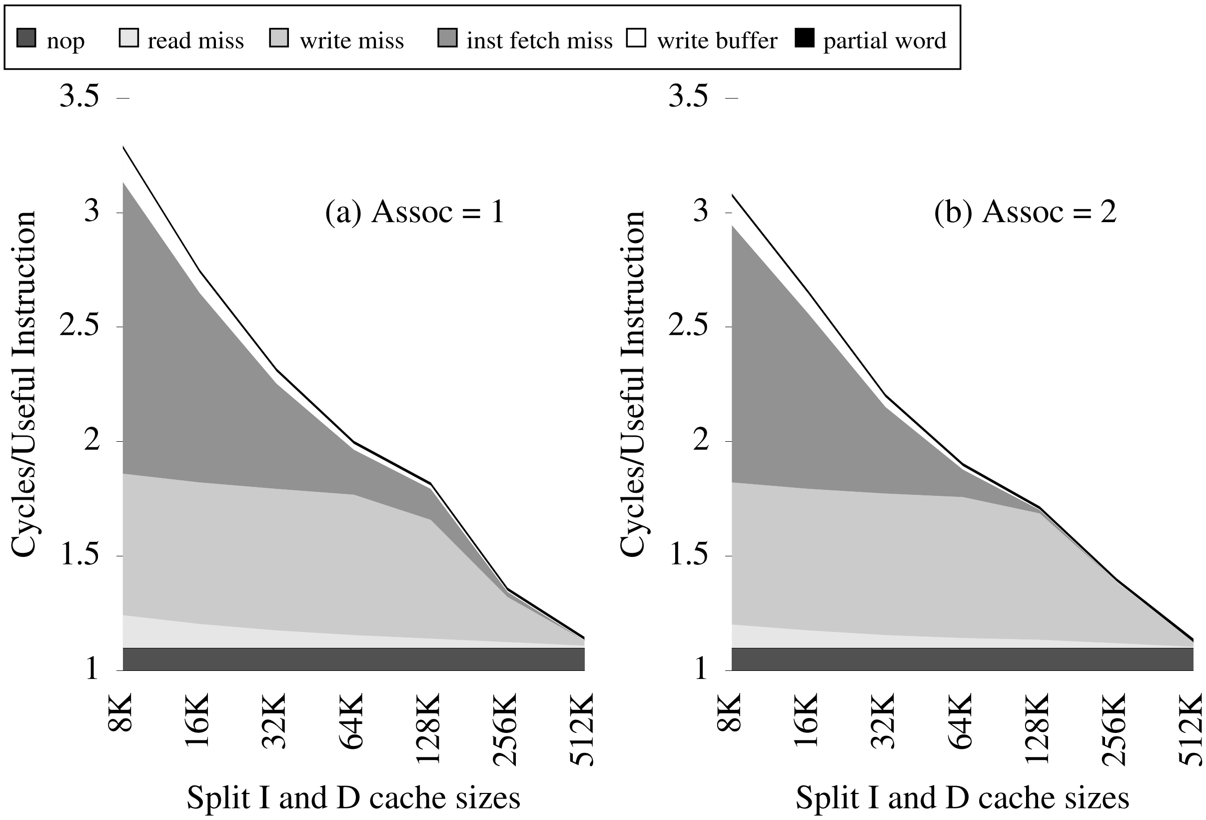

Figure 2.5: VLIW breakdown, write alloc, no subblk, block size=16

benchmarks: `CW` and `VLIW`. I fix the block size at 16 bytes and the write miss policy at *write allocate/subblock placement*.

Figure 2.6 gives the write-buffer costs for `VLIW` with caches of associativity one and two and in a memory system with page-mode writes; Figure 2.7 does the same in a memory system without page-mode writes. The graphs plot the CPI contribution of the write buffer against cache size; there is one curve for each write-buffer depth. Increasing the cache size or associativity reduces the number of read and instruction-fetch misses, and thus reduces the number of main-memory transactions. Reducing the number of main-memory transactions increases the effectiveness of the write buffer because the write buffer fills up less frequently and also has more cycles in which to retire writes (Section 2.1.1).

In memory systems with page-mode writes (Figure 2.6), the difference between the CPI contribution of a one-deep write buffer and a six-deep write buffer is less than 0.05. This is surprisingly small considering the burstiness of the writes and is due to the effectiveness of page-mode writes, as the following example illustrates.

Suppose that a program is allocating and initializing a 4-word object and that the write buffer is one-deep. Further, suppose that the write buffer is empty and that the instructions doing the allocation all hit in the instruction cache. The first write does not stall the CPU because the write buffer is empty. At the next write one cycle later, the write buffer is full and the CPU stalls. After 4 cycles (see Table 2.5), the write is placed in the write buffer. This write, however, is likely to be on the same DRAM page as the previous write because it is to the next address. It will therefore complete in one cycle. All subsequent writes to initialize this object find an empty write buffer because they also complete in one cycle due to page-mode writes.

Due to sequential allocation, it is likely that writes to initialize objects allocated one after another will also be on the same DRAM page. In the best case, with no read misses and refreshes, a *write buffer full* delay will happen only once per `N` words of allocation, where `N` is the size of the DRAM page. Thus, the write buffer depth has little effect on the performance of SML/NJ programs if the memory system has page-mode writes. To confirm this explanation, I measured the probability of two consecutive writes being on the same DRAM page. This probability averaged over the benchmarks is 96%.

The small effect of write-buffer depth on performance does not imply that a write buffer is useless if the memory system has page-mode writes. Instead, it says that a deep write buffer offers little performance improvement in a memory system with page-mode writes if the programs have *strong spatial locality* in their writes, and the majority of the reads and instruction fetches hit in the cache. *Strong spatial locality* means that the probability that two consecutive writes are to the same DRAM page is high.

Write-buffer depth is important, however, if the memory system does not have page-mode writes (Figure 2.7). In this case, a six-deep write buffer performs much better than a one-deep write buffer. Note that Figures 2.6 and 2.7 have different scales.

2.3.5 TLB Performance

Figure 2.8 gives the TLB miss contribution to the CPI for each benchmark program. We see that the CPI contribution of TLB misses falls below 0.01 for all our programs for a 64

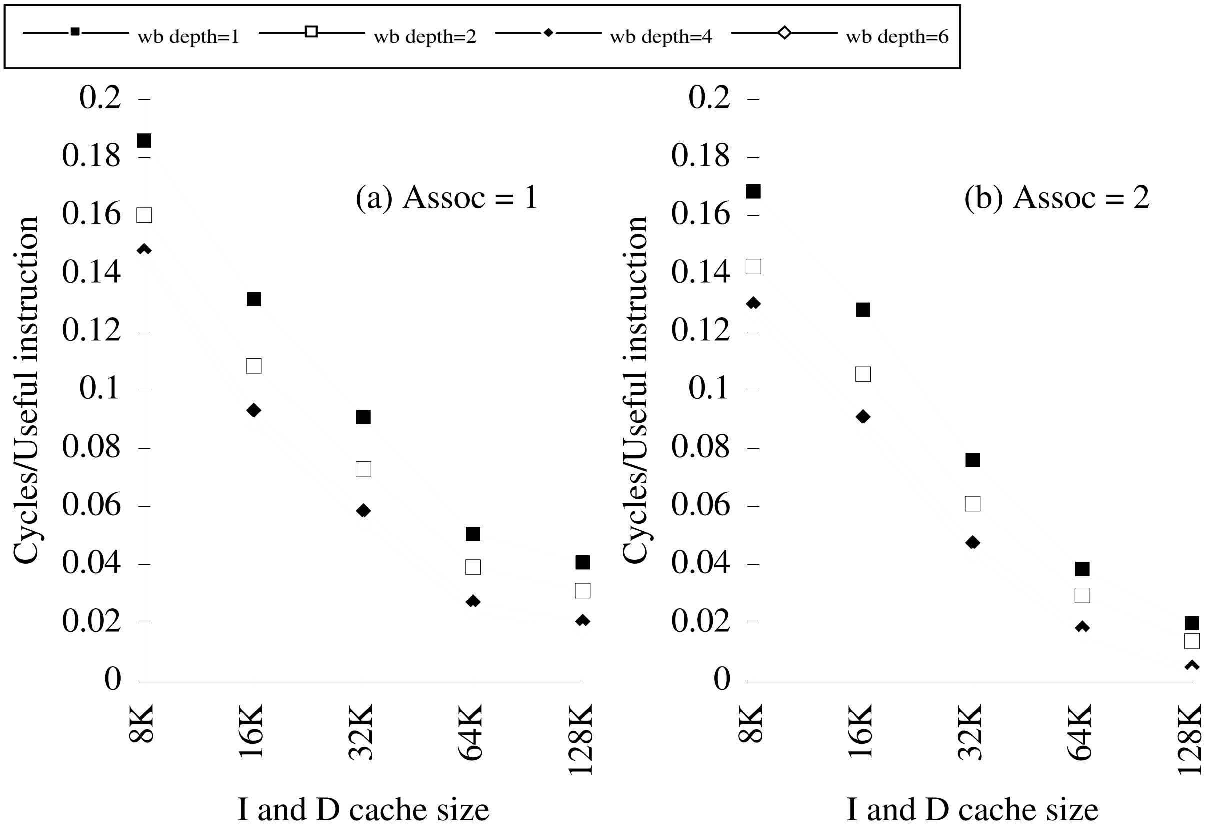

Figure 2.6: Write buffer CPI contribution for VLIW, With page-mode writes

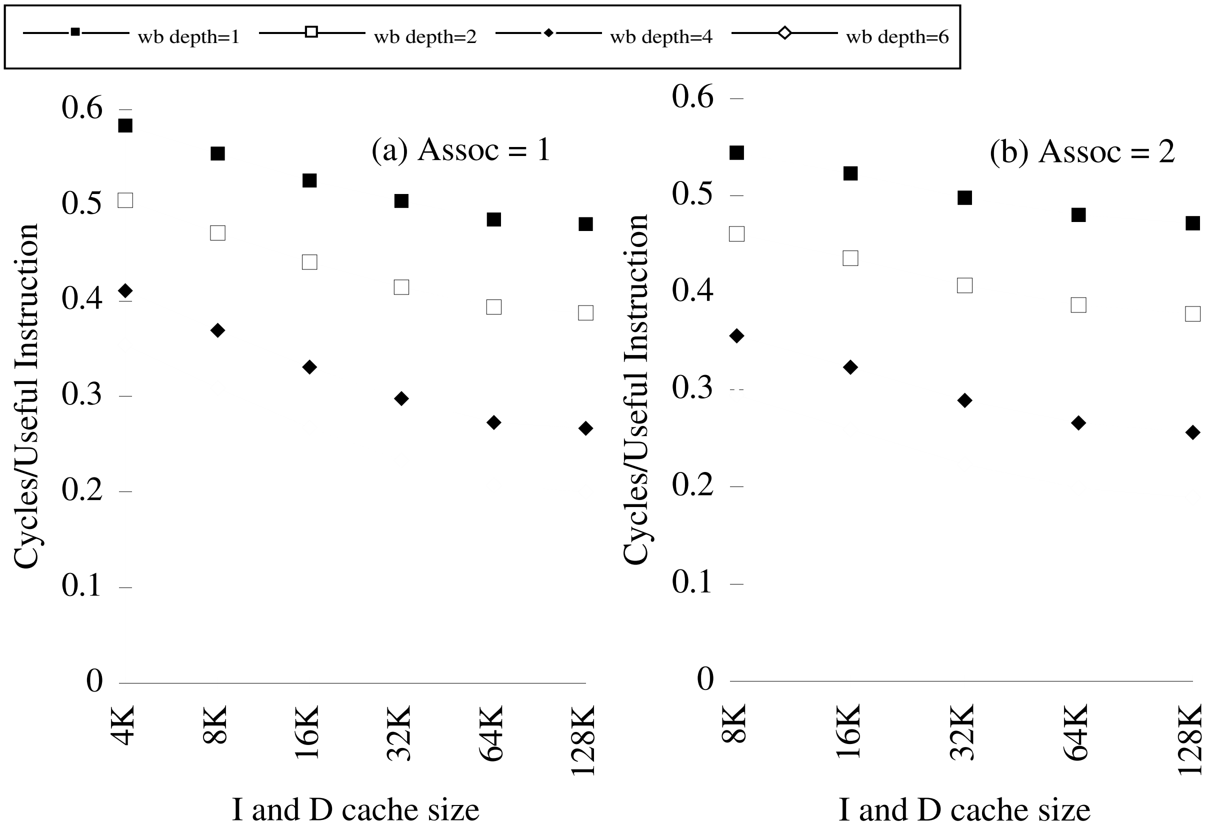

Figure 2.7: Write buffer CPI contribution for VLIW, Without page-mode writes

entry unified TLB. For half the benchmarks, it is below 0.01 even for a 32 entry TLB.

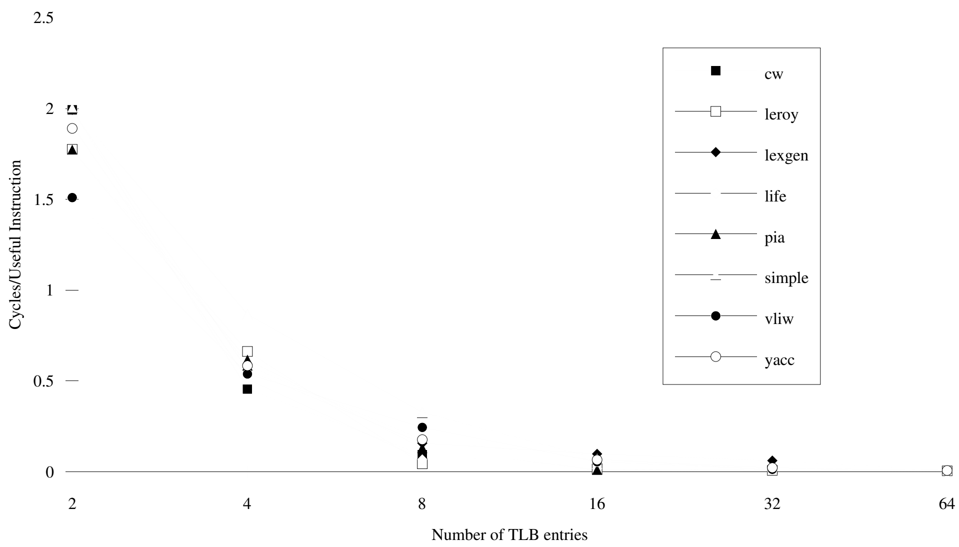

Figure 2.8: TLB contribution to CPI

2.3.6 Validation

To validate my simulations, I ran each of the benchmarks five times on a DECStation 5000/200 (running Mach 2.6) and measured the elapsed *user time* for each run. I ran the programs on a lightly-loaded machine but not in single-user mode. The simulations with *write allocate/subblock placement*, 64K direct-mapped caches, 16 byte blocks, and a 64 entry TLB correspond closely to the DECStation 5000/200 with the following three important differences. First, the simulations ignore the effects of context switches and system calls. Second, the simulations assume a *virtual address=physical address* mapping that can have many fewer conflict misses than the random mapping used in Mach 2.6 [49]. Third, the simulations assume that all instructions take exactly one cycle (plus memory-system overhead).

In order to minimize the memory-system effects of the virtual to physical mapping and context switches, I take the minimum CPI of the five runs for each program and compare it to the CPI obtained via simulations. I present my findings in Table 2.8: *measured (sec)* is the user time of the program in seconds, *measured CPI* is the CPI obtained from the measured time, *simulated CPI* is the CPI obtained from the simulations, *multi-cycle CPI* is the overhead of multi-cycle instructions when it could be accurately computed; *discrepancy* is the discrepancy between the simulated CPI plus the multi-cycle CPI and the measured CPI as a percentage of measured CPI.

Table 2.8 shows that with the exception of `PIA`, the discrepancy is less than 10% and that the actual runs validate the simulations. The discrepancy in `PIA` is due to multi-cycle instructions that comprise 4.8% of the total instructions executed. Because multi-cycle instructions do not cause stalls until their results are used, their cost can usually be

Program	Measured (sec)	Measured CPI	Simulated CPI	Multi-cycle CPI	Discrepancy (%)
CW	25.83	1.42	1.39	0.00	2.48
Knuth-Bendix	14.95	1.27	1.21	0.00	5.22
Lexgen	16.13	1.40	1.31	0.00	6.29
Life	17.16	1.23	1.21	0.00	1.19
PIA	6.41	1.43	1.18	*	17.62
Simple	29.81	1.33	1.21	*	9.03
VLIW	25.61	1.76	1.39	0.20	9.66
YACC	6.58	1.39	1.36	0.00	2.20

cannot be determined without simulating the CPU and/or FP unit pipelines.

Table 2.8: Measured versus Simulated

determined only by simulation. I was able to determine accurately the overhead of multi-cycle instructions for `VLIW` because the results of most multi-cycle instructions are used immediately afterwards. In the case of PIA, the distance between multi-cycle instructions and their use varies considerably. However, even if each multi-cycle instruction stalls the CPU for half its maximum latency, the discrepancy falls well below 10%. Thus, multi-cycle instructions can explain the discrepancy for `PIA`.

2.3.7 Extending the results

Section 2.3.3 demonstrated that heap allocation has a significant memory system cost if new objects cannot be allocated directly into the cache. In this section, I present an analytic model that predicts the memory system cost due to heap allocation when this is the case. The model formalizes the intuition presented in Section 2.3.1, and predicts the memory system cost due to heap allocation when the block size, the miss penalties, or the heap-allocation rate changes. I use the model to speculate about the memory system cost of heap allocation for caches without subblock placement if SML/NJ were to use a simple stack.

An analytic model

Recall that heap allocation with copying garbage collection allocates memory that typically has not been touched in a long time and is unlikely to be in the cache. This is especially true when the allocation area does not fit in the cache. When newly allocated memory is initialized, write misses occur. The rate of write misses depends upon the allocation rate (a words/instruction) and the block size (b words). Given the rate of write misses, I can calculate the memory system cost, C, due to heap allocation. The read and write-miss penalties are r_p and w_p respectively.

Under the assumption that the allocation area does not fit in the cache, *i.e.* initializing writes miss,

$$C_{\text{write alloc}} = w_p * \text{a/b}$$

Under the additional assumption that programs touch data soon after they allocate it,

Cache size (Kilobytes)	write no alloc/no subblock (%)	write alloc/no subblock (%)
8K	7	2
16K	7	2
32K	7	2
64K	11	6
128K	32	24
256K	129	111
512K	1848	1746

Table 2.9: Percent difference between analytical model and simulations

$$C_{\text{write no alloc}} = r_p * \text{a/b}$$

Because the benchmarks do few assignments, the cost of heap allocation should account for the difference in CPIs when the write-miss policy varies. Hence,

$$C_{\text{write alloc/no subblock}} \approx CPI_{\text{write alloc/no subblock}} - CPI_{\text{write alloc/subblock}}$$
$$C_{\text{write no alloc/no subblock}} \approx CPI_{\text{write no alloc/no subblock}} - CPI_{\text{write alloc/subblock}}$$

Table 2.9 shows the average (arithmetic mean) difference between the predicted cost, C and the actual difference in CPIs, as a percentage of the actual difference in CPIs. The values used to calculate Table 2.9 were: $b = 4$, r_p=15 and w_p=15.

The model is accurate for small caches of 128K or less, when the allocation area does not fit in the cache. As expected, the model is inaccurate when the allocation area fits in the cache. The percentage difference increases rapidly as the benefit of subblock placement becomes negligible. Thus, this model predicts the memory system cost of heap allocation only for small cache sizes.

SML/NJ with a stack

I can speculate about the memory-system cost of heap allocation in SML/NJ when a stack is used using this model. In the absence of first-class continuations, which the benchmarks do not use, callee-save continuations can be stack-allocated easily. The callee-save continuations correspond to procedure activation records. The first two columns of Table 2.10 give the rate of heap allocation with and without heap allocation of callee-save continuations.

Assuming only continuations are stack-allocated, column 3 of Table 2.10 presents an estimate of the memory-system cost of heap allocation for caches that do not have subblock placement and are too small to hold the allocation area. The block size is 16 bytes, the read-miss penalty 15 cycles, and the write-miss penalty for the no-subblock caches 15 cycles. Because the read and write miss penalties are the same, C is the same for write-allocate and write-no-allocate organizations.

This is an upper bound on the expected memory-system cost of heap allocation with a stack because it may be possible to stack-allocate additional objects [52]. We see that even

Program	Allocation rate including callee-save conts. (words/useful instruction)	Allocation rate excluding callee-save conts. (words/useful instruction)	C (cycles/instruction)
CW	0.12	0.04	0.15
Knuth-Bendix	0.23	0.12	0.44
Lexgen	0.11	0.03	0.12
Life	0.11	0.02	0.09
PIA	0.17	0.13	0.47
Simple	0.14	0.05	0.17
VLIW	0.16	0.06	0.23
YACC	0.14	0.07	0.24
Median	0.14	0.05	0.20

Table 2.10: Assuming procedure activation records are stack allocated in SML/NJ, this table presents the expected memory system cost of heap allocation for caches without subblock placement

with a simple stack, the memory system costs due to heap allocation for caches without subblock placement will probably be significant for SML/NJ programs.

2.4 Related Work

There have been many studies of the cache behavior of systems using heap allocation and some form of copying garbage collection. Peng and Sohi [68] examine the data-cache behavior of small Lisp programs. They use trace-driven simulation, and propose an ALLOCATE instruction for improving cache behavior that allocates a block in the cache without fetching it from memory. Wilson *et al.* [96, 97] argue that cache performance of programs with generational garbage collection will improve substantially when the youngest generation fits in the cache. Koopman *et al.* [50] study the effect of cache organization on combinator graph reduction, an implementation technique for lazy functional programming languages. They observe the importance of a write-allocate policy with subblock placement for improving heap allocation. Zorn [99] studies the effect of cache behavior on the performance of a Common Lisp system when stop-and-copy and mark-and-sweep garbage collection algorithms are used. He concludes that when programs are run with mark-and-sweep garbage collection they have substantially better cache locality than when run with stop-and-copy garbage collection.

My work differs from previous work in two ways. First, previous work uses the overall miss ratio as the performance metric, which is a misleading indicator of performance. The overall miss ratio neglects the fact that read and write misses may have different costs. Also, the overall miss ratio does not reflect the rates of reads and writes, which may affect performance substantially. I use memory system contribution to CPI as my performance metric, which accurately reflects the effect of the memory system on program running time.

Second, previous work does not model the entire memory system: it concentrates solely on caches. Memory-system features such as write buffers and page-mode writes interact with the costs of hits and misses in the cache and should be simulated to give a correct picture of memory-system behavior. I simulate the entire memory system.

Appel [7] estimates CPI for the SML/NJ system on a single machine using elapsed time and instruction counts. His CPI differs substantially from mine. However, Appel confirms my measurements by personal communication and later work [8]. The reason for the difference is that instructions were undercounted in his measurements.

Jouppi [45] studies the effect of cache write policies on the performance of C and Fortran programs. My class of programs is different from his, but his conclusions support mine: that a write-allocate policy with subblock placement is a desirable architectural feature. He finds that the write-miss ratio for the programs he studies is comparable to the read-miss ratio, and that write-allocate with subblock placement eliminates most of the cost of the write misses.

2.5 Conclusions

In this chapter, I study the memory-system performance of SML programs. To my surprise, I find the memory-system performance of SML programs is quite good on some memory systems. In particular, on an actual machine (the DECStation 5000/200), I find that the memory-system performance of SML programs is comparable to that of C and Fortran programs [19]: programs run only 3 to 13% slower due to data-cache misses than they would run with a zero-latency memory.

I also find that heap allocation has a dominant effect on the memory-system performance of SML programs. For memory-system configurations that do not support heap allocation well, the slowdown due to data-cache misses is often higher than 50%. I find that the memory-system features that are important for achieving good performance with heap allocation are subblock placement with a subblock size of one word, combined with write-allocate on write-miss, page-mode writes, and cache sizes of 32K or larger. Heap allocation leads to poor memory-system performance on machines whose caches are smaller than the allocation area of the programs (256K or larger for the benchmarks that I study) and do not have one or more of the features mentioned previously. That is, most workstations circa 1995 do not support heap allocation well.

These results have two implications for compiling SML programs. First, we should avoid heap allocation whenever possible. Although heap allocation does not have to lead to poor memory-system performance, it often does. Thus, we should prefer stack allocation. Second, we should focus on reducing the instruction counts of SML programs, instead of improving their memory-system performance. On machines with the right memory-system architectures, the memory-system performance of SML programs is comparable to that of C and Fortran programs. Because SML programs are so much slower than C and FORTRAN programs, even on these machines, the implication is that SML programs must be executing too many instructions.

Chapter 3

Cost of automatic storage management

In this chapter, I study the conjecture that SML programs spend a lot of time doing automatic storage management. This is joint work with Amer Diwan. Amer and I designed the experiment and did the measurements. Amer also made additional changes to the SML/NJ compiler and QPT [12, 53, 54] beyond those described in Chapter 2 so that we could make the measurements.

I measure the cost of storage management for eight programs on a DECstation 5000/200 [28]. I chose the DECStation 5000/200 because its memory system is favorable to programs that heap allocate intensively. Chapter 2 shows that a less favorable memory-system organization would increase the cost of storage management by increasing the cost of allocation.

As I mentioned previously, I use trace-driven simulation to make the measurements. Trace-driven simulation allows me to count the instructions spent performing various tasks, such as tagging integers and implementing the write barrier. The measurements include most of the instruction-level and memory-system costs of storage management. I measure instructions spent garbage collecting, allocating, checking if garbage collection is necessary, tagging, implementing a write barrier, and making code relocatable so that it can be placed in the heap and garbage-collected. In addition, I measure the memory-system cost incurred during garbage collection and the memory-system cost incurred during the rest of program execution. I also measure the effect of garbage collection displacing instructions and data used during the rest of program execution from the cache. I estimate upper bounds on the memory system cost due to the disruption of spatial locality by storage management, header words occupying space in the data cache, and instruction-cache misses from storage-management instructions.

The measurements show that SML programs spend 19% to 46% (with a median of 31%) of their execution time doing automatic storage management. They also show that the cost of automatic storage management is scattered throughout the programs: in particular the time spent doing storage-management tasks other than garbage collection is greater than the time spent collecting garbage. I discuss the implications of these results for compiling SML programs in the conclusion of this chapter.

I have organized the chapter in the following manner: Section 3.1 introduces terminology and describes the storage-management strategy used by the SML/NJ compiler, Section 3.2 describes the measurement techniques and benchmark programs, Section 3.3 presents measurements for eight SML/NJ programs, Section 3.4 reviews related work, and Section 3.5 concludes.

3.1 Background

The following sections introduce terminology and describe the storage-management technique that is used in the version of the SML/NJ compiler that I measure.

3.1.1 Terminology

Storage management refers to the management of memory by an individual program. In a garbage-collected program, the part of the program that is not the garbage collector is called the *mutator*. Execution of the mutator is called *mutation*. Storage management has two components in garbage-collected programs. The first component, which is obvious, is the execution of the garbage collector. The second component comprises tasks done outside the garbage collector to support storage management. The cost of these tasks is called *the storage-management cost during mutation*.

The number of instructions that it takes to perform a task is the *instruction-level cost* of that task. The time that a processor waits for memory while performing a task is the *memory-system cost* of that task.

3.1.2 Storage management in the SML/NJ compiler

I use version 0.91 of the SML/NJ compiler in this study. Storage management in this system has many components. One obvious component is garbage collection, but there are many additional components:

- checking whether garbage collection is needed
- allocating new objects
- tagging
- implementing a write barrier,
- and implementing position independent code.

The SML/NJ compiler uses heap-only allocation: *all* allocation is done on the heap. In particular, all activation records are allocated on the heap instead of a call stack. The heap is managed automatically using generational copying garbage collection [5, 6, 58].

In copying garbage collection [20, 32], an area of memory is reclaimed by copying the live (non-garbage) data to another area of memory. The area from which the data is copied can then be reused.

Version 0.91 of the SML/NJ compiler uses a simple variant of generational copying garbage collection [5]. It divides memory into an old generation and an allocation area. The mutator creates new objects in the allocation area. When the allocation area becomes full, the garbage collector copies the live data in the allocation area to the old generation in a *minor collection*. When the size of the old generation becomes sufficiently large, the garbage collector collects the entire heap in a *major collection*. The garbage collector copies live objects using a Cheney scan [20], which copies objects in a breadth-first order. In Section 3.2.4, I describe the criteria that the system uses to decide when to collect the whole heap. Generational garbage collection is efficient because most allocated objects die young (about 99% [7, p. 206]) and few objects are copied from the allocation area.

Before a mutator can allocate an object, it must check that there is sufficient space on the heap to allocate the object. If not, it must invoke the garbage collector. The SML/NJ compiler places these checks at the beginning of most extended basic blocks, instead of inserting them before every allocation. An extended basic block is a block of code with only forward jumps. Thus, the cost of a check may be amortized across across several allocations. The SML/NJ compiler places checks on only some of the extended basic blocks. For other extended basic blocks, the checks are redundant because there are checks along all paths to those blocks that suffice to establish that a garbage collection will not be needed. The SML/NJ compiler also places checks on many extended basic blocks that do no allocation because these checks are also used to implement asynchronous signals [74].

The SML/NJ compiler generates in-line machine code to allocate everything but arrays and strings. The in-line machine code is only two instructions long. Recall from Section 2.1.4 that only two instructions are needed because the garbage collector always reclaims the *entire* allocation area. The SML/NJ compiler generates procedure calls to allocate arrays and strings. I do not regard initializing newly allocated storage as part of allocation.

The compiler tags all objects so that the garbage collector can find all live objects and copy them. All objects except integers have a header word that describes the kind and the size of the object. The kind tells whether the object is a record, array, byte-array, etc. The compiler tags integers with a 1 in the least significant bit and it tags pointers with a 0 in the least significant bit. This means that programs often need to manipulate tags to do integer arithmetic operations.

The *write barrier* tracks all pointers from the old generation to objects in the allocation area. The objects that the write barrier tracks must be regarded as live when only the allocation area is collected. Otherwise, collection of the allocation area could create dangling pointers.

The compiler implements the write barrier using a *store list*. The only way that a program can create a pointer from the old generation to the new generation is by doing an assignment. Before a program does an assignment `x := t` where the source value `t` could be a pointer, it adds `x` to the store list. The garbage collector processes the store list during a minor collection and then discards it.

The SML/NJ interactive system places code in the heap so that it may be reclaimed by the garbage collector, but this means that code may be moved by the garbage collector. Thus, code needs to be position independent. The compiler implements position independence by doing all addressing of instructions using base-offset addressing. A program adjusts the base register every time it enters a module.

3.2 Methodology

I use trace-driven simulation to measure the cost of storage management. This allows me to measure the cost of storage management precisely, including the memory-system cost, and to separate the cost into its components.

In the following subsections, I describe what I measure for each component of the cost of storage management, the traces and trace-generation mechanism that I use, the memory system that I simulate, my benchmark programs, and the garbage collection sizing parameters that I use.

3.2.1 Measurement methodology for each component

Table 3.1 lists what I measure for each component of the cost of storage management. The first three entries are the cost of garbage collection. The remaining rows are the storage management costs in the mutator.

The one instruction-level cost of storage management that I do not measure is the effect of storage management on program optimization [18]. Diwan *et al.* [30] present techniques that allow extensive optimization even using copying collection with unambiguous roots.

Storage management also affects the memory-system cost that a mutator incurs. I am unable to measure this effect directly. I defer discussing this effect in detail and how I measure it to Section 3.3.3.

I measure the cost of position-independent code as the number of instructions spent updating the base register and the additional instructions that have to be executed relative to position-dependent code. In particular, jump tables are more expensive in position-independent code. For position-dependent code, the table address is an absolute address, but for position-independent code the table address must be computed. In addition, for position-dependent code, the table gives absolute addresses. In position-independent code, the table gives relative offsets and the address of the target must be computed also.

I use the modified versions of QPT and the SML/NJ compiler described in Chapter 2 to produce traces for SML/NJ programs. Amer Diwan extended QPT further by adding an event tracing facility to QPT.

An *event* is a specially marked instruction. Amer Diwan extended QPT so that when an event is encountered during the execution of a traced program, an event marker is inserted into the trace. The event marker identifies the event and also contains parameters to the event (if any). The simulator consuming the trace can take whatever actions are necessary when it encounters an event marker. For example, when an integer-tag event is encountered during run time, an integer-tag event marker is inserted into the trace. When the simulator sees the marker, it increments the count of instructions spent doing integer tagging.

I also use the event-tracing mechanism to mark different phases of garbage collection. For example, the first and last instructions for the Cheney scan are events. When the simulator encounters one of these events, it outputs a message noting the event and the memory-system

Root processing	Instructions to process the store list and registers, including copying objects immediately reachable from the store list and registers. Also any memory-system cost incurred while executing those instructions.
Cheney Scan	Instructions to do the breadth-first copying of objects. Also any memory-system cost incurred while executing those instructions.
Moveback	Instructions to move the old generation to one end of the free region. Also any memory-system cost incurred while executing those instructions.
Allocation	Instructions to increment the allocation pointer and initialize registers to point to newly-allocated objects. This does not include instructions to initialize newly-allocated storage before it is used.
GC Check	Instructions to check whether garbage collection is needed and to jump to the garbage collector entry code if garbage collection is needed.
Integer Tagging	Instructions to tag and untag integers.
Record Tagging	Instructions to write header words of records.
Position-Independent code	Instructions to update base register. Also, additional instructions needed relative to position dependent code to compute jump table addresses from the addressing register.
Store list	Instructions to add a record to the store list.

Table 3.1: Measurements for each component of the cost of storage management

statistics. The statistics are the total numbers of reads, writes, data-cache read misses, data-cache write misses, instruction-cache misses, and write buffer stalls through that point of program execution. From these statistics, I can compute the memory-system cost incurred during each phase of garbage collection.

Amer Diwan identified events to QPT by adding *event tables* to executable files. Each event table corresponds to one kind of event and lists the locations of all instructions at which that kind of event occurs. In addition, each event table also specifies the values of the parameters to the event for each instruction. When invoked on an executable file, QPT searches the executable file for these tables. Amer Diwan modified the SML/NJ compiler to emit these tables for "interesting" events. He created event tables for the garbage collector, which is written in C, by manually editing the assembly file produced by a compiler.

The tracing mechanism is non-intrusive: the traces produced by QPT correspond to addresses in the original programs rather than those in the instrumented programs.

3.2.2 Memory system simulation

I simulate the DECstation 5000/200 memory system using the extended version of Tycho [42] described in Section 2.2.1. Table 3.2 summarizes the memory system of the DECstation 5000/200. The DECstation 5000/200 has a split instruction and data cache. The instruction cache is direct mapped and is composed of blocks of 16 bytes each. The data cache is also direct mapped, but has a block size of 4 bytes. However, on a read miss, 16 bytes aligned on a 16 byte boundary are fetched from memory. Because the DECstation 5000/200 has a block size of 4 bytes, a write miss can write to the cache immediately without fetching a block from memory[1]. The DECstation also has a write buffer to avoid stalling the CPU on a write; the CPU needs to stall on a write only when the write-buffer fills up. Chapter 2 shows that this memory system is favorable to allocation-intensive programs.

3.2.3 Benchmark Programs

I use the same benchmarks that I use to study memory-system performance. I describe the benchmarks in Section 2.2.3.

Stefanović and Moss [87] find that the allocation of callee-save continuation closures on the heap has a profound impact on the young-object dynamics of ML programs. In the programs they measure[2], most objects are short-lived. They attribute the high mortality rate to the allocation of callee-save continuation closures on the heap. The age at which most objects die is program dependent, as is the percentage of objects that die at that age. In particular, YACC has a relatively high object survival rate when compared to Knuth-Bendix. Thus, I expect YACC to have a higher garbage collection cost than Knuth-Bendix because the garbage collector has to copy more objects from the allocation area to the old generation.

[1] Partial-word writes are treated differently, but because the benchmark programs do so few partial writes, I ignore them in my discussion without a loss of accuracy.

[2] This includes many of the programs we measured. However, they give data for only Knuth-Bendix and YACC in their paper.

Instruction cache
64K, direct mapped
Block size 16 bytes
On miss fetch aligned 16 bytes
Data cache
64K, direct mapped
Block size 4 bytes
Write through
On write miss, write word to cache
On read miss, fetch aligned 16 bytes
Write buffer
Six 4-byte entries

Table 3.2: Summary of the DECstation 5000/200 memory system

Task	Penalty (in cycles)
Write hit or miss	0
Read miss	15
Instruction-fetch miss	15
Non-page-mode write	5
Page-mode write	1

Table 3.3: Penalties of memory operations

Program	Roots	Cheney	Move back	Alloc.	Gc check	Int tag	Record tag	Pos. ind. code	Store list	Total
CW	0.06	0.05	0.01	0.04	0.05	0.02	0.04	0.03	0.01	0.31
Knuth-Bendix	0.00	0.09	0.01	0.08	0.05	0.00	0.08	0.02	0.00	0.33
Lexgen	0.00	0.06	0.01	0.03	0.06	0.08	0.03	0.04	0.00	0.31
Life	0.00	0.01	0.00	0.04	0.06	0.02	0.04	0.02	0.00	0.19
PIA	0.00	0.05	0.00	0.06	0.05	0.00	0.06	0.01	0.00	0.23
Simple	0.05	0.07	0.03	0.05	0.04	0.03	0.05	0.02	0.00	0.34
VLIW	0.01	0.06	0.01	0.04	0.04	0.05	0.04	0.03	0.00	0.28
YACC	0.01	0.25	0.04	0.04	0.04	0.02	0.04	0.02	0.00	0.46

Table 3.4: Breakdowns of storage management costs for benchmark programs. All numbers are fractions of total execution time

3.2.4 Garbage collection sizing parameters

I use the default strategy for sizing the allocation area and the old generation [5]. The heap is sized as `r` times the size of the old generation after the old generation is collected, where `r` is the desired *ratio* of heap size to live data. I use the default system value (`r=5`). The allocation area is sized as one-half of the free space (the heap space not occupied by the old generation). As the old generation grows after each collection of the allocation area, the free space decreases and the allocation area decreases. The old generation is collected when the remaining free space is less than the original size of the old generation (less than 1/5 the size of the heap).

In addition to the ratio, the garbage collector is controlled by the *softmax* and the *initial heap size*. The softmax is a desired upper limit on the heap size. It is exceeded only to prevent programs from running out of space. The softmax is 20 megabytes; the benchmark programs never reach this limit and are always able to resize their heaps to maintain the desired ratio of 5. The initial heap size is 1 megabyte.

3.3 Results

In this section, I present my results. First, I give the breakdown of the cost of storage management. Second, I show that the cost of automatic storage management is scattered throughout programs: focusing only on the cost of garbage collection is misleading. Third, I identify the different components of the memory-system cost of storage management and explain why it is difficult to measure these components. I then give measurements for most of these components and estimate upper bounds for the remaining components. These estimated costs range from a few percent to negligible on the DECStation 5000/200.

3.3.1 The cost of storage management

Figure 3.1 gives a breakdown of the cost of storage management. Table 3.4 gives these numbers as a table. For garbage collection, I measure:

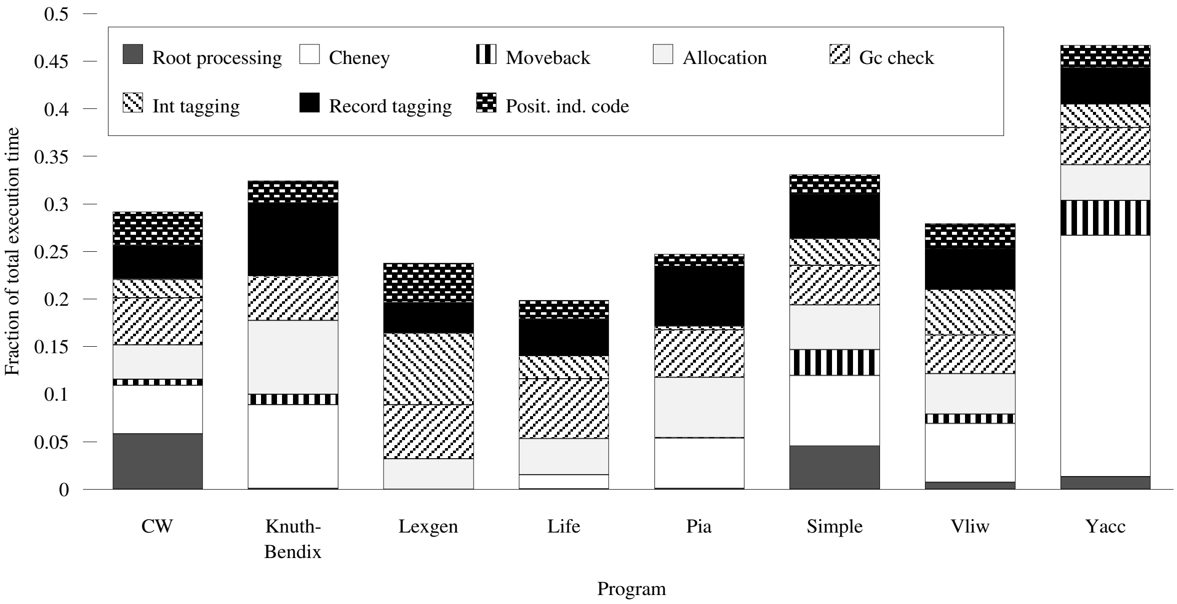

Figure 3.1: Breakdowns of storage management cost for benchmark programs.

- the cost of processing the store list and registers (roots),
- scanning and forwarding reachable objects (Cheney),
- and moving the heap back at the end of a major collection (moveback).

I also measure entry and exit costs, which are the costs of entering the collector from SML and returning from the collector to SML. These costs are insignificant, so I omit them from the graph. All the costs of garbage collection include memory-system costs.

For mutation, I measure:

- the instruction counts for allocation (allocation),
- checking whether garbage collection is needed (gc check),
- tagging integers (int tagging),
- storing header words (record tagging),
- and position-independent code (posit. ind. code).

The cost of adding elements to the store list is negligible for most programs, so I omit it from the graph.

Figure 3.1 suggests several opportunities for improving the performance of storage management in SML/NJ programs. First, eliminating position-independent code can reduce program execution time by 1% to 4%. Although placing code in the garbage-collected heap is useful in the setting of an interactive development environment, it is not useful for stand-alone executables.

Second, improving integer tagging, which accounts for 0% to 8% of overall execution time, ignoring the negligible instruction cache penalty, may reduce execution time. Integers are tagged with 1 and pointers are tagged with 0 in the least significant bit. By tagging integers with 0 and pointers with 1 and using displacement addressing, which is available on many architectures, many tagging operations can be removed. Representation analysis [56] may also eliminate some tagging operations.

Third, I speculate that using a hash-based scheme would improve the write barrier implementation. The cost of adding elements to the store list during mutation is 0% to 1% of overall execution time and the cost of processing the store list during garbage collection is 0% to 6% of overall execution time (Table 3.4, columns *Store list* and *Roots*). In a hash-based scheme, a table is used instead of a list, and duplicate entries are eliminated when elements are added to the table during mutation. The cost of adding entries during mutation should be similar for both schemes, because the cost of adding an element to the store list is already high (8 or 9 instructions). The cost of processing the table during garbage collection would be lower than the cost of processing the list, because the table has no duplicate entries.

Fourth, scanning and forwarding reachable objects takes 1% to 25% of overall execution time. This can be reduced by coding the inner loops of the collector carefully and by increasing the size of the allocation area. Increasing the size of the allocation area allows more objects to die, and thus fewer objects need to be copied during garbage collection. However, this involves a trade-off: increasing the size of the allocation area may reduce copying time but increase the memory-system cost. For the DECStation 5000/200 memory-system organization, which is favorable to heap allocation, increasing the allocation area size is unlikely to change the memory-system cost. In Chapter 2, I show that halving the cache size for the DECStation 5000/200 organization affects performance little. Some other memory-system organizations, such as the SPARCStationII [24], are more sensitive to cache size. Increasing the allocation area size can increase the memory-system cost greatly.

An interesting point to note about Figure 3.1 is that the cost of checking whether garbage collection is needed (gc check) is larger than the cost of allocation (allocation) for CW, Lexgen, and Life. This is despite the fact that the check and allocating an object both take two instructions on the MIPS, and that a check is sometimes for multiple allocations. I speculate that this is because the SML/NJ compiler overloads checks to implement asynchronous signals [74]. This results in checks in extended basic blocks that do no allocation, so that the allocation cost is not an upper bound on the cost of checking whether garbage collection is needed.

3.3.2 Most costs are incurred during mutation

To illustrate how inaccurate it can be to focus only on time spent garbage collecting, Figure 3.2 compares the cost of garbage collection against the cost of instructions executed during mutation to support storage management. The garbage-collection cost includes the memory-system cost, but the storage-management cost during mutation does not include the memory-system cost. Still, the storage-management cost during mutation is larger than the garbage-collection cost for seven of the programs. This shows that regarding the cost of garbage collection as the cost of storage management is inaccurate: most of the cost is scattered throughout the program.

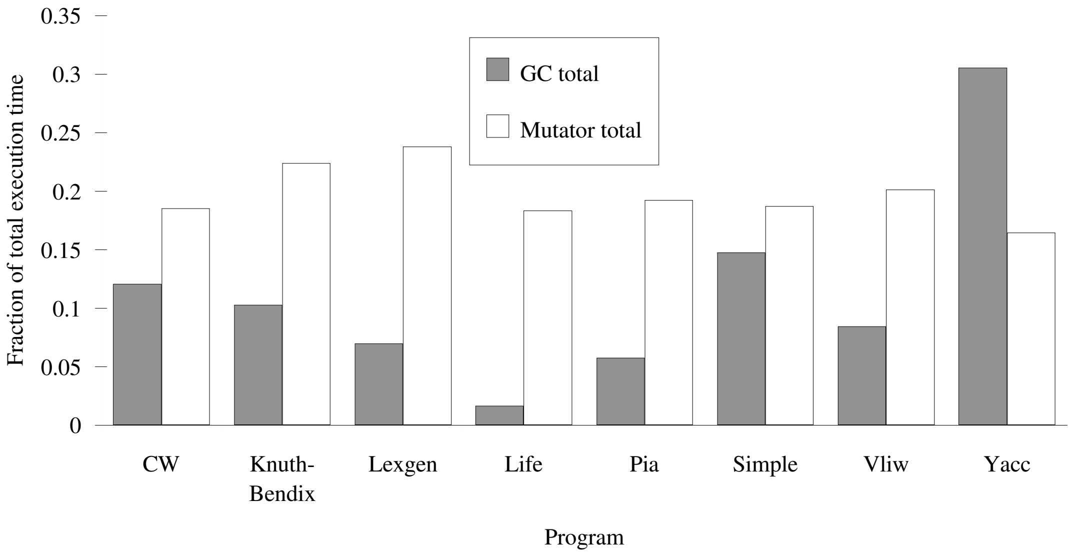

Figure 3.2: Comparison of garbage collection and mutation storage-management costs

3.3.3 The memory-system cost of storage management

Because the cache is a global shared resource, it can be difficult to pinpoint the exact cause of a cache miss. One instruction may cause a cache miss that knocks data out of the cache that is used by a subsequent instruction. Thus, the subsequent instruction will also have a cache miss. This makes it difficult to correlate misses with their actual source: an instruction at which a miss occurs may not be the actual cause of the miss.

Storage management may incur memory-system cost:

1. during garbage collection.

2. by collection displacing mutator data and instructions from the cache.

3. by allocating memory that is not resident in the cache. When the mutator initializes or uses the memory, cache misses may occur.

4. by increasing the size of data by adding header words. This reduces spatial locality and the effective size of the cache.

5. by reducing the spatial locality of mutation by rearranging the layout of data in memory. This may also improve spatial locality.

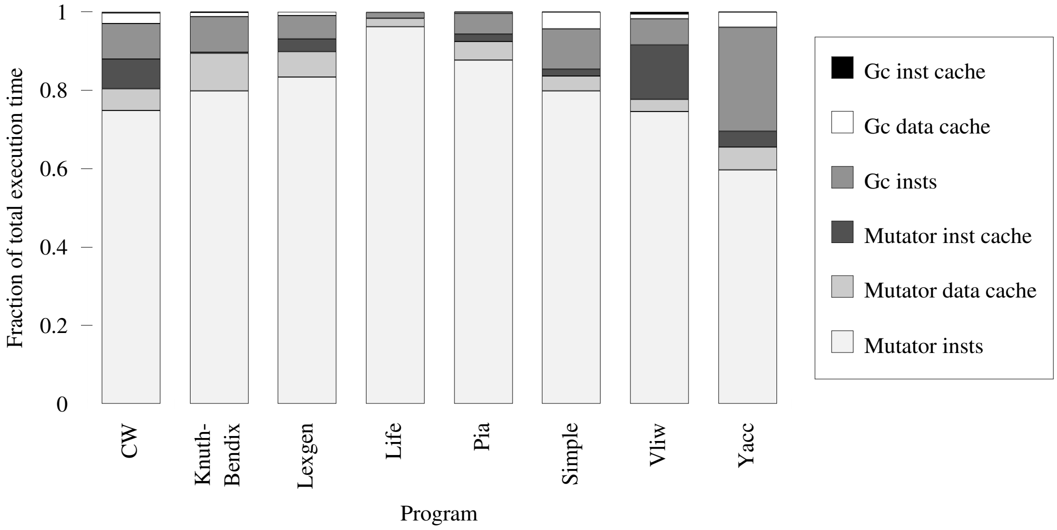

Figure 3.3: Breakdown of memory-system cost during collection and mutation.

6. by changing the code size of the mutator by requiring code for tagging, the write barrier, and position-independent code. This may increase instruction-cache misses.

I measure some of the components of the memory-system cost of storage management, and place upper bounds on the remaining components

I measure the memory-system cost incurred during garbage collection. Figure 3.3 presents the memory-system costs incurred during mutation and collection. The memory-system cost that occurred during collection is 0.1% to 4.3% of overall execution time. The cost of instruction-cache misses during collection is negligible.

I also measure the effect of garbage collection displacing instructions and data used during mutation from the cache. The effect is negligible. The cost of instruction misses during mu-

Program	Before (miss rate)	After (miss rate)
CW	0.03	0.02
Knuth-Bendix	0.04	0.04
Lexgen	0.03	0.03
Life	0.01	0.01
PIA	0.01	0.01
Simple	0.01	0.01
VLIW	0.02	0.01
YACC	0.03	0.04

Table 3.5: Data-cache miss rates before and after garbage collections

tation caused by garbage collection is bounded from above by the cost of instruction misses during garbage collection, because the machine has a split instruction and data cache. Because the cost of instruction misses during garbage collection is low (see Figure 3.3), this effect is negligible. To measure the effect of data-cache misses during mutation caused by collection, I collected memory-system statistics for intervals before and after each collection. The arithmetic mean of the sizes of the intervals measured before each collection was 143,000 instructions. The mean of the sizes of the intervals after each collection was 166,000 instructions. Table 3.5 shows the average cache-miss rates for the intervals for each program; the `Before` column gives the data-cache read-miss rates for the intervals just before each garbage collection and the `After` column gives the data-cache read-miss rates for the intervals after each garbage collection. If garbage collection were disturbing the cache locality of mutation, I would expect the miss rates after garbage collection to be noticeably higher than those before garbage collection. I see only a slight variation. This suggests that data references during collection do not cause significant data-cache misses during mutation.

Because the DECStation 5000/200 memory system has no penalty for write misses, besides write-buffer penalties that are small enough to be negligible, there is no cache penalty for initializing writes that miss. If there were a penalty for write misses[3], the programs would run 24% to 72% slower than they do now. In other words, with a penalty for cache write misses, most of the cost of storage management would be for initializing newly-allocated objects.

Although I am unable to measure the remaining components of the memory-system cost of storage management exactly, I place upper bounds on these components. Recall that header words and copying during garbage collection may worsen the spatial locality of the program. To bound this, note that obtaining perfect spatial locality could change the cache misses by at most a factor of 2. This is because without header words, at most two of the smallest objects (2 word cons cells) can fit in 16 bytes (which is the amount fetched on a read miss). Table 3.6 shows an upper bound on the disruption of spatial locality by garbage collection and other storage-management tasks. I compute the upper bound by halving the cost of mutator data cache misses. This table demonstrates that the cost of disrupted spatial locality is small. Of course, the cost of disrupted spatial locality may be more substantial with larger fetch sizes or larger cache-miss penalties.

Header words may also increase the memory-system cost of mutation by decreasing the effective cache size. Because a header word occupies 1/3 of the space used for the typical smallest object, a list cell, at most 1/3 of the space in the cache is being occupied by header words. The cache would need to be 50% larger in practice to achieve the same effective cache size. I can give a generous estimate of the cost of header words by subtracting the data-cache cost for a 128K cache from the cost for 64K cache, that is, generously assuming that without header words the cache size is effectively doubled. Table 3.7 gives the improvement in going to a 128K cache; this is an upper bound on the memory-system costs due to header words decreasing the effective cache size.

The effect of effectively increasing the size of data depends on cache boundary conditions. If data fits well within the cache, then increasing the size does not matter much. However, if data just fits in the cache, then increasing the size of the data may cause cache misses. If data

[3]In particular, if write misses were blocking and required the cache block to be fetched from main memory.

Program	Upper Bound (% execution time)
CW	2.8
Knuth-Bendix	4.8
Lexgen	3.2
Life	1.1
PIA	2.4
Simple	1.9
VLIW	1.6
YACC	2.9

Table 3.6: Upper bound on disruption of spatial locality by storage management

Program	Upper Bound (% execution time)
CW	1.8
Knuth-Bendix	3.7
Lexgen	2.6
Life	1.0
PIA	1.8
Simple	0.8
VLIW	0.7
YACC	1.0

Table 3.7: Upper bound on data cache costs due to smaller effective cache size

Program	Est. cost (% execution time)
CW	1.8
Knuth-Bendix	0.1
Lexgen	0.9
Life	0.0
PIA	0.4
Simple	0.4
VLIW	3.8
YACC	1.1

Table 3.8: Estimate of instruction cache costs due to storage management instructions

does not fit in the cache, increasing the size of the data leads to proportionately more cache misses. I explored the effect of cache size on SML/NJ programs thoroughly in Chapter 2, and found that there were no dramatic boundary conditions for SML/NJ programs on the DECStation 5000/200.

Just as header words increase the size of data, storage-management instructions increase the size of a program. This may cause additional instruction-cache misses. The fraction of instruction misses during mutation that are on average due to storage management is the fraction of instructions executed during mutation in support of storage management. Figure 3.8 gives an estimate of the cost due to these extra instruction-cache misses for each of the programs.

3.4 Related work

This study is more comprehensive in its measurements than previous works studying the cost of storage management in garbage-collected systems. Ungar [92] measures the time spent garbage collecting and the cost of integer tagging in a Smalltalk system, but does not measure other costs incurred during mutation. Zorn [100] compares the cost of two simulated garbage-collection algorithms. In contrast, I measure an actual implementation. He measures the memory-system cost using the cache-miss ratio, which is an inaccurate indicator of performance because it does not separate the cost of read and write misses. Wilson *et al.*[96] and Peng and Sohi [68] also measure the memory-system cost of garbage collection using the cache-miss ratio. They do not measure the instruction-level cost of garbage collection or costs incurred during mutation. Reinhold [73] measures the cost of garbage collection for a Scheme system, including the change in memory-system performance of entire programs, but does not measure costs incurred during mutation.

Steenkiste [86] studies ways to reduce the cost of tagging in Lisp. He also studies instructions used for stack allocation. He is primarily concerned with hardware support to improve tag checking required for dynamic typing. He finds that tag insertion and removal costs about 4.5% with the best software scheme.

There have been several studies of the cost of storage management in languages with explicitly-managed heap storage and stack allocation of procedure activation records. Detlefs [27] measures time spent in allocation and deallocation routines, but does not measure the cost of managing the stack. Grunwald *et al.* [35] finds that the implementation of explicit heap management can affect the performance of allocation-intensive C programs significantly.

3.5 Conclusion

In this chapter, I study the cost of automatic storage management for SML programs. Unlike other work measuring the cost of storage management, I measure both the time spent garbage collecting and costs incurred during mutation.

The measurements show that SML programs spend 19% to 46% (with a median of 31%) of their execution time doing automatic storage management, and that the cost of automatic storage management is scattered throughout programs.

In the previous chapter, I showed that the memory-system performance of SML programs implied that that SML programs must be executing too many instructions. The measurements in this chapter show that automatic storage management is *not* the main reason SML programs are executing too many instructions: even halving the cost of automatic storage management would reduce the execution time of SML programs by only 15%. Thus, I must look elsewhere to determine why SML programs are so much slower than C and FORTRAN programs.

Part II

Optimization

In the first part of this thesis, I demonstrated that SML programs were executing too many instructions. Furthermore, I showed that automatic storage management was *not* the main reason so many extra instructions were being executed.

In this part of the thesis, I describe a new approach to compiling SML programs based on two ideas:

- A "pay-as-you-go" compilation strategy.
- Applying optimizations known to improve loops in conventional languages to recursive SML functions.

Specifically, I describe the TIL compiler and I demonstrate how to apply to apply several optimizations known to improve loops to SML programs. I focus on two sets of optimizations: code motion optimizations, such as common-subexpression elimination and invariant removal and array-bounds checking optimizations.

I have organized this part of the thesis to reflect the structure of the TIL compiler. In Chapter 4, I describe the *typed intermediate language framework* used by TIL and LMLI, the specific intermediate langauge used by TIL. I also give a general overview of the structure of TIL and an example that illustrates how TIL translates SML programs to machine code. In Chapter 5, I describe TIL's machine-independent optimizer, excluding the "loop" optimizations. In Chapter 6, I describe the loop optimizations. Finally, in Chapter 7, I describe how TIL translates optimized programs to machine code.

Chapter 4

The Typed Intermediate Language (TIL) Framework

In this chapter, I provide background material on the TIL (Typed Intermediate Language) compiler. This compiler is primarily joint work with Greg Morrisett. Perry Cheng, Chris Stone, Robert Harper, and Peter Lee also helped construct TIL.

TIL is organized around the framework of *typed* intermediate languages: TIL uses a typed intermediate language in most phases of compilation, and propagates type-related information through all levels of the compiler: type-related information is propagated even through register allocation.

By keeping type information around, TIL is able to use efficient *specialized* data representations, even though Standard ML is a polymorphic language. Previous ML compilers have used a *universal data representation* for compiling at least some parts of an ML program. A universal data representation forces every different type of data to have the same-sized representation, usually a machine word, where part of the machine word is dedicated to a tag. The tag tells a garbage collector whether the data is a pointer that the garbage collector must trace, or an integer or floating point number that garbage collector does not need to trace. The rest of the machine word is dedicated to holding actual data. For example, on a machine with a 32 bit word size, 1 bit might be used for the tag, and 31 bits would be used for data. This data might be a pointer or an integer.

With a universal data representation, data items that are naturally smaller than the untagged part of the machine word, such as characters, are padded to fit in the untagged part of the machine word. Data items that are naturally larger than the untagged part of the machine word, such as floating-point numbers, are represented as pointers.

Forcing each piece of data to have a universal representation solves two important problems in compiling polymorphic languages. First, how do we create one piece of code for a polymorphic function ? If everything has the same size, then any data can be passed to a polymorphic function. Second, how does a garbage collector find data that is in use, so that it avoids reclaiming that data ? The tag provides the garbage collector with the information that it needs.

TIL takes a fundamentally different approach to these problems that is well-suited to a language such as SML. Instead of forcing data to all have the same representation, TIL propagates type information through all phases of compilation and passes type information around *at run time* if necessary. This solves the problem of how to pass data of different sizes to polymorphic functions: the functions simply examine the type of their argument and do different things for different types. It also solves the problem of how to allow the garbage collector to find data that is live: the garbage collector can use type information to determine how to traverse data structures.

I have organized this chapter in the following manner. First, I introduce LMLI [63], the primary intermediate language of TIL. Next, I give an overview of the phases of TIL and the garbage collection technique used by TIL. Finally, I give an example that illustrates how TIL translates SML programs to machine code.

4.1 LMLI

The key difficulty with using a typed intermediate language is formulating a type system that is expressive enough to statically type check expressions that branch on types at run time. TIL uses an *intensionally polymorphic* language called LMLI, which was designed by Morrisett and Harper [63]. LMLI is the actual intermediate language used by the TIL compiler, and it is the language that I use in the next two chapters on optimization. Morrisett and Harper based their design on their intermediate language λ_i^{ML} [37], which they used to explore the theory of intensional polymorphism. In this section, I describe the syntax of λ_i^{ML} and informally describe its semantics. I then present the syntax of LMLI and informally describe LMLI's semantics. Finally, I give an example of an LMLI program.

4.1.1 Overview of λ_i^{ML}

My description of λ_i^{ML} follows Morrisett's presentation [63]. The syntax for λ_i^{ML} is presented in Figure 4.1. There are four syntactic classes: kinds, constructors, types, and terms. Kinds describe constructors, while types describe terms. Constructors can be thought of as the representation of types *at run time*. There is an explicit injection of constructors into types: $T(\mu)$ is the type represented by the constructor μ.

Constructors consist of type variables, the base type Int, function types, constructor functions, constructor applications, and Typerec expressions. Typerec allows constructors to be defined by structural induction on monotypes, which are constructors of kind Ω. Monotypes include type variables, Int, and constructors formed from Arrow. If Typerec μ of $(\mu_{\text{int}}; \mu_{\text{arrow}})$ is given some $\mu = \mathsf{Int}$, it selects μ_{int}. If the Typerec is given some μ of the form $\mathsf{Arrow}(\mu_1, \mu_2)$, it applies itself to μ_1 and μ_2 respectively (i.e. the Typerec "unrolls" itself). It then applies μ_{arrow} to the resulting types.

Typerec is useful for specifying the types of functions that operate inductively over the structure of types. Harper and Morrisett [37] present several examples of such functions, including a function for flattening nested tuples and functions for marshalling and unmarshalling data that is transmitted over a network.

Terms consist of variables, integer literals, functions, polymorphic functions, function applications, and polymorphic function application. `typerec` allows the definition of terms by structural induction on monotypes, and works in a manner similar to Typerec. Informally,

(kinds)	κ	$::=$	$\Omega \mid \kappa_1 \to \kappa_2$
(constructors)	μ	$::=$	$t \mid \mathsf{Int} \mid \mathsf{Arrow}(\mu_1, \mu_2) \mid \lambda t{::}\kappa.\, \mu \mid \mu_1\, \mu_2 \mid$ $\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_{\mathrm{int}}; \mu_{\mathrm{arrow}})$
(types)	σ	$::=$	$T(\mu) \mid \mathsf{int} \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma$
(expressions)	e	$::=$	$x \mid i \mid \lambda x{:}\sigma.\, e \mid \Lambda t{::}\kappa.e \mid e_1\, e_2 \mid e[\mu] \mid$ $\mathtt{typerec}\ \mu\ \mathtt{of}\ [t.\sigma](e_{\mathrm{int}}; e_{\mathrm{arrow}})$

Figure 4.1: Abstract syntax of λ_i^{ML}

it can be thought of as being similar to a "fold" operation over lists, except that it works over monotypes. $\mathtt{typerec}\ \mu\ \mathtt{of}\ [t.\sigma](e_{\mathrm{int}}; e_{\mathrm{arrow}})$ evaluates μ to a normal form. If the normal form of μ is Int, then the `typerec` evaluates e_{int}. If the normal form of μ is $\mathsf{Arrow}(\mu_1, \mu_2)$, the `typerec` applies itself to μ_1 and μ_2 to produce values v_1 and v_2, respectively. It then applies the expression e_{arrow} to μ_1, μ_2, v_1 and v_2.

Morrisett and Harper [37] show that typechecking for λ_i^{ML} is decidable, despite its rich type system.

4.1.2 Overview of LMLI

In this section, I describe LMLI, the intermediate language of the TIL compiler.

Kinds, constructors, and types

In Figure 4.2, I describe the syntax of kinds, constructors, and types of LMLI. Kinds are extended to describe lists of constructors and tuples of constructors. These lists and tuples are useful in building compound constructors such as record constructors and sum constructors.

Every constructor is annotated explicitly with its kind. Raw constructors consist of type variables, arity-0 constructors, arity-1 constructors, recursive constructors, Typecase, and introductory and eliminatory forms for lists and tuples of constructors.

The introductory and eliminatory forms for lists of constructors are straightforward:

- Nil is the empty list of constructors.
- $\mathsf{Cons}(\mu_1, \mu_2)$ places μ_1 on the front of the list μ_2.
- Fold takes a list of constructors. If the list is Nil, it evaluates μ_2. Otherwise, it applies itself to the rest of the list to produce some constructor c. It applies cf to the pair of the first element of the list and c.

(kinds)	κ	$::=$	$\Omega \mid \kappa_1 \rightarrow \kappa_2 \mid \mathsf{List}\ \kappa \mid \mathsf{Tuple}(\kappa^*)$
(constructors)	μ	$::=$	$rc::\kappa$
(raw con.)	rc	$::=$	$t \mid c_0 \mid c_1(\mu) \mid \mathsf{Rectype}(t = \mu)^*\ \mathsf{in}\ \mu \mid cf \mid \mu_1\mu_2 \mid$ $\mathsf{Typecase}\ \mu_1\ \mathsf{of}\ a^*\ \mathsf{default}{:}\mu_2 \mid \mathsf{Let}\ t{::}\kappa{=}\mu_1\ \mathsf{in}\ \mu_2 \mid$ $\mathsf{Nil} \mid \mathsf{Cons}(\mu_1, \mu_2) \mid \mathsf{Fold}\ \mu_1\ \mathsf{of}\ \mathsf{Nil}{:}\mu_2\ \mathsf{Cons}{:}cf \mid$ $\mathsf{Listcase}\ \mu_1\ \mathsf{of}\ \mathsf{Nil}{:}\mu_2\ \mathsf{Cons}{:}cf \mid \mathsf{Tuple}(\mu^*) \mid \mathsf{Proj}(i, \mu)$
(arity-0 con.)	c_0	$::=$	$\mathsf{Int} \mid \mathsf{Real} \mid \mathsf{String} \mid \mathsf{Intarray} \mid \mathsf{Realarray} \mid \mathsf{Exn} \mid \mathsf{Enum}\ i$
(arity-1 con.)	c_1	$::=$	$\mathsf{Ptarray} \mid \mathsf{Arrow} \mid \mathsf{Record} \mid \mathsf{Sum} \mid \mathsf{Enumorrec}\ i \mid$ $\mathsf{Enumorsum}\ i \mid \mathsf{Excon} \mid \mathsf{Deexcon}$
(typecase arm)	a	$::=$	$c_0{:}\mu \mid c_1 : cf$
(con. functions)	cf	$::=$	$\lambda t{::}\kappa.\mu$
(types)	σ	$::=$	$\mathsf{T}\ (\mu) \mid \forall(t{::}\kappa)^*.\sigma \mid \mathsf{int} \mid \mathsf{real} \mid \mathsf{string} \mid \mathsf{realarray} \mid \mathsf{stringarray} \mid$ $\mathsf{exn} \mid \mathsf{enum}\ i \mid \mathsf{ptrarray}\ \sigma \mid \sigma_1 \rightarrow \sigma_2 \mid \mathsf{sum}\ [\sigma_1 ... \sigma_n] \mid$ $\mathsf{enumorrec}\ i[\sigma_1 ... \sigma_n] \mid \mathsf{enumorsum}\ i[\sigma_1 ... \sigma_n] \mid \mathsf{record}\ [\sigma_1 ... \sigma_n] \mid$ $\mathsf{excon}\ \sigma \mid \mathsf{deexcon}\ \sigma$

Figure 4.2: Abstract syntax for kinds and types of LMLI

- Listcase takes a list of constructors. If the list is Nil, it evaluates μ_2. Otherwise, it applies the constructor function cf to the pair of the first element and the rest of the list.

The introductory and eliminatory forms for tuples of constructors are also straightforward:

- Tuple creates a tuple of constructors.

- Proj projects the ith constructor from the tuple.

The arity-0 constructors include

- integers,

- floating point numbers,

- strings,

- integer arrays,

- real arrays,

- and exception packets.

Enum i is a special case of a sum type: it represents enumerated types. For example, the SML type `datatype d = RED | BLUE` is represented as the type Enum 2 in LMLI.

Arity-1 constructors include the following:

- Ptrarray, which constructs pointer arrays. Pointer arrays are arrays of values that are represented at the machine level as pointers. These include arrays of all types except Int and Real. The different array types allow LMLI to use specialized representations for arrays. In particular, values stored in arrays of type Intarray and Realarray are never boxed.

- Arrow, which constructs function types given a pair of constructors.

- Record which constructs a record type given a list of constructors.

- Sum, which constructs a sum type given a list of constructors.

- Enumorrec and Enumorsum, which construct special cases of sum types.
  - Enumorrec i represents the special case of a sum where i of the cases carry nullary types, and the final case carries a record type.
  - Enumorsum is the analogous sum where the final case is a sum type.

For example, the SML type `datatype d = NONE | SOME of int * int` is represented as

$$\mathsf{Enumorrec}\ 1(Cons(\mathsf{Int}, Cons(\mathsf{Int}, Nil)))$$

The special sum types Enum, Enumorrec, and Enumorsum allow LMLI to directly express the various strategies for representing SML datatypes given by Appel [7].

The Rectype constructor defines recursive types. It takes a list of variables and constructors and a constructor μ, binds the variables mutually recursively to the constructors, and places μ in their scope. For example, the SML type

```
datatype intlist = nil | cons of int * intlist
```

is translated to

```
Rectype intlist=Enumorsum 1(Cons(Record(Cons(int,intlist)),Nil)) in intlist
```

LMLI as implemented in TIL supports Typerec. However, Typerec currently is not used in TIL. The simpler form Typecase is used instead. Typecase takes a list of arms and a default arm. Given a monotype μ of the form p_0, it selects the arm corresponding to p_0. Given a monotype μ of the form $p_1(\mu_1)$, it selects the arm corresponding to p_1 and applies the constructor function to μ_1. If no arm corresponds to the monotype, it selects the default arm.

Term-level expressions and declarations

In Figure 4.3, I give the syntax of term-level expressions and declarations of LMLI. Every expression is annotated with its type. Expressions include floating-point constants (ranged over by r), string constants (ranged over by s), values of enumerated sums, primitive operators of one argument, primitive operators of two arguments, and coercions (these operators are described below).

The expression `update`(ar, e_1, e_2, e_3) updates an array. It evaluates e_1, e_2 and e_3 to the values A, i, and v and updates the i^{th} value of A to be v. `update` does no bounds checking. Also note that `update` is a specialized array operator: there are three different forms of the operator: one for integer arrays, another for floating point arrays, and another version for pointer arrays.

The expression `extern`(s,σ) denotes an external identifier of type σ. The expressions `make_vararg` and `call_vararg` are used to implement functions with variable numbers of arguments [63]. The expressions `eq` and `neq` implement structural polymorphic equality and inequality. They each take a constructor as an argument, and return an equality and inequality function, respectively. These operators can actually be coded in LMLI, but they are easier to optimize if they are kept as primitive operators. At the end of optimization, in fact, they are replaced by LMLI expressions.

The expression `record` creates a record from a list of expressions. The expression `inject` creates a sum, where each case of the sum takes multiple arguments. Sums are deconstructed using `switch`, which is also overloaded to allow branching on integer values. For sums, `switch` examines the case for the sum. It then selects the arm corresponding to the case,

(expressions)	e	$::=$	$re : \sigma$
(bindings)	b	$::=$	$x : \sigma = e \mid t :: \kappa = \mu \mid \texttt{fix}\ (x : \sigma = f)^* \mid$ $\texttt{fixtype}\ (x : \sigma = tf)^*$
(raw exp.)	re	$::=$	$x \mid i \mid r \mid s \mid \texttt{enum}\ i \mid p_1(e) \mid p_2(e_1, e_2) \mid cp(e) \mid$ $\texttt{update}(ar, e_1, e_2, e_3) \mid \texttt{extern}(s, \sigma) \mid \texttt{new\_exn}\ \mu \mid$ $\texttt{make\_vararg}(\mu, e) \mid \texttt{call\_vararg}(\mu_1, e_1, e_2) \mid$ $\texttt{eq}\ \mu \mid \texttt{neq}\ \mu \mid \texttt{record}(e^*) \mid \texttt{inject}(i, e^*) \mid$ $\texttt{switch}(st)\, e\ \texttt{of}\ (i : f)^*\ \texttt{default} : oe \mid$ $\texttt{typecase}\, \mu\ \texttt{of}\, [t.\sigma]\, ta^*\, \texttt{rectype} : otf\, \texttt{default} : otf \mid$ $\texttt{tlistcase}\ \mu\ \texttt{of}\, [t.\sigma]\, \texttt{nil} : e\, \texttt{cons} : tf \mid \texttt{let}\ b\ \texttt{in}\ e \mid$ $f \mid tf \mid e(e^*) \mid e[\mu^*] \mid \texttt{raise}\ e \mid \texttt{handle}\ (e, f)$
(functions)	f	$::=$	$\lambda(x : \sigma)^*.e$
(type fun.)	tf	$::=$	$\Lambda(t :: \kappa)^*.e$
(switch type)	st	$::=$	$\mathsf{Int} \mid \mathsf{Sum} \mid \mathsf{Enum} \mid \mathsf{Enumorrec} \mid \mathsf{Enumorsum}$
(typecase arm)	ta	$::=$	$c_0 : e \mid c_1 : tf$
(opt. exp)	oe	$::=$	$e \mid \epsilon$
(opt. type fun.)	otf	$::=$	$tf \mid \epsilon$

Figure 4.3: Abstract syntax for term-level expressions and declarations of LMLI

(arity 1 op.)	p_1	::=	`real` | `not` | `floor` | `sqrt` | `sin` | `cos` | `arctan` | `exp` | `ln` | `size` *ar* | `select` *i* |
(arity 2 op.)	p_2	::=	`divi` | `muli` | `plusi` | `minusi` | `modi` | `eqi` | `lti` | `gti` | `ltei` | `gtei` | `divui` | `mului` | `plusui` | `minusui` | `ltui` | `gtui` | `lteui` | `gteui` | `divr` | `mulr` | `plusr` | `minusr` | `eqr` | `ltr` | `gtr` | `lter` | `gter` | `alloc` *ar* | `sub` *ar* | `excon` | `de_excon` | `eqptr`
(coercions)	*cp*	::=	`roll` | `unroll` | `enum_enumorrec` | `rec_enumorrec` | `enum_enumorsum` | `sum_enumorsum` | `enum2int` | `fromstring` | `tostring` | `chr`
(arrays)	*ar*	::=	`Intarray` | `Realarray` | `Ptrarray`

Figure 4.4: Primitive term-level operations for LMLI

if one matches, and applies the function for the arm to the values from which the sum was constructed. If no arm matches, it evaluates the default expression.

For example, a switch expression of the form

```
switch(Sum) (inject(2,["hello",5])) of
    ...
    2 : λx,y.e
   ...
```

selects the arm corresponding to 2, since the case of the sum is 2. It then binds x and y to the values `"hello"` and `5` respectively, and evaluates e. `inject` creates values of a sum type, so it is referred to as the introductory form for sums. `switch` deconstructs values of a sum type, so it referred to as the eliminatory form for sums.

The expression `typecase` branches on the value of a constructor at run time. The constructor is required to be a monotype, that is, have kind Ω. If the constructor is a primitive constructor of 0 arguments, the corresponding arm is selected (if there is one) and the associated expression is evaluated. If the constructor is a primitive constructor of 1 argument, say $c_1(c_2)$, the arm corresponding to c_1 is selected (if there is one), and the associated type function is applied to c_2. If the constructor is a recursive constructor, the constructor is unrolled once, and the function associated with the `rectype` arm (if there is one) is applied to the unrolled constructor. If no arm matches, the default arm is used.

The `typerec` expression in λ_i^{ML} is not provided at the term-level in LMLI, but can be written using `typecase` and a recursive function definition. Hence, it is redundant at the term-level in LMLI.

The `tlistcase` expression branches on lists of constructors. The `let` declaration allows binding of expressions to variables, binding of constructors to constructor variables, definition of mutually-recursive functions, and definition of mutually-recursive polymorphic functions. Polymorphic functions take constructors as arguments.

LMLI allows the definition of functions of multiple arguments. A function of multiple arguments is written $\lambda x, y, z.e$, where x,y,and z are the multiple arguments. A function f applied to multiple arguments is written $f(1,2,3)$. A polymorphic function applied to multiple arguments is written $f[Int, Int, Int]$.

`raise` raises an exception. `handle`(e, f) evaluates an expression e and applies f if an exception is raised but not caught during the evaluation of e.

In Figure 4.4, I list the primitive operators and coercions. The single argument primitive operators include `real` (convert an integer to a floating-point number) `not` (bitwise integer negation), `floor` (convert a floating-point number to an integer), various operations on floating-point numbers, `size` (size of a 1-dimensional array), and `select` (select the i^{th} field of a record). The primitive operators of two arguments include the usual arithmetic and comparison operators on integers. The integer comparison operators are `eqi` (equality), `lti` (less than), `gti` (greater than), `ltei` (less than or equal), and `gtei` (greater than or equal). The arithmetic and comparison operators for unsigned integers (ending in `ui`) and floating point numbers (ending in `r`) are similar. The operator `alloc`(i, v) creates a 1-dimensional array of size i and initializes it to v. The operator `sub`(a, i) returns the i^{th} element of a 1-dimensional operator. The `alloc` and `sub` operators are unsafe: they do no bounds checking. LMLI does not provide multi-dimensional arrays; they must be implemented using 1-dimensional arrays.

The coercion operators coerce values in a type-safe mannner from one type to another; they have no effect at run time. The operators `roll` and `unroll` are used to type values of recursive types. The operators `enum_enumorrec` and `rec_enumorrec` inject values of enumerated types and values of record type, respectively, into `enumorrec` sums. The operators `enum_enumorsum` and `sum_enumorsum` provide similar operations for `enumorsum` sums. The operators `tostring` and `fromstring` coerce values of string types to the actual representation of strings. LMLI uses a string representation similar to that used by Appel [7]. Strings are represented as either a character or a pair consisting of the length of the string (in bytes) and an integer array. Characters are simply an enumerated type, so the string representation is simply

```
Enumorrec 256 [Record(Int,Intarray)]
```

The operator `enum2int` coerces an enumerated type to an integer, while the operator `chr` coerces an integer to a string (*i.e.* a value of an enumerated type between 0 and 255).

4.1.3 An example

Figure 4.5 illustrates the passing and construction of constructors in an LMLI program. The binding `con list` binds `list` to a constructor corresponding to the SML list datatype. It is a constructor function that when given a constructor c as an argument, builds a recursive constructor that represents the list type instantiated to c. The binding `val map` defines a map function for lists, which maps a function `f` across a list. The map function first takes two constructors as arguments. The binding `val v` applies this map function to convert a

```
con list:: Ω → Ω = λt :: Ω.Rectype t'=Enumorrec 1 ([t,t']) in t'

val map : ∀ α,β.(α  →  β)  →  T(list α)  →  T(list β) =
    Λ(α,β).λf.
       fix m = λy.
         switch(Enumorrec) y
         of 0: enum 0
          | 1: λz.record [f (#0 z),m (#1 z)]

val v : T(list String) = map [Int,String] (λx.makestring x) [1,2,3]

val y : T(list (list String)) = map [String,list String] (λx.[x]) v
```

Figure 4.5: Application of the polymorphic function map illustrates passing and construction of constructors at run time

list of integers to a list of strings. The function is first applied to two constructors before being applied to its other arguments. The binding `val y` applies the map function to convert the list of strings to a list of lists of strings. When the map function is applied here, the second constructor actually will be the result of a *constructor application*. At run time, the constructor function `list` will be applied to the constructor `String` to build a constructor which represents the the SML type string list.

4.2 An overview of TIL

Figure 4.6 shows the various compilation phases of TIL. The phases through and including closure conversion use typed intermediate language that are variants of LMLI. The phase after closure conversion use an untyped language where variables are annotated with garbage collection information. The low-level phases of the compiler use languages where registers are annotated with garbage collection information.

4.2.1 Front end

The first phase of TIL uses the front-end of the ML Kit Compiler [15] to parse and elaborate (type check) SML source code. The ML Kit Compiler produces annotated abstract syntax for all of SML, which it typechecks using the Hindley-Milner typechecking algorithm. It then compiles a subset of this abstract syntax to an explicitly-typed language called Lambda. The compilation to Lambda converts pattern matching to decision trees and expands various derived forms according to [61].

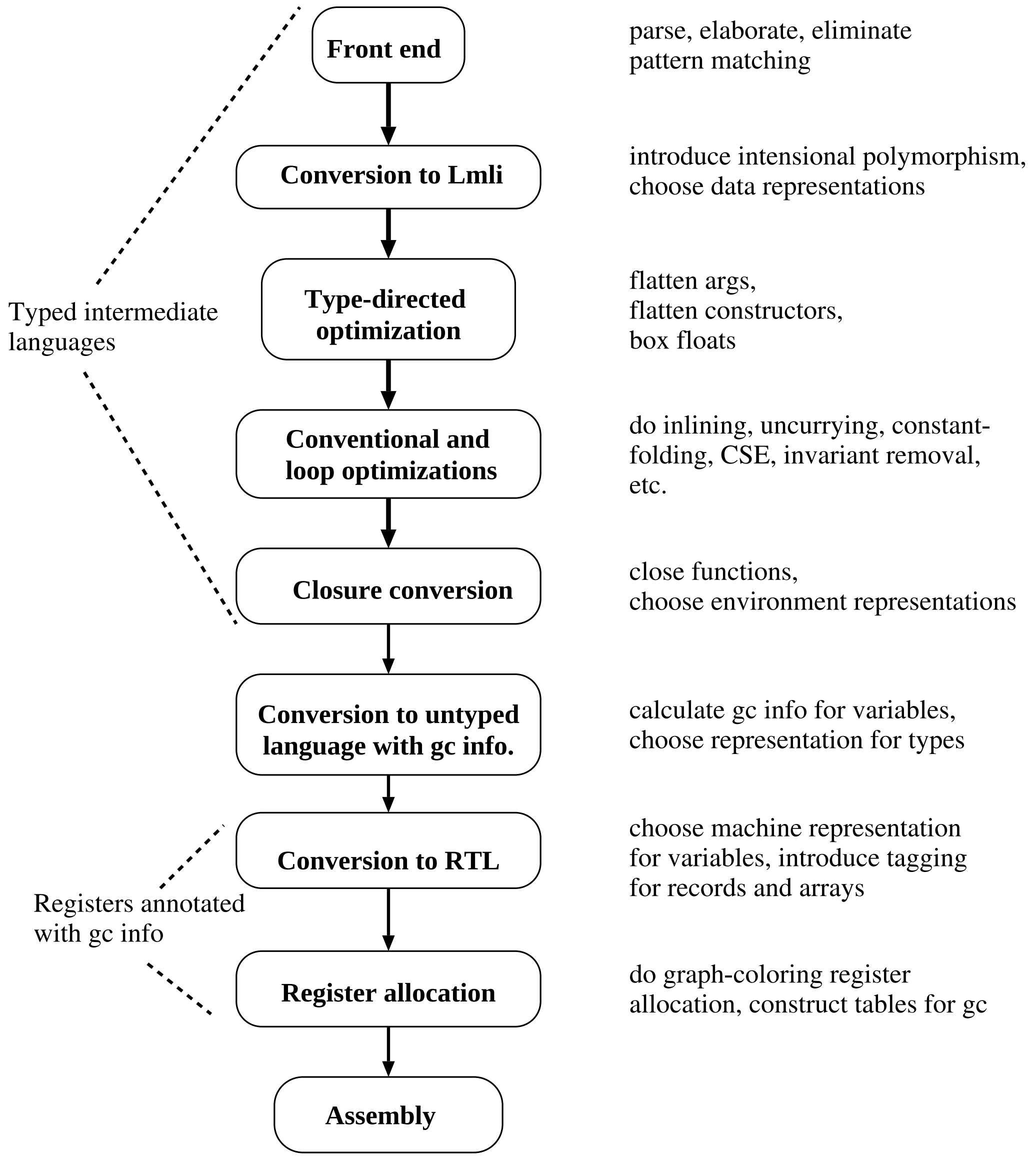

Figure 4.6: Phases of the TIL compiler

4.2.2 Translation to LMLI

The next phase of TIL translates Lambda programs to LMLI programs and uses the intensionally polymorphic language constructs of LMLI to provide specialized arrays, multi-argument functions, efficient data representations for user-defined datatypes, and tag-free polymorphic equality.

The translation to LMLI specializes arrays into one of three cases: int arrays, float arrays, and pointer arrays. It replaces the polymorphic array primitives with new versions that use intensional polymorphism to select the appropriate creation, subscript, and update operations for polymorphic arrays.

For example, the translation replaces polymorphic subscript with a new version that uses a `typecase` expression to determine the type of the array and to select the appropriate specialized subscript operation:

```
val newsub = Λα.λ(x:Typecase α ..., i:int).
             typecase α of
              Int : sub Intarray (x, i)
              Real : sub Realarray (x, i)
              default : sub PtrArray (x, i)
```

Note that if the type of the array can be determined at compile-time, the optimizer can eliminate the `typecase`:

```
newsub[Real](a, 5) ↪ sub Realarray(a, 5)
```

The translation to LMLI also makes polymorphic equality explicit as a term in the language.

After this phase translates an SML program to an LMLI program, it does several *type-directed* optimizations. A type-directed optimization is an optimization that uses the *type* of an expression to determine how to translate the expression. Thus, the following optimizations are applied even when some call sites of a function are unknown.

SML provides only single-argument functions; multiple arguments are passed in a record. The first optimization, *argument flattening*, translates each function that takes a record as an argument to a function that takes the components of the record as multiple arguments. These arguments are passed in registers, avoiding allocation to create the record and memory operations to access record components. If a function takes an argument of variable type α, then TIL uses `typecase` to determine the proper calling convention, according to the instantiation of α at run time.

As with functions, datatype constructors in SML take a single argument. For example, the cons data constructor (`::`) for an α list, takes a single record, consisting of an α value and an α list value. Naively, such a constructor is represented as a pair, consisting of a tag (e.g., `cons`), and a pointer to the record containing the α value and the α list value. The tag is a small integer value used to distinguish among the constructors of a datatype (e.g., `nil` vs `::`). The second optimization, *constructor flattening*, rewrites all constructors that take records as arguments so that the components of the records are flattened.

In addition, constructor flattening eliminates unneeded tag components by specializing sums to `Enumorrec` or `Enumorsum` sums. For example, it specializes the sum used in the definition of the `list` datatype to an `Enumorrec` sum. This corresponds to dropping the tag from the flattened representation of the `cons` constructor. After naive constructor flattening, cons applied to `(hd,tl)` is represented as a three element record consisting of the tag, `hd`, and `tl`. The tag is unnecessary, because the record (represented as a pointer) can always be distinguished from `nil`.

The third optimization boxes all floating-point values, except for values stored in floating-point arrays. This makes record operations faster: if floating-point values were unboxed, then record offset calculations could not always be done at compile-time. The problem is that floating-point values are 64 bits, unlike scalars and pointers, which are 32 bits. If floating-point values were unboxed, then the sizes of fields whose types were polymorphic type variables would not be known until run time.

Straight-line floating point code still runs fast: the optimizer later eliminates unecessary box/unbox operations during the constant-folding phase.

In all, the combination of type-directed optimizations reduces running times by roughly 40% and allocation by 50% [63, Chapter 8]. However, much of this improvement can be realized by other techniques; For example, SML/NJ uses Leroy's unboxing technique to achieve comparable improvements for calling conventions [81]. The advantage of the TIL approach is that it uses a single mechanism (intensional polymorphism) to specialize calling conventions, flatten constructors, unbox floating-point arrays, and eliminating tags for both polymorphic equality and garbage collection.

4.2.3 Optimizations

TIL employs an extensive set of optimizations, which are described in Chapters 5 and 6.

Before optimization, TIL translates Lmli to a subset of Lmli called B-form. B-form, based on A-Normal-Form [33], is a more regular intermediate language than Lmli that facilitates optimization. The translation from Lmli names all intermediate computations and binds them to variables by a `let`-construct. It also names all potentially heap-allocated values, including strings, records and functions. Finally, it allows nested `let` expressions only within switches (branch expressions). Hence, the translation from Lmli to B-form linearizes and names nested computations and values.

4.2.4 Closure conversion

TIL uses a type-directed, abstract closure conversion in the style suggested by Minamide, Morrisett, and Harper [62] to convert B-form programs to Lmli-Closure programs. Lmli-Closure is an extension of B-form that provides constructs for explicitly constructing closures and their environments.

For each escaping B-form function, TIL generates a closed piece of code, a type environment, and a value environment. The code takes the free type variables and free value variables of the original function as extra arguments. The types and values corresponding to these free variables are placed in records. These records are paired with the code to form an abstract closure. TIL uses a flat environment representation for type and value environments [7].

For known functions, TIL generates closed code but avoids creating environments or a closure. Following Kranz [52], we modify the call sites of known functions to pass free variables as additional arguments.

TIL closes over only variables which are function arguments or are bound within functions. The locations of other "top-level" variables are resolved at compile-time through traditional linking, so their values do not need to be stored in a closure.

4.2.5 Conversion to an untyped language

To simplify the conversion to low-level assembly code, TIL translates Lmli-Closure programs to an untyped language called U-Bform. This conversion is described in detail in Section 7.3.

U-Bform is a much simpler language than Lmli because similar type-level and term-level constructs are collapsed to the same term-level constructor. For example, in the translation from Lmli-Closure to Ubform, TIL replaces `typecase` with a conventional `switch` expression. This simplifies the generation of low-level code, because there are many fewer cases.

TIL annotates variables with *representation information* that tells the garbage collector what kinds of values variables must contain (*e.g.*, pointers, integers, floats, or pointers to code). The representation of a variable x may be unknown at compile time, in which case the representation information is the name of the variable y that will contain the type of x at run time.

4.2.6 Conversion to RTL

Next TIL converts U-Bform programs to RTL,a register-transfer language similar to ALPHA or other RISC-style assembly language. This conversion is described in detail in Section 7.4.

RTL provides an infinite number of pseudo-registers each of which is annotated with representation information. Representation information is extended to include locatives, which are pointers into the middle of objects. Pseudo-registers containing locatives are never live across a point where garbage collection can occur. RTL also provides heavy-weight function call and return mechanisms, and a form of interprocedural goto for implementing exceptions.

The conversion of U-Bform to RTL decides whether U-Bform variables will be represented as constants, labels, or pseudo-registers. It also eliminates exceptions, inserts tagging operations for records and arrays, and inserts garbage collection checks.

4.2.7 Register allocation and assembly

Before doing register allocation, TIL converts RTL programs to ALPHA assembly language with extensions similar to those for RTL. Then TIL uses conventional graph-coloring register allocation to allocate physical registers for the pseudo-registers. It also generates tables describing layout and garbage collection information for each stack frame, as described in Section 4.2.8. Finally, TIL generates actual ALPHA assembly language and invokes the system assembler, which does instruction scheduling and creates a standard object file.

4.2.8 Nearly Tag-Free Garbage Collection

Nearly tag-free garbage collection uses type information to eliminate data representation restrictions due to garbage collection. The basic idea is to record enough representation information at compile time so that, at any point where a garbage collection can occur, it is possible to determine whether or not values are pointers and hence must be traced by the garbage collector. Recording the information at compile time makes it possible for code to use untagged representations. Unlike so-called conservative collectors (see for example [16, 26]), the information recorded by TIL is sufficient to collect all unreachable objects.

Collection is "nearly" tag-free because tags are placed only on heap-allocated data structures (records and arrays); values in registers, on the stack, and within data structures remain tagless. TIL constructs the tags for monomorphic records and arrays at compile time. For records or arrays with unknown component types, TIL generates code that (partially) constructs tags at run time. As with other polymorphic operations, we use intensional polymorphism to construct these tags.

Registers and components of stack frames are not tagged. Instead, TIL generate tables at compile time that describe the layout of registers and stack frames. TIL associates these tables with the addresses of call sites within functions at compile time. When garbage collection is invoked, the collector scans the stack, using the return address of each frame as an index into the table. The collector looks up the layout of each stack-frame to determine which stack locations to trace. TIL records additional liveness information for each variable so that the garbage collector can avoid tracing pointers that are no longer needed.

This approach is well-understood for monomorphic languages requiring garbage collection [17]. Following Tolmach [90], we extended it to a polymorphic language as follows: when a variable whose type is unknown is saved in a stack frame, the type of the variable is also saved in the stack frame. However, unlike Tolmach, we evaluate substitutions of ground types for type variables eagerly instead of lazily. This is due in part for technical reasons (see [63, Chapter 7]), and in part to avoid a class of space leaks that might result with lazy substitution.

4.3 An example

This section shows an SML function as it passes through the various stages of TIL. The following SML code defines a dot product function that is the inner loop of the integer matrix multiply benchmark:

```
val sub2 : 'a array2 * int * int -> 'a

fun dot(cnt,sum) =
  if cnt<bound then
   let val sum'=sum+sub2(A,i,cnt)*sub2(B,cnt,j)
   in dot(cnt+1,sum')
   end
 else sum
```

The function sub2 is a built-in 2-d array subscript function which the front end expands to

```
fun sub2 ({columns,rows,v}, s :int, t:int) =
  if s <0 orelse s>=rows orelse t<0 orelse
  t>=columns then raise Subscript
  else unsafe_sub1(v,s * columns + t)
```

Figures 4.7 through 4.12 show the actual intermediate code created as dot and sub2 pass through the various stages of TIL. For readability, I have renamed variables, erased type information, and performed some minor optimizations, such as eliminating selections of fields from known records.

Figure 4.7 shows the functions after they have been converted to Lmli. The sub2 function takes a type as an argument. A function parameterized by a type is written as Λt., while a function parameterized by a value is written as λi. In the dot function, the sub2 function is first applied to a *type* and then applied to its actual values. Each function takes only one argument, often a record, from which fields are selected. The quality of code at this level is quite poor: there are eight function applications, four record constructions, and numerous checks for array bounds.

Figure 4.8 shows the Lmli fragment after it has been converted to B-form. Functions have been transformed to take multiple arguments instead of records and every intermediate computation is named.

Figure 4.9 shows the B-form fragment after it has been optimized. All the function applications in the body of the loop have been eliminated. sub_ai(av,a) is an application of the (unsafe) integer array subscript primitive. All of the comparisons for array bounds checking have been safely eliminated, and the body of the loop consists of 9 expressions. This loop could be improved even further; TIL does not implement any form of strength reduction and induction variable elimination.

Figure 4.10 shows the B-form fragment after it has been converted to U-Bform. Each variable is now annotated with *representation information*, to be used by the garbage collector. INT denotes integers and TRACE denotes pointers to tagged objects. The function is now *closed*, since it was closure converted before converting to U-Bform.

Figure 4.11 shows the U-Bform fragment after it has been converted to RTL. Every pseudo-register is now annotated with precise representation information for the collector. The representation information has been extended to include LOCATIVE, which denotes pointers into the middle of tagged objects. Locatives cannot be live across garbage-collection points. The (*) indicates points where the sub_ai primitive has been expanded to two RTL instructions. This indicates that induction-variable elimination would also be profitable at the RTL level. The return instruction's operand is a pseudo-register containing the return address.

Figure 4.12 shows the actual DEC ALPHA assembly language generated for the dot function. The code between L1 and L3 corresponds to the RTL code. The other code is epilogue and prologue code for entering and exiting the function. Note that no tagging operations occur anywhere in this function.

```
sub2 =
   let fix f = Λty.
        let fix g = λarg.
            let a = (#0 arg)
                s = (#1 arg)
                t = (#2 arg)
                columns = (#0 a)
                rows = (#1 a)
                v = (#2 a)
                check =
                  let test1 = lti(s,0)
                  in Switch_enum test of
                       1 => λ.enum(1)
                     | 0 => λ.
                        let test2 = gti(s,rows)
                        in Switch_enum test2 of
                             1 => λ.enum(1)
                           | 0 => λ.
                             let test3 = lti(t,0)
                             in  Switch_enum test3 of
                                 1 => λ.enum(1)
                               | 0 => λ.gti(t,columns)
                             end
                        end
                  end
            in Switch_enum check of
                  1 => λ.raise Subscript
                | 0 => λ.unsafe_sub1 [ty] {v,t + s * columns}
            end
        in g
   in f
   end

fix dot=
    λi.let cnt = (#0 i)
            sum = (#1 i)
            d = lti(cnt,bound)
        in  Switch_enum d
            of 1 => λ.let sum' = sum +
                               ((sub2 [Int]) {A,i,cnt}) *
                              ((sub2 [Int]) {B,cnt,j})
                           in  dot{cnt+1,sum'}
                           end
             |  0 => λ.sum
        end
```

Figure 4.7: After conversion to Lmli

```
sub2 = ...
fix dot = λcnt,sum.
          let test = lti(cnt, bound)
              r =
               Switch_enum test of
                   1 => λ.
                    let a = sub2[Int]
                        b = a(A,i,cnt)
                        c = sub2[Int]
                        d = c(B,cnt,j)
                        e = b*d
                        f = sum+e
                        g = cnt+1
                        h = dot(g,f)
                    in h
                    end
                 | 0 => λ.sum
          in  r
          end
```

Figure 4.8: B-form before optimization

```
fix dot =
     λcnt,sum.
      let test = lti(cnt,bound)
          r = Switch_enum test of
                 1 =>
                 λ.
                  let a = t1 + cnt
                      b = sub_ai(av,a)
                      c = columns * cnt
                      d = j + c
                      e = sub_ai(bv,d)
                      f = b*e
                      g = sum+f
                      h = 1+cnt
                      i = dot(h,g)
                  in  i
                  end
              |  0 => λ.sum
      in  r
      end
```

Figure 4.9: Bform after optimization

```
fix dot =
 λbound:INT,columns:INT,bv:TRACE,av:TRACE,t1:INT,
    j:INT,cnt:INT,sum:INT.
            let test:INT = gti(bound,cnt)
                r:INT =
                  Switchint test of
                    1 =>
                    let a:INT = t1 + cnt
                        b:INT = sub_ai(av,a)
                        c:INT = columns * cnt
                        d:INT = j + c
                        e:INT = sub_ai(bv,d)
                        f:INT = b*e
                        g:INT = sum+f
                        h:INT = 1+cnt
                        i:INT = dot(bound,columns,bv,
                                    av,t1,j,h,g)
                    in i
                    end
                 | 0 => sum
            in  r
            end : INT
```

Figure 4.10: After conversion to U-Bform

```
dot(([bound(INT),columns(INT),bv(TRACE),
    av(TRACE),t1(INT),j(INT),cnt(INT),
    sum(INT)],[]))
{ L0:  gti bound (INT) , cnt(INT) , test(INT)
        bne test(INT),L1
        mv sum(INT),result (INT)
        br L2
   L1:  addl t1(INT) , cnt(INT) , a(INT)
(*)     s4add a(INT) , av(TRACE) , t2(LOCATIVE)
(*)     ldl b(INT) , 0(t2(LOCATIVE))
        mull columns(INT) , cnt(INT) , c (INT)
        addl j(INT) , c(INT) , d (INT)
(*)     s4add d (INT) , bv(TRACE) , t3(LOCATIVE)
(*)     ldl e (INT) , 0(t3 (LOCATIVE))
        mull/v b (INT) , e (INT) , f(INT)
        addl/v sum(INT) , f (INT) , g (INT)
        addl/v cnt(INT) , 1 , h (INT)
        trapb
        mv h (INT),cnt(INT)
        mv g (INT),sum(INT)
        br L0
   L2:  return retreg(LABEL) }
```

Figure 4.11: After conversion to RTL

```
        .ent Lv2851_dot_205955
 # arguments : [$bound,$0] [$columns,$1] [$bv,$2]
 #             [$av,$3] [$t1,$4] [$j,$5]
 #             [$cnt,$6] [$sum,$7]
 # results    : [$result,$0]
 # return addr  : [$retreg,$26]
 # destroys   : $0 $1 $2 $3 $4 $5 $6 $7 $27
Lv2851_dot_205955:
        .mask (1 << 26), -32
        .frame $sp, 32, $26
        .prologue 1
        ldgp        $gp, ($27)
        lda        $sp, -32($sp)
        stq        $26, ($sp)
        stq        $8, 8($sp)
        stq        $9, 16($sp)
        mov        $26, $27
L1:
        cmplt        $6, $0, $8
        bne        $8, L2
        mov        $7, $1
        br        $31, L3
L2:
        addl        $4, $6, $8
        s4addl        $8, $3, $8
        ldl        $8, ($8)
        mull        $1, $6, $9
        addl        $5, $9, $9
        s4addl        $9, $2, $9
        ldl        $9, ($9)
        mullv        $8, $9, $8
        addlv        $7, $8, $7
        addlv        $6, 1, $6
        trapb
        br        $31, L1
L3:
        mov        $1, $0
        mov        $27, $26
        ldq        $8, 8($sp)
        ldq        $9, 16($sp)
        lda        $sp, 32($sp)
        ret        $31, ($26), 1
        .end Lv2851_dot_205955
```

Figure 4.12: Actual DEC ALPHA assembly language

4.4 Conclusion

In this chapter, I have given an overview of the TIL compiler and the typed intermediate language framework that it uses. I have described LMLI, the intensionally-polymorphic intermediate language used in TIL. I have given an example of an LMLI program that illustrates the use of intensionally-polymorphic constructs, namely the construction, passing, and analysis of types at run time.

I also have described each phase of the TIL compiler and given an example that illustrates how TIL compiles SML programs to machine code.

Chapter 5

The TIL optimizer

In this chapter, I describe TIL's optimizer. The optimizer is based on the SML/NJ optimizer [7, 10]. It includes all the optimizations that Appel found to be important for SML [7]. However, it uses improved algorithms for inlining and uncurrying. I improved these algorithms because it is particularly important to eliminate higher-order functions before doing loop optimizations [83].

I have organized this chapter as follows. First, I discuss the intermediate language used by the optimizer, B-form. B-form is a subset of LMLI, the intermediate language that I described in Chapter 4. In contrast to the intermediate language that Appel uses, B-form is a direct-style, typed intermediate language. Second, I discuss the notation that I use to describe algorithms and data structures. Third, I discuss some assumptions that I make throughout the optimizer. Fourth, I discuss an effects analysis that I use in the optimizer. Fifth, I discuss the algorithms for inlining and uncurrying in-depth. Finally, I cover the other optimizations used in the optimizer. I defer my discussion of the order in which optimizations are applied until after I have discussed the loop optimizations.

5.1 B-form intermediate language

In this section I discuss B-form, the intermediate language used by the optimizer. B-form is a subset of LMLI that is a typed, direct-style intermediate language similar to A-Normal-Form [33]. I use a subset of LMLI instead of LMLI to simplify the writing of the optimizer. As I will explain, the subset is more easily manipulated by the optimizer.

It is important to design the intermediate language for an optimizer carefully. In designing an intermediate language, there is a tension between choosing a language that retains information and simplifying the language by discarding information. A complex language that retains information makes an optimizer larger and harder to write, but a simpler language may discard information that is difficult to recover once it is thrown away.

In choosing to use a direct-style, typed intermediate language, I have chosen to favor a language that retains information. Other compilers for functional languages have taken the route of simplifying the language — but losing information — by either using a continuation-passing style intermediate language or an untyped intermediate language.

In Sections 5.1.1 and 5.1.2, I discuss why I chose to use a typed language instead of an untyped language and a direct-style language instead of a continuation-passing style language. In Section 5.1.3, I cover practical issues of naming and simplicity. Finally, in Section 5.1.4, I describe B-form.

5.1.1 Typed versus untyped

One important design choice for an intermediate language is whether the language is typed or untyped. For example, the Glasgow Haskell optimizer uses a typed language, but the SML/NJ optimizer uses an untyped intermediate language. Most compilers use an untyped intermediate language.

There are several advantages to my choice of an intensionally-polymorphic typed intermediate language during optimization. First, distinguishing constructor-level computations from term-level computations is useful during optimization, because the constructor-level computations always terminate and are side-effect free. This makes code motion easier for constructor-level computations than term-level computations. Second, it helps me find bugs in the optimizer — I can typecheck programs after every optimization phase.

On the other hand, using a typed intermediate language has several potential disadvantages. First, the typed language has more constructs than a corresponding untyped language. This makes the optimizer larger and means that it takes more time to write the optimizer. Second, typed programs can be much larger than corresponding untyped programs. This means the optimizer may use more memory and be slower. Third, types can constrain optimization.

I claim that the advantages of a typed intermediate language outweigh the potential disadvantages. It may require more time to write the code for the optimizer because the code needs to handle more constructs, but it requires less time to debug the code for the optimizer: the typed intermediate language makes it easier to find and isolate bugs. In Chapter 10, I show that types do not unreasonably increase intermediate program sizes. With appropriate techniques that are not difficult to implement, typed programs are usually no more than twice the size of their untyped versions. At the end of optimization, typed programs are only 15% larger than their corresponding untyped versions.

5.1.2 Direct-style versus continuation-passing style

Another important design choice for an intermediate language is whether the language is direct-style (DS) or continuation-passing style (CPS). For example, the Glasgow Haskell optimizer uses a DS language, but the SML/NJ optimizer uses a CPS language. Most compilers use a DS intermediate language.

From a theoretical point of view, the two languages are essentially equivalent: programs can be converted from one representation to the other in asymptotically-efficient time [25, 77, 78]. The choice of which representation to use is a matter of engineering.

I chose to use a DS intermediate language. This has two advantages. First, a DS language is more amenable to the kinds of interprocedural program analyses that I want to do than a CPS language. For code motion optimizations, I want to determine the run-time nesting of recursive functions, which requires a distinction between function call and return. It is unclear how to make this distinction when analyzing CPS programs, because functions never return. Second, a DS language makes it possible to re-use existing compilation techniques, such as interprocedural register allocation.

On the other hand, using a DS language has a disadvantage. Sabry and Felleisen [78] showed that the β_v rule (inlining) is less powerful for DS programs than for CPS programs. They showed that you need several additional rules (that is, optimizations) for DS programs.

These optimizations, however, are not onerous to implement. I claim that the advantages of a DS language — in particular, being able to re-use existing compilation techniques — more than offset the disadvantage of having to write several additional optimizations.

5.1.3 Practical engineering issues

The intermediate language should satisfy some practical requirements also. First, any program term about which the optimizer may need to store information should be named. For example, the language should name every function. Second, the language should not have redundant constructs because this requires redundant code in the optimizer. For example, LMLI contains two ways of creating functions over values, `fix` and λ; B-form should contain only one way of creating functions over values. Third, the language should apply functions and primitive operations only to trivial expressions, such as constants or variables (an expression is trivial if it can be evaluated without risk of non-termination or side effects), because this simplifies many optimizations. For example, it makes inlining a function easy: simply apply the β_v rule of the call-by-value λ-calculus.

5.1.4 B-form

B-form is a conceptual subset of LMLI similar to A-Normal-Form [33]. Figure 5.1 shows the syntax for B-form constructors and types. B-form has the same kinds as LMLI and the same arity-0 and arity-1 primitive constructors as LMLI. The language of constructors is divided into three levels: constructor values, constructor expressions, and constructor declarations. Constructor values consist of constructor variables, the empty constructor list Nil, and arity-0 primitive constructors. Constructor expressions use only constructor values, except for branching constructor expressions. Those contain constructor functions or constructor declarations in their arms. Constructor declarations use Let to name program terms manipulated by the optimizer. There are three kinds of bindings: constructor expression bindings, constructor function bindings, and recursive constructor bindings. Types remain almost unchanged from LMLI: the only change is that the type expression $\mathsf{T}(-)$ must contain a constructor declaration. The "..." denotes the rest of the LMLI types.

Figure 5.2 shows the abstract syntax for term-level expressions and declarations of B-form. Like constructors, these are divided into three levels: *values*, *expressions*, and *declarations*. A B-form *program* is a declaration. Values consist of term-level variables, integers, floating point numbers, enumerated values, the special case of the empty record, and external values. An external value is the name of a value defined outside the B-form program, such as a function written in C. Expressions use only values or constructor values, except for branching expressions. Again, declarations name terms manipulated by the optimizer. They

(con. values)	cv	$::=$	$t :: \kappa \mid \mathsf{Nil} :: \kappa \mid c_0$
(constructors)	μ	$::=$	$cv \mid c_1(cv) \mid cv_1\, cv_2 \mid \mathsf{Typecase}\ cv\ \mathsf{of}\ a^*\ \mathsf{default}\ cd :: \kappa$ $\mathsf{Cons}(cv_1, cv_2) \mid \mathsf{Fold}\ cv\ \mathsf{of\ Nil} : cd\ \mathsf{Cons} : cf \mid$ $\mathsf{Listcase}\ cv\ \mathsf{of\ Nil}{:}cd\ \mathsf{Cons}{:}cf \mid$ $\mathsf{Tuple}(cv^*) \mid \mathsf{Proj}(i, cv)$
(con. bindings)	cb	$::=$	$t :: \kappa = \mu \mid t :: \kappa = cf \mid t :: \kappa = \mathsf{Rectype}(t_i, cd_i)^*.cd$
(con. decl.)	cd	$::=$	$cv \mid \mathsf{Let}\ cb\ \mathsf{in}\ cd$
(con. functions)	cf	$::=$	$\lambda t :: \kappa.cd$
(types)	τ	$::=$	$\mathsf{T}\ cd \mid \ldots$
(typecase arm)	a	$::=$	$c_0 : cd \mid c_1 : cf$

Figure 5.1: Abstract syntax for constructors of B-form

consist of expression bindings, constructor bindings, mutually recursive function bindings, mutually recursive type function bindings, exception raise, and exception handling. Note that all functions must be named in B-form.

I assume that all B-form programs satisfy the "value" restriction: the bodies of *all* polymorphic functions must be operationally equivalent to values. Wright [98] originally proposed the value restriction for SML programs as a simple way of remedying problems that arose in typing expressions which are polymorphic and use side-effects. Wright showed that nearly all SML programs satisfied the value restriction to start with, and those that did not could be trivially modified to satisfy the restriction.

It is reasonable for me to make the value restriction for B-form programs, because all compiler developers for SML plan to (or do) enforce the value restriction for SML programs and the translation of SML programs to B-form programs preserves the restriction. Second, all the optimizations presented in this chapter preserve the value restriction.

5.2 Notation

In this section, I discuss the notation that I use to describe algorithms and data structures. I write each algorithm as a sequence of data structure definitions followed by a sequence of mutually-recursive function definitions. I write the definitions in a pseudo-code language similar to SML. Evaluation of function parameters is call-by-value, and evaluation of expres-

(values)	v	$::=$	$x \mid i \mid r \mid \texttt{enum}(i) : \sigma \mid \texttt{Unit} \mid \texttt{extern}(s, \sigma) \mid cp(v) : \sigma$
(expressions)	e	$::=$	$v \mid s \mid p_1(v) \mid p_2(v_1, v_2) \mid \texttt{update}(ar, v_1, v_2, v_3) \mid$ $\texttt{new\_exn}\ cv \mid \texttt{make\_vararg}(cv, v) \mid \texttt{call\_vararg}(cv_1, v_1, v_2) \mid$ $\texttt{eq}\ cv \mid \texttt{neq}\ cv \mid \texttt{record}(v^*) \mid \texttt{inject}(i, v^*) \mid$ $\texttt{switch}(st)\ v\ \texttt{of}\ (i : f)^*\ \texttt{default}\ : od \mid$ $\texttt{typecase}\ cv\,\texttt{of}\,[t.\sigma]\, ta^*\,\texttt{rectype}\ : otf\ \ \texttt{default}\ : od \mid$ $\texttt{tlistcase}\ cv\,\texttt{of}\,[t.\sigma]\,\texttt{nil} : d\,\texttt{cons} : tf \mid$ $v(v^*) \mid v[v^*]$
(declarations)	d	$::=$	$v : \sigma \mid \texttt{let}\ x : \sigma = e\ \texttt{in}\ d \mid \texttt{let}\ cb\ \texttt{in}\ d \mid$ $\texttt{let fix}\ (x : \sigma\texttt{=}f)^*\ \texttt{in}\ d \mid \texttt{let fixtype}\ (x : \sigma = tfn)^*\ \texttt{in}d$ $\texttt{raise}\ (v, \sigma) \mid \texttt{let}\ x : \sigma = d_1\ \textsf{handle}\ f\,\textsf{in}\ d_2$
(functions)	f	$::=$	$\lambda(x{:}\sigma)^*.decl$
(type fun.)	tf	$::=$	$\Lambda(t :: \kappa)^*.decl$
(typecase arm)	ta	$::=$	$c_0 : d \mid c_1 : tf$
(opt. decl)	od	$::=$	$d \mid \epsilon$
(opt type fun.)	otf	$::=$	$tf \mid \epsilon$

Figure 5.2: Abstract syntax for term-level expressions and declarations of B-form

sions proceeds from left to right. I write case expressions as

case x of
$c_1 : e_1$
...
$c_n : e_n$

A case expression analyzes x to see if it has the form c_1, c_2 or so on. If it finds that x matches c_i, then it evaluates e_i. I write "if" expressions as

if e
then e_t
else e_f

An if expression evaluates e. If e is true, then it evaluates e_t. Otherwise it evaluates e_f. I write let declarations as

let $x = e_1$
in e_2
end

A let declaration evaluates e_1, binds x to the result, and then evaluates e_2. The expression $(e_1; \ldots; e_n)$ evaluates expressions sequentially: first it evaluates e_1, then it evaluates e_2, and so on up to e_n.

Occasionally I overload notation, and apply a function f to an optional B-form declaration when I have defined f only for declarations. The overloaded notation means apply f if the optional declaration is present. Similarly, sometimes I apply a function f to optional B-form functions.

I use *finite maps* as the primary data structures for most algorithms. I write a finite map mapping x_1 to y_1, x_2 to y_2 and so on as $\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots\}$. I require that all the x_i be distinct. I write the empty map as $\emptyset$. I write a map extended by an entry as $M[x \mapsto y]$. I write a map M_1 extended by the entries in map M_2 as $M_1 + M_2$. I assume that the domains of M_1 and M_2 are disjoint.

To distinguish program terms manipulated by algorithms from the definitions of the algorithms, I enclose program terms manipulated by algorithms in Quine brackets, following notation used in denotational semantics. For example, I write a B-form `let` expression being used by an algorithm as

$$\ulcorner \texttt{let } x = e \texttt{ in } x \urcorner$$

Program terms may contain meta-variables, which range over other pieces of syntax and are variables used by the algorithms. I write meta-variables in an italic font, such as *this*. In the previous example, x and e are meta-variables that range over B-form variables and expressions respectively. I write actual pieces of syntax in a teletype font, such as `this`. For example,

$$\ulcorner \texttt{let y=a+b in y} \urcorner$$

is a B-form `let`-declaration binding the B-form variable `y` to the B-form expression `a+b`.

I use SML notation for reference cells to indicate updateable state. The `ref` operator allocates and initializes a store location: $A = \texttt{ref } \emptyset$ binds the variable A to a store location which contains the empty map. The ! operator reads a store location: $!A$ is the current value of the map stored in A. The := operator writes a store location; for example $A := \{x \mapsto y\}$ updates A to contain the singleton map mapping x to y.

5.3 Assumptions

In this section, I discuss some assumptions that I have made throughout the optimizer. Throughout the description of the optimizer, I assume that bound variables have been renamed so that each bound variable has a unique name. Assuming that each bound variable has a unique name avoids artificial constraints on optimization due to scoping, with information having to be deleted because a variable has been rebound. For example, consider a program of the form:

```
val a = b + e
val a = b
val d = b + e
```

After `a` is bound to `b+e`, the value of the expression `b+e` becomes *available*. However, when `a` is then rebound to `b`, the value of `b+e` becomes unavailable, *i.e.* , `d` cannot be bound to `a`. Renaming variables to be unique means that the value of `b+e` is available when `d` is bound, so `d` can be assigned to that value:

```
val a = b+e
val a' = b
val d = a
```

Renaming variables also makes it easy to keep program-wide maps from variables to program properties. For example, inlining keeps a count of the number of times a function has been applied so that it can identify functions that are applied only once. If two functions could have the same name, then keeping a program-wide map of counts would lead to an incorrect count of the number of times those functions have been applied.

The first phase of the optimizer renames bound variables so that the invariant that all bound variables are unique becomes true. Subsequent phases of the optimizer preserve the invariant by renaming variables when necessary.

5.3.1 Effects analysis

In this section, I discuss a simple effects analysis for B-form programs. I first motivate the problem of effects analysis and discuss my approach to it. Next, I present an algorithm that formalizes the effects analysis. After that, I present a memoized version of the algorithm that computes the effects for all expressions. Because the analysis is simple, it can be computed efficiently — the memoized version runs in $O(N)$ time, where N is the size of the program. Finally, I discuss related work.

An expression has an *effect* if it reads, updates, or allocates updateable data, such as arrays, if it raises an exception, or if it does not terminate. An expression is *pure* if it is equivalent to a value and does not have an effect. In general, moving expressions with effects around in a program changes the meaning of the program. Moving pure expressions around, on the other hand, does not change the meaning of the program.

The problem of determining when expressions read or write the store has been studied extensively for conventional languages such as C and Fortran. This is because in C and FORTRAN, variables are updateable, and hence an expression that uses a variable implicitly dereferences the store. Similarly, an expression that assigns to a variable implicitly updates the store. This implicit use of the store makes careful tracking of side-effects to the store a central component of conventional data flow analysis [3]. Indeed, it has led to the development of the static single assignment (SSA) intermediate representation, where variables can be assigned to only once. The fact that variables can be assigned to only once simplifes many analyses.

In SML, however, variables are *not* updateable. Variables may only be bound. In this aspect, SML differs from Scheme and Lisp, where variables are updateable just like C and FORTRAN. If you are unfamiliar with the distinction between binding versus assignment of variables, you should consult references on SML [67, 91] for a further explanation of this point.

This means that tracking side-effects to the store is less important than it is for more conventional languages, because side-effects occur less frequently. Thus, I use a simple analysis for determining whether an expression depends on the store: I note whether an expression reads the store, writes the store, or allocates a new location in the store. My hope is that even a simple side-effects analysis will lead to significant improvements in practice. I shall demonstrate in Chapter 9 that this is true. In Chapter 11, Future Work, I discuss more sophisticated approaches to side-effects analysis.

SML requires that arithmetic operations check for overflow, unlike C and FORTRAN and that exceptions be raised in the correct (left-to-right) order. This means that the optimizer must be careful when moving arithmetic expressions around. Again, I rely on a simple analysis: I note whether an expression may raise an exception.

Finally, sometimes the optimizer can move only terminating expressions. In particular, invariant removal cannot move a function application that has no side-effects and does not raise exceptions because that could change the termination behavior of the program. Again, I use a simple analysis and note whether an expression may not terminate.

An algorithm for effects analysis

In this section, I present an algorithm for computing the effects of B-form expressions and declarations. The algorithm computes the set of basic effects that each expression or declaration may have. The basic effects are read the store (R), write the store (W), allocate a new location in the store (A), raise an exception (E), or non-termination (N). The algorithm uses a lattice whose set of elements is $\mathcal{P}(\{R, W, A, E, N\})$. Lattice elements are ordered by the subset relation $\subseteq$. The least upper bound $\sqcup$ of two lattice elements is computed using set union $\cup$.

I define Algorithm 1 in Figure 5.3 using two mutually recursive functions, $\mathrm{eff}_{\mathrm{d}}$ and $\mathrm{eff}_{\mathrm{e}}$. The function $\mathrm{eff}_{\mathrm{d}}$ computes the effect of a declaration, while the function $\mathrm{eff}_{\mathrm{e}}$ computes the effect of an expression. A declaration that is simply a value has no effect. The effect of a `let`-declaration that binds an expression is the least upper bound of the effect of the expression and the body of the declaration. The effect of a `let`-declaration that binds a constructor variable is the effect of the body of the declaration, because constructors have no effects. Similarly, the effect of a `fix`-declaration is the effect of its body, because binding variables to functions has no effect. A `raise` declaration has the effect {E}. Finally, the effect of a `handle` declaration is the least upper bound of the effects of the declaration wrapped by the handler, the handler, and the body of the declaration.

Expressions involving primitive operators have no effect, can just raise on exception, or can read or allocate from the store. The set pure-ops defines the pure primitive operators. The set exn-ops defines the primitive operators that can just raise exceptions. The update expression updates the store. The new_exn expression has an allocate effect, because it creates a new exception that can be distinguished from other exceptions. The effects of the branching expressions `switch`, `typecase`, and `tlistcase` are simply the least upper bounds of the effects of the branches.

I do not analyze the effect of term-level function applications, so I make the worst-case assumption that a term-level function application may have any effect. Because B-form programs satisfy the value restriction (see Section 5.1.4), however, polymorphic function applications have no effect. The remaining kinds of expressions (record creation, sum creation, functions to deal with variables numbers of arguments, and polymorphic equality and inequality), are pure and have no effect.

Algorithm 1 has an $O(N)$ running time, where N is program size. It visits each syntax node for the program at most once, and does a constant amount of work for each node.

An asymptotically efficient version

Many optimizations need to analyze the effect of each expression. For these optimizations, applying Algorithm 1 leads to an $O(N^2)$ running time, where N is program size. You can avoid this behavior by memoizing Algorithm 1 and computing the effects in one pass over the program.

Figure 5.4 presents the changes needed to memoize Algorithm 1. The memoized algorithm computes a mapping M from variables to the effects of the expressions or declarations bound to those variables. The memoized algorithm has an $O(N)$ running time.

Related work

Many researchers have studied effects analysis for functional languages, but have concentrated on analyzing the effects of higher-order functions. Their analyses produce detailed effects information but at considerable cost in terms of compile-time. In contrast, I have ignored the problem of higher-order functions by assuming that a term-level function application may have any effect, but my analysis is asymptotically efficient. As I demonstrate in Chapter 10, it is quite reasonable to ignore higher-order functions in practice.

Neirynck and others [64, 65] present an analysis based on abstract interpretation for determining whether an expression uses or updates the store in a simply-typed λ-calculus. However, she is concerned primarily with demonstrating soundness of the analyses, and does

$$
\begin{array}{lcl}
\mathrm{eff}_{\mathrm{d}}(d) & = & \text{case } d \text{ of} \\
& & \lceil v:\sigma \rceil : \emptyset \\
& & \lceil \texttt{let } x:\sigma = e \texttt{ in } d' \rceil \;:\mathrm{eff}_{\mathrm{e}}(e) \sqcup \mathrm{eff}_{\mathrm{d}}(d') \\
& & \lceil \texttt{let } cb \texttt{ in } d' \rceil \;:\mathrm{eff}_{\mathrm{d}}(d') \\
& & \lceil \texttt{let fix } x_1:\sigma_1 = f_1 \ldots \texttt{ and } x_n:\sigma_n = f_n \texttt{ in } d' \rceil \;:\mathrm{eff}_{\mathrm{d}}(d') \\
& & \lceil \texttt{let fixtype } x_1:\sigma_1 = tf_1 \ldots \texttt{ and } x_n:\sigma_n = tf_n \texttt{ in } d' \rceil \;:\mathrm{eff}_{\mathrm{d}}(d') \\
& & \lceil \texttt{raise } (v,\sigma) \rceil \;:\{E\} \\
& & \lceil \texttt{let } x:\sigma = d_1 \texttt{ handle } \lambda v.d_2 \texttt{ in } d_3 \rceil \;:\mathrm{eff}_{\mathrm{d}}(d_1) \sqcup \mathrm{eff}_{\mathrm{d}}(d_2) \sqcup \mathrm{eff}_{\mathrm{d}}(d_3)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{eff}_{\mathrm{e}}(e) & = & \text{case } e \text{ of} \\
& & \lceil p_1(v) \rceil : \begin{cases} \emptyset : \text{if } p_1 \in \text{pure-ops} \\ \{E\} : \text{if } p_1 \in \text{exn-ops} \end{cases} \\
& & \lceil p_2(v_1, v_2) \rceil : \\
& & \quad \texttt{case } p_2 \texttt{ of} \\
& & \quad \lceil \texttt{sub } ar \rceil : \{R\} \\
& & \quad \lceil \texttt{alloc } ar \rceil : \{A\} \\
& & \quad \text{otherwise} : \begin{cases} \emptyset : \text{if } p_2 \in \text{pure-ops} \\ \{E\} : \text{if } p_2 \in \text{exn-ops} \end{cases} \\
& & \lceil \texttt{update}(ar, v_1, v_2, v_3) \rceil : \{W\} \\
& & \lceil \texttt{new\_exn}(cv) \rceil : \{A\} \\
& & \lceil \texttt{switch } (st)\, v \texttt{ of } (i:\lambda x.d_i)^* \texttt{ default: } od \rceil \;:(\sqcup \mathrm{eff}_{\mathrm{d}}(d_i)) \sqcup \mathrm{eff}_{\mathrm{d}}(od) \\
& & \lceil \texttt{typecase } st \texttt{ of } [cv]t.\sigma \texttt{ rectype } :(i:ta_i)^* \texttt{ default} : otf \rceil \; od : \\
& & \qquad (\sqcup \mathrm{eff}_{\mathrm{ta}}(ta_i)) \sqcup \mathrm{eff}_{\mathrm{tf}}(otf) \sqcup \mathrm{eff}_{\mathrm{d}}(od) \\
& & \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } :d_1 \texttt{ cons} : \Lambda t.d_2 \rceil \;:\mathrm{eff}_{\mathrm{d}}(d_1) \sqcup \mathrm{eff}_{\mathrm{d}}(d_2) \\
& & \lceil \texttt{v}(v_1 \ldots v_n) \rceil : \{R, W, A, E, N\} \\
& & \lceil \texttt{v}[cv_1 \ldots cv_n] \rceil : \emptyset \\
& & \text{otherwise} : \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{eff}_{\mathrm{ta}}(a) & = & \text{case } a \text{ of} \\
& & \lceil c_0 : d \rceil : \mathrm{eff}_{\mathrm{d}}(d) \\
& & \lceil c_1 : \lambda x.d \rceil : \mathrm{eff}_{\mathrm{d}}(d)
\end{array}
$$

$$
\begin{array}{lcl}
\text{pure-ops} & = & \{\texttt{real}, \texttt{not}, \texttt{size}, \texttt{select i}, \texttt{eqi}, \texttt{lti}, \texttt{gti}, \texttt{ltei}, \texttt{gtei}, \texttt{divui}, \texttt{mului}, \\
& & \texttt{plusui}, \texttt{minusui}, \texttt{ltui}, \texttt{gtui}, \texttt{lteui}, \texttt{gteui}, \texttt{eqr}, \texttt{ltr}, \texttt{gtr}, \texttt{lter}, \texttt{gter}, \\
& & \texttt{excon}, \texttt{de\_excon}, \texttt{eqptr}\}
\end{array}
$$

$$
\begin{array}{lcl}
\text{exn-ops} & = & \{\texttt{floor}, \texttt{sqrt}, \texttt{sin}, \texttt{cos}, \texttt{arctan}, \texttt{exp}, \texttt{ln}, \texttt{divi}, \texttt{muli}, \texttt{plusi}, \texttt{minusi}, \texttt{modi}, \\
& & \texttt{divr}, \texttt{mulr}, \texttt{plusr}, \texttt{minusr}\}
\end{array}
$$

Figure 5.3: Algorithm 1: compute effects of B-form declarations and expressions

$$
\begin{array}{lcl}
M & = & \texttt{ref}\ \emptyset \\
\\
\mathrm{eff}_{\mathrm{d}}(d) & = & \text{case } d \text{ of} \\
 & & \lceil v:\sigma \rceil : \emptyset \\
 & & \lceil \texttt{let}\ x:\sigma = e\ \texttt{in}\ d' \rceil\ : \\
 & & \quad \text{let } y = \mathrm{eff}_{\mathrm{e}}(e) \\
 & & \quad \text{in } M := !M + \{x \to y\}; y \sqcup \mathrm{eff}_{\mathrm{d}}(d') \\
 & & \quad \text{end} \\
 & & \lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil\ : \mathrm{eff}_{\mathrm{d}}(d') \\
 & & \lceil \texttt{let fix}\ x_1:\sigma_1 = f_1 \ldots\ \texttt{and}\ x_n:\sigma_n = f_n\ \texttt{in}\ d' \rceil\ : \\
 & & \quad (\forall 1 \leq i \leq n. \mathrm{eff}_{\mathrm{f}}(f_i); \mathrm{eff}_{\mathrm{d}}(d')) \\
 & & \lceil \texttt{let fixtype}\ x_1:\sigma_1 = tf_1 \ldots\ \texttt{and}\ x_n:\sigma_n = tf_n\ \texttt{in}\ d' \rceil\ : \\
 & & \quad (\forall 1 \leq i \leq n. \mathrm{eff}_{\mathrm{tf}}(tf_i); \mathrm{eff}_{\mathrm{d}}(d')) \\
 & & \lceil \texttt{raise}\ (v,\sigma) \rceil\ : \{E\} \\
 & & \lceil \texttt{let}\ x:\sigma = d_1\ \texttt{handle}\ \lambda v.d_2\ \texttt{in}\ d_3 \rceil\ : \\
 & & \quad \text{let } y = \mathrm{eff}_{\mathrm{d}}(d_1) \text{ and } z = \mathrm{eff}_{\mathrm{d}}(d_2) \\
 & & \quad \text{in } M := M + \{x \to y \sqcup z\}; y \sqcup z \sqcup \mathrm{eff}_{\mathrm{d}}(d_3) \\
 & & \quad \text{end} \\
\\
\mathrm{eff}_{\mathrm{f}}(f) & = & \text{case } f \text{ of} \\
 & & \lceil \lambda(x_1 \ldots x_n).d \rceil : \mathrm{eff}_{\mathrm{d}}(d) \\
\\
\mathrm{eff}_{\mathrm{tf}}(tf) & = & \text{case } tf \text{ of} \\
 & & \lceil \Lambda(t_1 \ldots t_n).d \rceil : \mathrm{eff}_{\mathrm{d}}(d)
\end{array}
$$

Figure 5.4: Changes needed to memoize Algorithm 1

not describe the asymptotic complexity of the analysis when it is applied to a language with all the features of ML. The complexity of her analysis appears to be at least exponential.

Lucassen and Gifford [59, 34] describe a type-and-effect system where expressions are explicitly annotated with types and descriptions of effects. Jouvelot and Gifford [46] show how to reconstruct types-and-effects in a language with explicit polymorphism, where quantification occurs over types and effects. Talpin and Jouvelot [88] show how to infer types-and-effects in a manner similar to Hindley-Milner type reconstruction for ML. Leroy and Weis [55] study another extension to the ML type system.

5.4 Inlining

Inlining selectively replaces function calls with copies of the bodies of the called functions. It is an important optimization for eliminating higher-order functions. In addition, it is also important for eliminating small functions, where the cost of a function call and return may be as large as the cost of evaluating the body of a function.

For example, consider applying inlining to the following program:

```
fun map f =
  let fun loop nil = nil
        | loop (h::t) = f h :: loop t
  in loop
  end
val inc = λx.x+1
val mapinc = map inc
val a = mapinc [1,2,3,4]
```

Inlining the function `map` produces the following program:

```
val inc = λx.x+1
val mapinc = let val f = inc
                 fun loop nil = nil
                   | loop (h::t) = f h :: loop t
             in loop
             end
val a = mapinc [1,2,3,4]
```

Inlining the function `inc` produces

```
val mapinc = let fun loop nil = nil
                   | loop (h::t) = h+1 :: loop t
             in loop
             end
val a = mapinc [1,2,3,4]
```

The final program should run faster than the original program, because a call to an unknown function has been replaced by an addition operation.

This example illustrates how inlining often improves programs in the presence of higher-order functions. Inlining first changes a call to an unknown function to a call to a known function. Then it eliminates the call to the known function, if the known function is small enough.

5.4.1 Algorithms for inlining

The decision to inline a function at a particular call must balance program speed against program size. Inlining the function usually makes a program faster, but it also makes the program bigger. In fact, repeatedly inlining a recursive function within its body could make a program "infinitely" bigger, and cause the optimizer to not terminate.

Given that inlining is a trade-off between program speed and program size, and that inlining recursive functions within their own bodies is dangerous, I use the following heuristics for deciding when to inline a function:

- If a function is called only once, inline it.
- If the size of a function is less than some constant c, and the function is not recursive, then expand all the calls to the function.

The first heuristic does not change the size of a program, so it is always safe. Repeated application of the second heuristic, however, is dangerous. Although each application of the heuristic increases the size of the program by only a constant factor, repeated application of the heuristic could produce a program which is exponentially larger than the original program. To avoid this, the second heuristic should applied only a small number of times.

The following two algorithms for inlining are refinements of these heuristics. The first algorithm, given below, inlines functions called only once.

Algorithm 2 *Inlining functions called only once*

- Phase 1 (analysis): identify functions to inline.

  Keep an updateable map M from variables naming functions bound by `fix` or `fixtypes` to the following counts: the number of recursive applications, the number of non-recursive applications, and the number of other uses.

  Traverse the program from top to bottom. On encountering a `fix` or `fixtype`, initialize the counts for all the functions bound by the `fix` or `fixtype` to 0.

  At any recursive application of a variable naming a function, increment the number of recursive applications by 1. A variable naming a function is recursively applied if it is applied within the body of the function it names *or* it is applied within the body of any other function defined by the same `fix` or `fixtype`.

  At any other application of a variable naming a function, increment the number of non-recursive applications by 1.

  At any other occurrence of variable naming a function, increment the number of other uses by 1.

After traversing the program, let the set I be those variables in the domain of M such that the count of recursive applications is 0, the count of non-recursive applications is 1, and the number of other uses is 0.

- Phase 2 (transformation): inline those functions.

  If I is empty, then return the original program.

  Otherwise, let S be a map from function names to functions or polymorphic functions. (expressions in syntactic classes f and tf).

  Traverse the program from top to bottom. At each `fix` or `fixtype` do the following:

  1. Rewrite the functions defined by the `fix` or `fixtype`, so that any functions applied only once within their bodies are inlined.
  2. Add the rewritten definitions of any functions in I to S. Place only the rewritten definitions of functions not in I in the `fix` or `fixtype`, and continue rewriting the body of the `fix` or `fixtype`.

  At any application or type application of a function $f \in \text{dom}(S)$, replace the application of f with the body of $S(f)$. Do this as follows. Let the application have the form

  ```
  let x  = f(x_1, ..., x_n)
  in d
  ```

  First, rewrite d to d' so that all functions applied only once in d are inlined.

  Now, place the body of the function inline by calling the function `expand` (defined below) with $S(f)$, the argument list $x_1, \ldots, x_n$, and d'.

Function 1 *expand the body of a function*

Expand first binds the formal parameters to the actual parameters. It then calls the function flatten to turn the declaration `let x =` d `in` d' into a valid B-form declaration The let-declaration is not a valid B-form declaration because d is a nested declaration.

```
expand(⌈λ a_1 ... a_n.d⌉, x_1 ... x_n, d') =
 ⌈let a_1 = x_1
      ...
      a_n = x_n
  in flatten(x,d,d')
  end ⌉
```

Function 2 *flatten a nested declaration.*

Create a valid B-form declaration equivalent to the LMLI declaration $\texttt{let}\ x = d_1\ \texttt{in}\ d_2$.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{flatten}(x, d_1, d_2) =} \\
\quad \text{let append}(d) & = & \text{case } d \text{ of} \\
 & & \lceil v : \sigma \rceil : \lceil \texttt{let}\ x = v\ \texttt{in}\ d_2 \rceil \\
 & & \lceil \texttt{let}\ v : \sigma = e\ \texttt{in}\ d' \rceil\ : \lceil \texttt{let}\ v : \sigma = e\ \texttt{in}\ \text{append}(d') \rceil \\
 & & \lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil\ : \lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil \\
 & & \lceil \texttt{let fix}\ fl\ \texttt{in}\ d' \rceil : \lceil \texttt{let fix}\ fl\ \texttt{in}\ \text{append}(d')\ \texttt{end} \rceil \\
 & & \lceil \texttt{let fixtype}\ fl\ \texttt{in}\ d' \rceil : \lceil \texttt{let fixtype}\ fl\ \texttt{in}\ \text{append}(d')\ \texttt{end} \rceil \\
 & & \lceil \texttt{raise}\ (v, \sigma) \rceil\ : \lceil \texttt{raise}\ (v, \text{type-of}(d_2)) \rceil \\
 & & \lceil \texttt{let}\ v : \sigma = d'\ \texttt{handle}\ f\ \texttt{in}\ d'' \rceil\ : \\
 & & \quad \lceil \texttt{let}\ v : \sigma = d'\ \texttt{handle}\ f\ \texttt{in}\ \text{append}(d'') \rceil \\
\quad \text{in append}(d_1) & & \\
\quad \text{end} & &
\end{array}
$$

Note that variables need to be renamed in the bodies of expanded functions to preserve the invariant that all bound variables names are unique. This renaming can be avoided for a function that is applied only once, provided that the definition of the function is removed when the function body is expanded.

Algorithm 3 *Inlining functions called multiple times using original copies of those functions.*

This algorithm inlines non-recursive functions whose sizes are less than some constant c.

- Phase 1 (analysis): identify functions to inline

  First, identify functions that are applied recursively. Repeat the analysis from phase 1 of Algorithm 2.

  Second, calculate the sizes of functions. Figure 5.5 shows mutually-recursive functions that measure the sizes of the syntax trees of functions, polymorphic functions, expressions, and declarations. The functions to calculate sizes of constructors are similar.

  Sizes of functions should not be calculated by traversing the program syntax tree and applying the size_f function to every function in the program. This has an $O(N^2)$ running time, because it recomputes the sizes of nested functions. Instead, sizes should be calculated in one bottom-up pass over the syntax tree. The size_d function should be modified to record the sizes of functions as it calculates them.

  Let the set I be those variables naming functions whose sizes are less than the constant c and whose count of recursive applications is 0.

- Phase 2 (transformation): inline those functions.

  If I is empty, then return the original program.

  Otherwise, let S be an updateable map from function names to functions or polymorphic functions. Traverse the program from top to bottom. At each `fix` or `fixtype`,

$$
\begin{aligned}
\mathrm{size}_{\mathrm{f}}(f) &= \text{case } f \text{ of} \\
&\quad \lceil \lambda(x_1 \ldots x_n).d \rceil : 1 + \mathrm{size}_{\mathrm{d}}(d) \\[1em]
\mathrm{size}_{\mathrm{tf}}(tf) &= \text{case } tf \text{ of} \\
&\quad \lceil \Lambda(t_1 \ldots t_n).d \rceil : 1 + \mathrm{size}_{\mathrm{d}}(d) \\[1em]
\mathrm{size}_{\mathrm{d}}(d) &= \text{case } d \text{ of} \\
&\quad \lceil v : \sigma \rceil : 1 \\
&\quad \lceil \texttt{let } x : \sigma = e \texttt{ in } d' \rceil \ : 1 + \mathrm{size}_{\mathrm{e}}(e) + \mathrm{size}_{\mathrm{d}}(d') \\
&\quad \lceil \texttt{let } cb \texttt{ in } d' \rceil \ : 1 + \mathrm{size}_{\mathrm{cb}}(cb) + \mathrm{size}_{\mathrm{d}}(d') \\
&\quad \lceil \texttt{let fix } x_1 : \sigma_1 = f_1 \ldots \texttt{ and } x_n : \sigma_n = f_n \texttt{ in } d' \rceil \ : \\
&\qquad 1 + \textstyle\sum_{i=1}^{n} \mathrm{size}_{\mathrm{f}}(f_i) + \mathrm{size}_{\mathrm{d}}(d') \\
&\quad \lceil \texttt{let fixtype } x_1 : \sigma_1 = tf_1 \ldots \texttt{ and } x_n : \sigma_n = tf_n \texttt{ in } d' \rceil \\
&\qquad 1 + \textstyle\sum_{i=1}^{n} \mathrm{size}_{\mathrm{tf}}(tf_i) + \mathrm{size}_{\mathrm{d}}(d') \\
&\quad \lceil \texttt{raise } (v, \sigma) \rceil \ : 1 \\
&\quad \lceil \texttt{let } x : \sigma = d_1 \texttt{ handle } f \texttt{ in } d_2 \rceil \ : 1 + \mathrm{size}_{\mathrm{d}}(d_1) + \mathrm{size}_{\mathrm{f}}(f) + \mathrm{size}_{\mathrm{d}}(d_2) \\[1em]
\mathrm{size}_{\mathrm{e}}(e) &= \text{case } e \text{ of} \\
&\quad \lceil \texttt{switch } (st)\, v \texttt{ of } (i : f_i)^* \texttt{ default: } od \rceil \ : \\
&\qquad 1 + \textstyle\sum \mathrm{size}_{\mathrm{f}}(f_i) + \mathrm{size}_{\mathrm{d}}(od) \\
&\quad \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma](i : ta_i)^* \texttt{ rectype } : otf \texttt{ default} : od \rceil \ : \\
&\qquad 1 + \textstyle\sum \mathrm{size}_{\mathrm{ta}}(ta_i) + \mathrm{size}_{\mathrm{tf}}(otf) + \mathrm{size}_{\mathrm{d}}(od) \\
&\quad \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : d \texttt{ cons} : tf \rceil \ : \\
&\qquad 1 + \mathrm{size}_{\mathrm{d}}(d) + \mathrm{size}_{\mathrm{tf}}(tf) \\
&\quad \lceil \texttt{record}(v_1 \ldots v_n) \rceil : n \\
&\quad \lceil \texttt{inject}(i, v_1 \ldots v_n) \rceil : n \\
&\quad \lceil \texttt{v}(v_1 \ldots v_n) \rceil : n \\
&\quad \lceil \texttt{v}[cv_1 \ldots cv_n) \rceil : n \\
&\quad \text{otherwise} : 1 \\[1em]
\mathrm{size}_{\mathrm{ta}}(a) &= \text{case } a \text{ of} \\
&\quad \lceil c_0 : d \rceil : \mathrm{size}_{\mathrm{d}}(d) \\
&\quad \lceil c_1 : tf \rceil : \mathrm{size}_{\mathrm{tf}}(tf)
\end{aligned}
$$

Figure 5.5: Functions to calculate the sizes of functions, type functions, expressions, and declarations

record in the map S any functions defined there that are in I. Then rewrite the functions defined by the `fix` or `fixtype` and the declaration that is the body of the `fix` or `fixtype`.

At any application or type application of a function $f \in \text{dom}(S)$, replace the application with the body of $S(f)$ using the same procedure used by Algorithm 2. Note that $S(f)$ is the original copy of f, not the rewritten version of f.

A subtlety of Algorithm 3 is that it uses copies of functions before inlining has been applied to their bodies. This can cause problems when Algorithm 3 is applied repeatedly to a program; functions in inner loops may not be inlined. Consider the following program:

```
fix inc = λx.x+1
fix f = λx.
    let fix g = λy.
              let fix loop = λi.
                     ... loop (inc i)
              in loop y
              end
    in g (x+1)
    end
val a = f 5
val b = a 2
```

One application of Algorithm 3 produces the following program:

```
fix inc = λx.x+1
fix f = λx.
    let fix g = λy.
              let fix loop = λi.
                     ... loop (i+1) ...
              in loop y
              end
    in g (x+1)
    end
val a =
    let fix g′ = λy′.
              let fix loop′ = λi′.
                     ... loop′(inc i′) ...
              in loop′ y′
              end
    in g′ (5+1)
    end
val b = a 2
```

The algorithm inlines the application of `inc` in `f`, but it also creates a new application of `inc` when it inlines the original body of `f`. If you apply the algorithm a second time, the algorithm does not inline `inc` in an inner loop. Only when you apply the algorithm a *third* time does it inline `inc`.

Fixing this problem leads to a third algorithm for inlining, which is based on Algorithm 3. This algorithm is the one that TIL actually uses. In this variant, most of the analysis phase is combined with the transformation phase:

Algorithm 4 *Inlining functions called multiple times using new copies of those functions.*

- Phase 1 (analysis): use the analysis from phase 1 of Algorithm 2 to identify recursive functions.

- Phase 2 (analysis and transformation):

  Let S be an updateable map from function names to functions or constructor functions. Rewrite the program from top to bottom. At each `fix`, `fixtype`, or constructor function binding, do the following:

  1. Rewrite the functions defined there so that inlining is applied to their bodies.
  2. Calculate the sizes of the rewritten functions.
  3. Add those rewritten definitions whose sizes are less than c and that are not applied recursively to S.
  4. Place the rewritten definitions of functions in the `fix` or `fixtype`, and rewrite the body of the `fix` or `fixtype`.

  At any application of a function $f \in \mathrm{dom}(S)$, replace the application using the same procedure used by Algorithm 2.

5.4.2 Related work

My basic inlining strategy:

- inline a function if it is called only once

- or inline a function if it is suitably small and non-recursive

is a simpler and more conservative version of Appel's inlining strategy [7].

Appel's strategy inlines recursive functions, but my strategy does not (directly) inline recursive functions. From examining code produced by the SML/NJ compiler, I believe that inlining recursive functions using a simple local syntactic criteria often leads to code expansion without any significant improvement in execution time.

Appel's strategy also attempts to estimate the decrease in code size that may occur when an inlined function is optimized. I believe that such an estimate is likely to be inaccurate and thus not useful, given all the optimizations done by TIL and the way that they interact.

Even though my strategy is a simpler version of Appel's strategy, Chapter 10.1 shows that my strategy produces excellent results when compiling whole programs. Almost no higher-order or polymorphic functions remain after optimization. My inlining strategy is also fairly insensitive to the definition of what a "suitably small" function is. Code size varies only by a factor of two, even when this definition varies over a wide range.

5.5 Uncurrying

A common programming idiom in SML is to write a function in a "curried" style, where each of the arguments is passed in by a function call:

```
val f = λx.λ.y.λz.x+y+z
val t= ((f 1) 2) 3
```

This allows more flexibility in programming: the function may be partially applied to its first n arguments. For example, if one is writing an interpreter for a programming language, one may have a function `eval`:

```
val eval = λe.λE.  ...
```

which takes an environment `e`, and evaluates an expression `E` in the environment. When interpreting a function call `e_0(e_1, ... e_n)`, in some environment, eval could be partially applied to an environment, and then applied to each expression:

```
case exp
of APP(e_0,args) =>
        let val eval' = eval e
            val e_0' = eval' e_0
            val args' = map eval' args
```

Although currying allows more flexibility in programming, it can be inefficient when implemented straightforwardly. Passing each argument except the last one results in a function call, construction of a closure, and a function return. Also, currying introduces first-class functions, which means that the control-flow graph is no longer statically apparent [83].

For these two reasons, an important optimization for SML programs is *uncurrying*: taking a curried function and transforming it to a multi-argument function where all arguments are passed at once. For example, the uncurried version of `f` is

```
val f = λx,y,z.x+y+z
```

5.5.1 An algorithm for uncurrying

The algorithm for uncurrying is divided into four phases:

Algorithm 5 *Uncurrying*

- Phase 1 (analysis): identify the functions to be uncurried.

  Traverse the program in a top-down manner. When a function of the form

```
let fix f₁ = λx₁.let fix f₂ = λx₂. ...
                 in f₂
                 end
in f₁
end
```

is found, scan down the sequence of nested function definitions (following the "...") until a function $\mathtt{f}_n$ is encountered

1. whose body b does not fit the form:

```
let fix g = λy. ...
in g
end
```

2. or where $\mathtt{f}_n$ occurs free in b (that is, $\mathtt{f}_n$ is a recursive function).

Given the sequence of nested functions

```
let fix f₁ = λx₁.let fix f₂ =
                           λx₂. ... let fix fₙ=λxₙ.e
                                       in fₙ
                                       end
                  in f₂
                  end
in f₁
```

record $\mathtt{f}_1$ as a curried function of arity n which is to be uncurried. Do not mark any of the intermediate functions $\mathtt{f}_2$ through $\mathtt{f}_n$ as candidates to be uncurried. Instead, continue traversing e and b, looking for more functions to uncurry.

- Phase 2 (analysis): identify applications of those curried functions.

  The key idea here is the notion of a *partial application* of a curried function `f`. Because of the nature of B-form, the algorithm cannot just look for a sequence of nested applications:

  ```
  (((f e₁) e₂) ... eₙ)
  ```

  In B-form, each application is named and the sub-computations of $\mathtt{e}_1$ and so on are interspersed with the applications. For example, SML code of the form:

  ```
  val g = λx.λy.λz.x+y+z
  val a = (g (s+t) (t+u) (t+t))
  ```

  is represented in B-form as

```
fix g = λx.
          let fix h=λy.
                 let fix j = λz.
                        let val t1 = x+y
                            val t2 = y+z
                        in t2
                        end
                 in j
                 end
         in h
         end
val t3 = s+t
val g₁ = g t3
val t4 = t+u
val g₂ = g₁ t4
val t5 = t+t
val a = g₂ t5
```

The applications occur at g_1, g_2 and a. The computations of the arguments `s+t`, `t+u`, and `t+t` are interspersed between the applications.

A curried function `g` is *partially applied* to one argument `t3` at the binding of g_1, partially applied to two arguments `t3` and `t4` at the binding of g_2 and fully applied to the three arguments `t3`, `t4`, and `t5` at the binding of `a`.

The information for a partial application of a curried function `f` to n arguments can be represented as a pair consisting of `f` and a list of the n arguments (which are all required to be value expressions).

To identify all applications of curried functions, create a map M from variables to partial applications. Traverse the program in a top-down manner. At each application `val a = g x`, do the following:

1. If g is a curried function, then add an entry mapping `a` to the partial application $(\mathtt{g}, [x])$.
2. If `g` is in the domain of M, and $M(\mathtt{g}) = (\mathtt{f}, l)$, then add an entry mapping `a` to the partial application $(\mathtt{f}, l')$, where l' is l with x appended to its end.

- Phase 3 (transformation): create uncurried versions of functions.

  For each curried function $\mathtt{f}_1$ of the form:

```
let fix f₁ = λx₁.let fix f₂ = λx₂. ... let fix fₙ=λxₙ.e
                                         in fₙ
                                         end
             in f₂
             end
in ...
end
```

change its definition to have the form

```
let fix f₁' = λx₁,x₂ ... xₙ.e
    fix f₁ = λx₁'.let f₂ = λx₂'. ... let fₙ=λxₙ'
                                       in f₁'(x₁',x₂',... xₙ')
                                       end
                   in f₂
                   end
in ...
end
```

- Phase 4 (transformation): replace applications of curried functions with calls to the uncurried versions.

  1. For each variable x in the domain of M which maps to a fully applied curried function *i.e.*, such that $M(x) = (f, [v_1 \ldots v_n])$, where f is a curried function with arity n, replace the binding

     ```
     let val x = ...
     ```

     with

     ```
     let val x = f(v₁ ... vₙ)
     ```

  2. Do dead-code elimination with all partial applications of curried functions marked as pure. Note that these applications cannot be deleted outright, because the variables bound to those applications may be used elsewhere. For example, suppose `f` is a curried function of two arguments. The following may occur.

     ```
     let val x = f a
         val y = x b
     in ... x ...
     end
     ```

5.5.2 Asymptotic complexity

Under the assumption that some small constant c bounds the arities of curried functions, the running time of this algorithm is $O(N \log N)$, where N is the size of the progam. Each of the four main phases of the algorithm has an $O(N \log N)$ running time:

- Phase 1: scan the program to identify functions to uncurry.

  This phase traverses the program syntax tree in N steps. At each step, a function could be marked as a candidate for uncurrying, which takes log(N) time, with maps implemented using balanced trees. Hence, the running time of this phase is $O(N \log N)$.

- Phase 2: identify applications of curried functions.

  This phase also traverses the syntax tree in N steps. When an application node is encountered, it takes $O(\log N)$ time to process the node, assuming the map M is implemented as a balanced tree. The list in the pair stored in M can be kept in reverse order, so the append can be replaced by a constant-time cons operation. Thus, the running time of this phase is $O(N \log N)$.

- Phase 3: created uncurried versions of functions.

  Again, this phase traverses the syntax tree in N steps. When a function definition is encountered, the function is looked up in a map, taking $O(\log N)$ steps. If the function has arity n, then it takes $2n$ steps to create the uncurried version of the function and changed the curried version to call the uncurried version. However, because the sum of the arities of all the curried functions in the program is less than N, there are at most $2N$ steps involved in rewriting functions. Thus, this phase takes $N * \log N + 2N$ steps, and has a running time of $O(N \log N)$.

- Phase 4: replace applications of curried functions with calls to uncurried versions.

  This phase is divided into two parts. Yet again, the first part of this phase traverses a syntax tree in N steps. When it encounters the application of a function f, it looks up f in a map, taking $O(\log N)$ steps. If $f \in \mathrm{dom}(M)$, then it rewrites the application. Because I assumed some small constant c bounds the arities of curried functions, the rewriting can be done in constant time. Thus, the first part processes an application in $O(\log N)$ steps. Hence, the running time of the first part is $O(N \log N)$.

  The second part is dead-code elimination. Dead-code elimination can be done in $O(N \log N)$ time, so the asymptotic complexity of this entire phase is $O(N \log N)$.

Because each phase of the algorithm has an $O(N \log N)$ running time, the running time of the entire algorithm is $O(N \log N)$.

If I drop the assumption that some small constant c bounds the arities of curried functions, then the running time of phase 4 is $O(N^2)$. Thus, the running time of the entire algorithm becomes $O(N^2)$. However, empirical evidence demonstrates that $c = 5$ is a reasonable assumption.

5.5.3 Related work

This uncurrying strategy was inspired by the uncurrying strategy suggested by Appel [7]. His uncurrying strategy is to apply a simple syntactic transformation and then do inline expansion. His transformation is phrased in terms of CPS programs, but it can be used for B-form programs easily.

The transformation takes a B-form expression of the form

```
let fix f = λx.
        let fix g = λy = e
        in g
        end
in f
end
```

where g is not recursive, to an expression of the form:

```
let fix f = λx.let fun g = λy.f'(x,y)
            in g
            end
    and f' = λx,y.e
in f
end
```

The transformation η-expands the body of `g` and hoists the resulting function to the scope enclosing `g`.

Inlining then replaces any fully-applied occurrences of `f` with calls to `f'`. At any points where `f` is fully applied, `f` and then `g` will be inlined because the sizes of these functions are now small constants. This results in the fully-applied occurrence of `f` being replaced with a call to `f'` (for this to work correctly, inline expansion of `f'` in the body of g must be forbidden).

This always uncurries recursive functions of two arguments: if `e` has a recursive fully-applied call to `f` in it, that call will be replaced with a call to `f'`.

Unfortunately, this uncurrying strategy is not guaranteed to uncurry recursive functions of more than two arguments, because it is underspecified. To uncurry functions of several arguments, the syntactic transformation has to be applied several times in the correct order. Consider the SML code

```
let fun f a b c = e
in f
end
```

Its B-form representation is:

```
let fix f = λa.
       let fix g = λb.
              let fix h = λc.e
              in h
              end
       in g
       end
in f
end
```

The transformation can be applied either to the inner `let`-declaration which binds `g` or the entire expression. The result of applying it to the inner let-expression is

```
let fix f = λa.
      let fix g = λb.
             let fun h = λc.g'(b,c)
             in h
             end
          and g' = λb,c.e
      in g
      end
in f
end
```

Because the declaration that binds g also binds g', this expression does not fit the form for the transformation. and the transformation cannot be applied again.

The result of applying the transformation to the outermost let-declarations is

```
let fix f = λa.
       let fun g = λb.f'(a,b)
       in g
       end
     and f' = λa,b.
          let fix h = λc.e
          in h
          end
in f
end
```

In this case, the transformation can be applied again to f'. The resulting code is

```
let fix f = λa.
      let fun g = λb.f'(a,b)
      in g
      end
    and f' = λa,b.
          let fix h  = λc.f''(a,b,c)
          in h
          end
    and f'' = λa,b,c.e
in f
end
```

Now, assuming enough inlining, any fully applied occurrences of f will be uncurried.

In fact, uncurrying as implemented in the SML/NJ compiler applies the transformation in the wrong order, and as a result does not uncurry functions of more than two arguments.

Note that even applying the transformation in the right order may not succeed in uncurrying a function with many arguments. There is a crucial assumption that "enough" inlining occurs. I believe that my algorithm, even though it is more complex than this algorithm, is better because it reliably uncurries functions with many arguments.

5.6 Other optimizations

The optimizer also implements the following optimizations, which were originally described for SML by Appel and Jim [7, 10, 9]:

- **dead-code elimination:** eliminate unreferenced, pure expressions and functions.
- **constant folding:** reduce arithmetic operations, switches, and typecases on constant values, as well as projections from known records.
- **sinking:** push pure expressions used in only one branch of a switch into that branch. However, it does not push them into function definitions.

- **inlining switch continuations:** Inline the continuation of a switch when all but one branch raises an exception. For example, it transforms the expression

  let x = if y then e_2 else raise e_3
  in e_4
  end

  to

  if y then let x = e_2 in e_4 end else raise e_3.

  This makes expressions in e_2 available within e_4 for optimizations like common subexpression elimination.

5.7 Conclusion

In this chapter, I have described the optimizer for the TIL compiler. I have shown improved algorithms for inlining and uncurrying, which are important for eliminating higher-order and polymorphic functions. For inlining, I have shown that the order in which inlining is done — doing inlining in a function before considering inlining it — can produce better code for loops. For uncurrying, I have presented an algorithm which does a better job of uncurrying functions with a large number of arguments than the algorithm presented by Appel [7].

In Chapter 10, I will show that the algorithms for uncurrying and inlining eliminate nearly all higher-order or polymorphic functions when whole programs are compiled.

I found that using a typed intermediate language did make the optimizer bigger, but that types did not constrain any of the optimizations that I presented in this chapter. For example, inlining had to handle polymorphic functions in addition to non-polymorphic functions. It also had to propagate type information when it copied the body of a function. In general, this was the case for the optimizations that I presented here: they had to handle more language constructs and had to do some additional work to propagate type information.

I found that although using a typed intermediate language required more code, the typed intermediate language simplified the process of debugging the optimizer: typechecking a program before and after each optimization phase often helped me to quickly find which phase contained the bug.

Thus, I claim that a typed intermediate language can be used in an optimizer to support improved data representations without significantly burdening the implementor and without constraining optimization.

Chapter 6

Loop optimizations

In this chapter, I describe new algorithms that show how to apply several "loop" optimizations to SML programs. I focus on two sets of optimizations: code motion optimizations, such as common-subexpression elimination and invariant removal, and array-bounds checking optimizations. I could have, however, chosen a different set of optimizations, such as strength reduction, invariant removal, and loop unrolling, to demonstrate the importance of applying "loop" optimizations to recursive functions.

I chose the code motion optimizations because I believe they are likely to demonstrate the importance of applying "loop" optimizations: these kinds of optimizations are well-known to be important for more conventional languages [21]. The array-bounds checking optimizations are intended to support the code motion optimizations: Chow shows that array bounds checking can interfere with code motion optimizations [21].

It is important to emphasize that the algorithms are new, even though most of the optimizations are well-known. I designed new algorithms because I apply these optimizations to programs in a λ-calculus-based intermediate language (B-form). This makes it impossible to apply the textbook versions of optimizations [3]: the languages used in the textbooks differ too much from from the λ-calculus. Those languages are first-order, have a flat name space, are imperative and have looping constructs. In contrast, my λ-calculus-based intermediate language is higher-order, has lexical scoping, emphasizes variable binding instead of assignment, and uses only recursion.

I believe that there are several advantages to implementing these optimizations on a higher-level λ-calculus based language instead of a machine-level language such as RTL, triples, or SSA. First, there are some computations that are easier to move when they are represented at a high level, such as record creation and constructor-level computations. Second, and more fundamentally, the optimizer can use the invariants of the λ-calculus involving scoping of variables. In contrast, with a machine-level language, the compiler must decide how to represent environments for functions before translating to the machine-level language. This introduces data structures that seem likely to cause trouble for a traditional machine-level optimizer. For example, consider loop-invariant removal of an expression that uses values fetched from a closure: the optimizer would have to figure out that the data structure representing the closure was also invariant.

One important question is whether my algorithms are practical: are they efficient enough to be used in practice or on large programs? To address this question, I study the algorithmic efficiency of my algorithms. Most of the algorithms have an $O(N \log N)$ asymptotic complexity.

I have organized this chapter in the following manner. First, I discuss common-subexpression elimination. Second, I discuss eliminating redundant switches, the control-flow analog of common-subexpression elimination. Third, I discuss a limited form of invariant removal — hoisting constant expressions to the tops of programs. Fourth, I discuss a general form of invariant removal. I distinguish between the two forms of invariant removal because I believe that the limited form is likely to be more specific to TIL's use of intensional polymorphism: inlining polymorphic functions can eliminate the cost of passing types around, but some form of invariant removal is needed to reduce the cost of constructing (monomorphic) types. Fifth, I cover eliminating redundant comparisons, an optimization for reducing the cost of array-bounds checking. Finally, I discuss how to order all the optimizations discussed in this chapter and the previous chapter.

6.1 Common-subexpression elimination

In the following discussion, I assume that every bound variable is unique. An expression is a common subexpression if it is bound to two variables whose scope overlaps. For example, consider the program fragment:

```
let b = a+a
    c = a+a
in ...
end
```

The expression `a+a` is a common subexpression. Common-subexpression elimination (CSE) removes common subexpressions that do not read, update, or allocate updateable data, such as array. It may remove expressions that raise exceptions or not terminate. For example, it transforms the program fragment given above to

```
let b = a+a
    c = b
in ...
end
```

I can state the result of CSE for B-form programs succinctly. I say that all common subexpressions have been eliminated from a program P if

- For all declarations in P of the form

  let a = e
  in d
  end

  if e does not contain any declarations nested within it and the effect of e is a subset of $\{$E,N$\}$ according to Algorithm 1, then e does not appear in d.

- For all constructor declarations in P of the form

```
let t = μ
in d
end
```

if μ does not contain any constructor declarations nested within it, then μ does not appear in d.

6.1.1 An algorithm for eliminating common subexpressions

I present Algorithm 6, which does CSE for B-form programs, in Figures 6.1 through 6.4. The algorithm is given as a large number of mutually-recursive functions that traverse program syntax trees. Figure 6.1 defines functions that decide which expressions and constructors can be eliminated. Figures 6.2 and 6.3 contain functions for traversing expressions and declarations, and Figure 6.4 contains functions for traversing constructors and types.

The functions for traversing expressions and declarations take a program term and two maps as arguments. The first map, ae, contains available expressions. The second map, ac, contains available constructors. Available expressions are simply expressions that have been computed and bound to variables. The map ae maps expressions to the variables to which they are bound and the types of those variables, while ac maps constructors to the constructor variables to which they are bound.

Most of the functions rewrite program terms. Note that the functions rewrite types to eliminate common constructors from them, in addition to rewriting expressions and constructors.

The function $\mathrm{c_e}$ eliminates common subexpressions in terms. This occurs in the case that analyzes let-bound expressions of the form `let` $x : \sigma = e$ `in` b . The function first rewrites e to eliminate any common subexpressions in e. It binds the result to e'. The function next checks whether e' can be eliminated. To make CSE asymptotically efficient, expressions that contains expressions within them, such as `switch` expressions, are not eliminated. Expressions that use the store also are not eliminated, because that could change the meaning of the program. If e' can be eliminated, the function checks whether e' is an available expression that is already bound to some variable y (that is, it checks whether e' is in the domain of ae). If so, the function then compares the types of e' and y. If these are the same, the function replaces e' with y. If e' is not an available expression, the function extends the mapping ae to note that e' is bound to x, and rewrites the body of the let declaration using the new mapping.

The check that types are equal is needed to preserve the type correctness of programs. It is possible for two expressions to be identical syntactically, but to have different types. For example, this may occur with operators that create an element of a disjoint sum, such as the `inject` operator and the `enum_enumorsum` operator. The term `inject(1,"abc")` may have a different sum type every place it is used.

Note that checking whether two types are equal is non-trivial in this language because constructors may be bound to variables and, more fundamentally, the language of constructors is

$$
\begin{aligned}
\texttt{is\_elim\_exp}(e) \quad &= \quad \text{case } e \text{ of} \\
&\qquad \lceil \texttt{switch } \ldots \rceil : \text{false} \\
&\qquad \lceil \texttt{typecase } \ldots \rceil : \text{false} \\
&\qquad \lceil \texttt{tlistcase } \ldots \rceil : \text{false} \\
&\qquad \text{otherwise } : \texttt{eff}(e) \subset \{N, E\} \\
\\
\texttt{is\_elim\_con}(\mu) \quad &= \quad \text{case } \mu \text{ of} \\
&\qquad \lceil \texttt{Fold } \ldots \rceil : \text{false} \\
&\qquad \lceil \texttt{Listcase } \ldots \rceil : \text{false} \\
&\qquad \lceil \texttt{Typecase } \ldots \rceil : \text{false} \\
&\qquad \text{otherwise } : \text{true}
\end{aligned}
$$

Figure 6.1: Algorithm 6: functions to decide which expressions and constructors can be eliminated.

rich. In general, to check whether two types are equal you need to create a constructor environment, normalize the types, and then compare the normalized types for syntactic equality. The constructor environment maps free constructor variables to constructors. Normalization places all equal types in the same syntactic form by applying constructor functions, reducing `Typecase`s, and so forth. It is needed because of the richness of the constructor language.

A simple example of a program where type normalization is required is:

```
con s::Ω → Ω = λt :: Ω.Record [t,t]

val x : T(s int) = record (5,5)
val y : T(Record[Int,Int]) = record (5,5)
```

The types `T(s int)` and `T(Record [int,int])` are in fact equal under normalization.

You can compare types for syntactic equality, but that misses some equivalent types (such as those given in the preceding example). In practice, I use full normalization to check for type equality, but I have not studied whether simple syntactic equality produces equally good results.

Returning to the description of the algorithm for CSE, the function $\mathrm{cb_e}$ eliminates common subexpressions in constructors. The logic it uses for eliminating common constructors is similar to that used by $\mathrm{c_e}$ for eliminating common subexpressions. Note that the function $\mathrm{cb_e}$ differs from other functions in its return value. Other functions simply return a new term, while $\mathrm{cb_e}$ returns a rewritten constructor binding and an updated map of available constructors. This allows $\mathrm{cb_e}$ to be used in two places: when rewriting constructor variable bindings at the level of expressions and also when rewriting constructor variable bindings at the level of constructors. The function $\mathrm{cb_t}$rewrites types: if a type is a constructor, it rewrites the constructor. Otherwise, it rewrites the type in the obvious way (the "..." indicates this).

$\mathrm{c_d}(d, ae, ac) =$
case d of
$\lceil v : \sigma \rceil : \lceil v : \sigma \rceil$
$\lceil \texttt{let } x : \sigma = e \texttt{ in } b \rceil$:
    let $e' = \mathrm{c_e}(e, ae, ac)$
    in if is_elim_exp(e')
        then if $ae(e') = (y, \tau)$ and $\sigma = \tau$
            then $\lceil \texttt{let } x : \sigma = y \texttt{ in } \mathrm{c_d}(b, ae, ac) \rceil$
            else $\lceil \texttt{let } x : \sigma = e' \texttt{ in } \mathrm{c_d}(b, ae[e' \mapsto x], ac) \rceil$
        else $\lceil \texttt{let } x : \sigma = e' \texttt{ in } \mathrm{c_d}(b, ae, ac) \rceil$
    end
$\lceil \texttt{let } cb \texttt{ in } d \rceil$:
    let $(cb', ac') = \mathrm{c_{cb}}(cb, ac)$
    in $\lceil \texttt{let } cb' \texttt{ in } \mathrm{c_d}(d, ae, ac') \rceil$
    end
$\lceil \texttt{let fix } x_1 : \sigma_1 = f_1 \ldots \texttt{ and } x_n : \sigma_n = f_n \texttt{ in } d' \rceil$:
    $\lceil \texttt{let fix } x_1 : \mathrm{c_t}(\sigma_1, ac) = \mathrm{c_f}(f_1, ae, ac) \ldots \texttt{ and } x_n : \mathrm{c_t}(\sigma_n, ac) = \mathrm{c_f}(f_n, ae, ac)$
     $\texttt{in } \mathrm{c_d}(d', ae, ac) \rceil$
$\lceil \texttt{let fixtype } x_1 : \sigma_1 = tf_1 \ldots \texttt{ and } x_n : \sigma_n = tf_n \texttt{ in } d' \rceil$:
    $\lceil \texttt{let fixtype } x_1 : \mathrm{c_t}(\sigma_1, ac) = \mathrm{c_f}(tf_1, ae, ac) \ldots \texttt{ and } x_n : \mathrm{c_t}(\sigma_n, ac) = \mathrm{c_f}(tf_n, ae, ac)$
     $\texttt{in } \mathrm{c_d}(d', ae, ac) \rceil$
$\lceil \texttt{let } x_1 : \sigma = d_1 \texttt{ handle } f \texttt{ in } d_2 \rceil$:
    $\lceil \texttt{let } x_1 : \mathrm{c_t}(\sigma, ac) = \mathrm{c_d}(d_1, ae, ac) \texttt{ handle } \mathrm{c_f}(f, ae, ac) \texttt{ in } \mathrm{c_d}(d_2, ae, ac) \rceil$
$\lceil \texttt{raise } (v, \sigma) \rceil$:
    $\lceil \texttt{raise } (v, \sigma) \rceil$

$\mathrm{c_e}(e, ae, ac) =$
case e of
$\lceil \texttt{switch } (st)\, v \texttt{ of } (i : f_i)^* \texttt{ default: } od \rceil$:
    $\lceil \texttt{switch } (st)\, v \texttt{ of } (i : \mathrm{c_f}(f_i, ae, ac))^* \texttt{ default: } \mathrm{c_d}(od, ae, ac) \rceil$
$\lceil \texttt{typecase } cv \texttt{ of } [t.\sigma] ta_i^* \texttt{ rectype } : otf \texttt{ default} : od \rceil$:
    $\lceil \texttt{typecase } cv \texttt{ of } [t.\mathrm{c_t}(\sigma, ac)] \mathrm{c_{ta}}(ta_i, ae, ac)^*$
     $\texttt{rectype } : \mathrm{c_{tf}}(otf, ae, ac)$
     $\texttt{default} : \mathrm{c_d}(od, ae, ac) \rceil$
$\lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : d \texttt{ cons} : tf \rceil$:
    $\lceil \texttt{tlistcase } cv \texttt{ of } [t.\mathrm{c_t}(\sigma, ac)] \texttt{ nil } : \mathrm{c_d}(d, ae, ac) \texttt{ cons} : \mathrm{c_{tf}}(tf, ae, ac) \rceil$
otherwise : e

Figure 6.2: Algorithm 6, part 1: traversing expressions and declarations

$$
\begin{array}{ll}
\mathrm{c_f}(f, ae, ac) & = \\
 & \text{case } f \text{ of} \\
 & \lceil \lambda(x_1 \ldots x_n).d \rceil \ : \lceil \lambda(x_1 \ldots x_n).\mathrm{c_d}(d, ae, ac) \rceil \\
 & \\
\mathrm{c_{tf}}(tf, ae, ac) & = \\
 & \text{case } tf \text{ of} \\
 & \lceil \Lambda(t_1 \ldots t_n).d \rceil \ : \lceil \Lambda(t_1 \ldots t_n).\mathrm{c_d}(d, ae, ac) \rceil \\
 & \\
\mathrm{c_{ta}}(a, ae, ac) & = \\
 & \text{case } a \text{ of} \\
 & \lceil c_0 : d \rceil : \lceil c_0 : \mathrm{c_d}(d, ae, ac) \rceil \\
 & \lceil c_1 : tf \rceil : \lceil c_1 : \mathrm{c_{tf}}(tf, ae, ac) \rceil
\end{array}
$$

Figure 6.3: Algorithm 6, part 2: traversing expressions and declarations.

6.1.2 Correctness

For a lazy functional language, we can show the correctness of code motion optimizations such as CSE by using the β rule of the λ-calculus:

$$(\lambda x.M)N = M[N/x]$$

This rule states that the result of applying a function to an expression is equivalent to substituting the expression into the body of the function. For example,

$$(\lambda x.x + x)e_1 = e_1 + e_1$$

This rule corresponds to a call-by-name parameter passing strategy.

Note that we can also phrase the β rule in terms of `let`, because `let x =` N `in` M `end` is simply syntactic sugar for $(\lambda x.M)N$:

$$\texttt{let } x = N \texttt{ in } M \texttt{ end} = M[N/x]$$

Rephrasing the previous example,

$$\texttt{let } x = e_1 \texttt{ in } x + x \texttt{ end} = e_1 + e_1$$

To eliminate a common subexpression using the β rule, we do a β-expansion (convert $M[N/x]$ to $(\lambda x.M)N$ or `let` $x = N$ `in` M `end`), followed by several β-reductions.

For example,

```
let s = a+a
    t = b*b
    u = a+a
in s+t+u
end
```

$$
\begin{array}{lcl}
\mathrm{c}_{\mathrm{cb}}(cb, ac) & = & \text{case } cb \text{ of} \\
& & \lceil t :: \kappa = \mu \rceil : \\
& & \quad \text{let } \mu' = \mathrm{c}_{\mathrm{con}}(\mu, ac) \\
& & \quad \text{in if is\_elim\_con}(\mu') \\
& & \qquad \text{then if } ac(\mu') = t' \\
& & \qquad\qquad \text{then } (\lceil t :: \kappa = t' \rceil, ac) \\
& & \qquad\qquad \text{else } (\lceil t :: \kappa = \mu' \rceil, ac[\mu' \mapsto t]) \\
& & \qquad \text{else } (\lceil t :: \kappa = \mu' \rceil, ac) \\
& & \quad \text{end} \\
& & \lceil t :: \kappa = \lambda t' :: \kappa'.cd \rceil : \\
& & \quad (\lceil t :: \kappa = \lambda t' :: \kappa'.\mathrm{c}_{\mathrm{cd}}(cd, ac) \rceil, ac) \\
& & \lceil t :: \kappa = \texttt{Rectype}(t_i, cd_i)^*.cd' \rceil : \\
& & \quad (\lceil t :: \kappa = \texttt{Rectype}(t_i, \mathrm{c}_{\mathrm{cd}}(cd_i, ac)).\mathrm{c}_{\mathrm{cd}}(cd', ac) \rceil, ac) \\
\\
\mathrm{c}_{\mathrm{con}}(\mu, ac) & = & \text{case } \mu \text{ of} \\
& & \lceil \texttt{Typecase } cv \texttt{ of } a_i^* \texttt{ default } cd :: \kappa \rceil : \\
& & \quad \lceil \texttt{Typecase } cv \texttt{ of } \mathrm{c}_{\mathrm{tca}}(a_i, ac) \texttt{ default } \mathrm{c}_{\mathrm{cd}}(cd, ac) :: \kappa \rceil \\
& & \lceil \texttt{Fold } cv \texttt{ of } \texttt{ Nil } : cd_1 \texttt{ Cons } : \lambda t :: \kappa.cd_2 \rceil : \\
& & \quad \lceil \texttt{Fold } cv \texttt{ of } \texttt{ Nil } : \mathrm{c}_{\mathrm{cd}}(cd_1, ac) \texttt{ Cons } : \lambda t :: \kappa.\mathrm{c}_{\mathrm{cd}}(cd_2, ac) \rceil \\
& & \lceil \texttt{Listcase } cv \texttt{ of } \texttt{ Nil } : cd_1 \texttt{ Cons } : \lambda t :: \kappa.cd_2 \rceil : \\
& & \quad \lceil \texttt{Listcase } cv \texttt{ of } \texttt{ Nil } : \mathrm{c}_{\mathrm{cd}}(cd_1, ac) \texttt{ Cons } : \lambda t :: \kappa.cd_2 \rceil \\
& & \text{otherwise} : \mu \\
\\
\mathrm{c}_{\mathrm{tca}}(a, ac) & = & \text{case } a \text{ of} \\
& & \lceil c_0 : cd \rceil : \lceil c_0 : \mathrm{c}_{\mathrm{cd}}(cd, ac) \rceil \\
& & \lceil c_1 : \lambda t :: \kappa.cd \rceil : \lceil c_1 : \lambda t :: \kappa.\mathrm{c}_{\mathrm{cd}}(cd, ac) \rceil \\
\\
\mathrm{c}_{\mathrm{cd}}(cd, ac) & = & \text{case } cd \text{ of} \\
& & \lceil cv \rceil : \lceil cv \rceil \\
& & \lceil \texttt{Let } cb \texttt{ in } cd' \rceil : \\
& & \quad \text{let } (cb', ac') = \mathrm{c}_{\mathrm{cb}}(cb, ac) \\
& & \quad \text{in } \lceil \texttt{Let } cb' \texttt{ in } \mathrm{c}_{\mathrm{cd}}(cd', ac') \rceil \\
& & \quad \text{end} \\
\mathrm{c}_{\mathrm{t}}(\sigma, ac) & = & \text{case } \sigma \text{ of} \\
& & \lceil \texttt{T } cd : \mathrm{c}_{\mathrm{cd}}(cd, ac) \rceil \\
& & \ldots
\end{array}
$$

Figure 6.4: Algorithm 6, part 3: traversing constructors and types.

is equivalent by β expansion to

```
let x = a+a
    s = x
    t = b*b
    u = x
in s+t+u
end
```

By β reduction, we can substitute `x` for `s` and `x` for `u` to obtain:

```
let x = a+a
    t = b*b
in x+t+x
end
```

If we want to preserve variable names, we can rename `x` to `s`:

```
let s = a+a
    t = b*b
in s+t+s
end
```

SML, however, is not a lazy functional language. It uses a call-by-value parameter passing strategy, instead of a call-by-name parameter passing strategy. For SML, we can only use the weaker β_v rule [70]:

$$(\lambda x.M)V = M[V/x]$$

where V is a value. A value is a constant, such as 0, a variable, or a λ-expression. This rule does not allow us to prove correctness for any interesting cases of CSE, where computation is being done.

To prove the correctness of CSE of expressions that do not raise exceptions and do terminate, we can extend the syntactic class of values beyond constants, variables, and λ-expressions to include other effect-free expressions such as creating a record from values and selecting a field of a record.

Proving the correctness of CSE of expressions that may raise exceptions or not terminate requires more than extending the syntactic class of values, however. For example, it is obvious that for the expression

```
let x = a+a
    ...
    y = a+a
in e
end
```

the second occurrence of `a+a` must not raise an overflow exception. At that point in the program, we know that `a+a` is equivalent to a value v that is bound to `x`. The β rule, however, does not support this form of contextual reasoning.

To demonstrate the correctness of CSE of expressions that may raise exceptions or not terminate, we can use *contextual assertions* [43, 60] or *conditional lambda-theories* [93]. These variants of the call-by-value λ-calculus justify the correctness of the transformations that are done by Algorithm 6.

The important point is that there is a strong connection between the transformations that are done by Algorithm 6 and variants of the call-by-value λ-calculus.

6.1.3 Asymptotic complexity

The maps for available expressions and available constructors can be implemented using balanced trees. The comparison operators for expressions and constructors can use a lexicographic ordering. Note that because expressions and constructors containing nested declarations are excluded, all constructors and expressions stored in these maps can be regarded as having constant size[1]. This means that expressions stored in the balanced trees can be compared in constant time, and that operations on the balanced trees can be done $O(\log N)$, where N is the program size.

Because the algorithm traverses each node in the syntax tree only once, and it does only $O(\log N)$ work at each node, the algorithm has a running time of $O(N \log N)$.

6.2 Redundant switch elimination

Redundant switch elimination is the control-flow analog of common-subexpression elimination. Common-subexpression elimination removes redundant expressions; redundant switch-elimination removes redundant switches. A switch on some variable x is redundant if it is nested within a branch of another switch on x. For example, consider the following B-form expression which operates on boolean values[2]:

```
let b = switch(enum) x
        of 0 : λ().
                let y = switch(enum) x
                          of 0 : λ().enum(1)
                           1 : λ().enum(0)
                in y
                end
         1 : λ().enum(1)
in b
end
```

The nested `switch` that is bound to `y` is redundant, because we know that `x` is bound to the enumerated sum value 0 at that point.

[1] It is not strictly true that the sizes are constants because records and sums can have various sizes, but for all practical purposes they are bounded by a small finite constant.

[2] false is the enumerated sum value 0, and true is the enumerated sum value 1.

Redundant switches arise naturally from array-bounds checking and after inlining and CSE. For example, consider a simple loop that sums the contents of an array A. The SML version is:

```
val bound = len A
fun loop (cnt,sum) =
      if cnt<bound then loop(cnt+1,sum+sub1(A,cnt))
      else sum
val s = loop(0,0)
```

The B-form version of the loop function, after making array bounds checking explicit, is:

```
fix loop = λ(cnt,sum).
             let t1 = cnt<bound
                 t2 = switch(enum) t1
                      of 0 : λ().sum
                         1 : λ().
                           let cnt' = cnt+1
                               t3 = cnt >= 0
                               t5 = switch (enum) t3
                                      of 0 : enum 0
                                         1 : let t4 = cnt < bound
                                                 t6 = switch(enum) t4
                                                      of 0 : enum 0
                                                         1 : enum 1
                                             in t6
                                             end
                               elem = switch(enum) t5
                                      of 0 : λ().raise Subscript
                                         1 : λ().let t7 = sub1(A,cnt)
                                              in t7
                                              end
                               sum' = sum + elem
                               t8 = loop(cnt',sum')
                           in t8
                           end
             in t2
             end
```

The switch expression bound to $t5$ computes the bounds check predicate $0 \leq$ `cnt` $<$ `bound` and the switch expression bound to `elem` branches on the predicate. CSE replaces `t4` with `t1`, so the subsequent switch becomes redundant; `t6` is known to be bound to `enum 1`. In larger programs, CSE and inlining interact to expose unnecessary switches.

The algorithm for eliminating switches, Algorithm 7, is similar conceptually to Algorithm 6 for CSE. Because the algorithms are similar, I only sketch the changes needed to Algorithm 6. Instead of keeping a map of available expressions and available constructors, the algorithm keeps a map M that tells which switch arms (branches) on variables with sum types have been taken. The algorithm infers this information when it traverses particular arms of switches. For example, when the algorithm processes the switch on `t1` that is bound to `t2` in the previous example, the map contains an entry $\{t1 \mapsto 0\}$ when the algorithm traverses the 0 arm, and $\{t1 \mapsto 1\}$ when the algorithm traverses the 1 arm. In addition, the map also contains the list of variables bound by that arm. This list is empty for the branches in the previous example because the arms do not bind any variables.

When the algorithm encounters a switch on some variable x, it checks to see whether $x \in \mathrm{dom}(M)$. If so, and if $M(x) = (i, vl)$, the algorithm deletes the switch and replaces it with the body of branch i. It also binds the variables bound by the arm i to the variables in vl. If a branch i does not exist, then it uses the default branch. All switches in B-form are required to be complete, so either branch i or a default branch must exist.

It is straightforward to extend the algorithm to eliminate redundant branches on constructors, such as those done by `typecase` and `tlistcase`. The current implementation of the algorithm does not eliminate redundant branches on constructors, however. It would be interesting to extend the implementation and investigate the benefits of eliminating redundant branches on types. I would expect eliminating redundant branches on types to be especially important for programs that are separately compiled, where the compiler cannot eliminate polymorphism.

We can show the correctness of the transformations done by Algorithm 7, like the correctness of the transformations done by Algorithm 6, using variants of the call-by-value λ-calculus. Specifically, we need to use contextual assertions [43, 60] or conditional lambda-theories [93] to show the correctness of the transformations done by Algorithm 7.

Algorithm 7 has an $O(N \log N)$ asymptotic complexity, like Algorithm 6. The algorithm traverses each node in the program syntax tree. The algorithm processes each node using a constant number of operations on maps on variables. If you implement these maps using balanced trees, each operation takes at most $O(\log N)$ time. Thus, the total running time of Algorithm 7 is $O(N \log N)$. This could be improved to an average running time of $O(N)$, by implementing maps using hash tables, at the expense of a slightly more complicated program (after the algorithm processes a branch of a switch, it needs to delete the corresponding entry for the variable from the hash table).

6.3 Hoisting constant expressions

An important optimization is moving expressions from frequently-executed parts of programs to less frequently-executed parts of the programs. A special case of this is hoisting all constant expressions and constructors to the tops of programs. This ensures that constant expressions and constructors are evaluated only once, instead of perhaps being evaluated many times within recursive functions.

The definitions of constant expressions and constructors are more subtle than what you might expect. Consider defining an expression to be constant if it uses only constants, that is, if it has no free variables. The problem with this definition is that B-form forces expressions that intuitively should be constant, such as a constant list, to contain free variables. For example, in the following program:

```
let a = 2 :: nil
    b = 1 :: a
in ...
end
```

the expression bound to `a` is constant with this definition, while the expression bound to `b` is not. This problem can be corrected by regarding an expression as constant if all its free variables are bound to constant expressions or constructors.

This definition is still incomplete for two reasons. First, an expression may be constant, but its type may not be. For example, the expression `inject(1,1)`, which injects an integer into a sum, is constant, but the sum type may not be constant. Hoisting the sum to the top of the program could move the expression out of the scope of its free constructor variables. Thus, an expression is constant only if the type of the expression is constant. Second, an expression may have an effect. For correctness, moving a constant expression should not change the meaning of the program. In other words, constant expressions must be pure.

The following mutually-recursive definitions summarize these requirements:

Definition 1 (Constant expressions, constant constructors, and constant types) *An expression is constant if*

- *all its free variables are bound to constant expressions or constructors,*
- *its type is constant,*
- *and it is pure.*

A constructor is constant if all its free constructor variables are bound to constant constructors.

A type is constant if all its free constructor variables are bound to constant constructors.

6.3.1 An algorithm for hoisting

Figures 6.5 through 6.7 present Algorithm 8, which hoists constant expressions and constant constructors to the tops of programs. The algorithm is divided into an analysis phase and a transformation phase. Figure 6.5 contains the analysis phase and Figure 6.6 and Figure 6.7 contain the transformation phase. The analysis phase computes two sets, E and C, which contain the names of constant expressions and constant constructors, respectively. It traverses programs in a top-down manner. When it encounters a `let`-bound expression, it first decides whether the expression is constant and should be moved before traversing the body of the `let`-declaration. It computes the free variables of the expression, and checks to see that they are in E and C. It also checks whether the expression has an effect. The logic for `let`-bound constructors is similar.

The transformation phase rewrites programs in a top-down manner. Given a program P, it rewrites P to delete constant expressions and constructor bindings. It stores the deleted expressions and constructors in the list B. It then prefixes the bindings in B to the rewritten program (note that the list B is kept in reverse order).

When the phase encounters a `let`-declaration of the form `let` $x : \sigma = e$ `in` d, it first rewrites e to e'. Next, it checks whether e is a constant expression. If so, it deletes the binding of x and adds a binding with e' to the list B. The logic for constructors differs only slightly: the constructor is not rewritten.

6.3.2 Correctness

We can show the correctness of the transformations done by this optimization like we show the correctness of the transformations done by CSE, by using variants of the call-by-value λ-calculus.

For the simple case of moving constants and variables, we can use the β_v rule. For example, to move z to the top of the following program:

```
let x = 5
    fix f y =
       let z = 3
       in z * y
       end
in f x
end
```

we can do a β_v-expansion:

```
let z' = 3
    x = 5
    fix f y =
      let z = z'
      in z * y
      end
in f x
```

followed by a β_v-reduction to substitute `z'` for `z`:

```
let z' = 3
    x = 5
    fix f y = z' * y
in f x
end
```

We can then rename `z'` to `z`

```
let z = 3
    x = 5
    fix f y = z * y
in f x
end
```

$$
\begin{array}{lcl}
E & = & \texttt{ref}\ \emptyset \\
\\
C & = & \texttt{ref}\ \emptyset \\
\\
\mathrm{c_d}(d) & = & \text{case } d \text{ of} \\
& & \lceil v:\sigma \rceil : () \\
& & \lceil \texttt{let}\ x:\sigma = e\ \texttt{in}\ d' \rceil\ : \\
& & \quad \text{let } f = \mathrm{fv}(e) \cup \mathrm{fv}(\sigma) \\
& & \quad \text{in if } \forall x \in f.x \in (!E) \text{ and } \forall t \in f.t \in (!C) \text{ and } \mathrm{eff_e}(e) = \emptyset \\
& & \qquad \text{then } E\ := (!E) \cup \{x\} \\
& & \qquad \text{else } (); \\
& & \quad \mathrm{c_d}(d') \\
& & \quad \text{end} \\
& & \lceil \texttt{let}\ cb\ \texttt{in}\ d \rceil\ : \\
& & \quad \text{if } \forall t \in \mathrm{fv}(cb).t \in (!C) \\
& & \quad \text{then } C\ := (!C) \cup \{\mathrm{getv}(cb)\} \\
& & \quad \text{else } () \\
& & \lceil \texttt{let fix}\ x_1:\sigma_1:f_1 \ldots\ \texttt{and}\ x_n:\sigma_n = f_n\ \texttt{in}\ d' \rceil\ : \\
& & \quad (\forall i.\mathrm{c_f}(f_i); \mathrm{c_d}(d')) \\
& & \lceil \texttt{let fixtype}\ x_1:\sigma_1:tf_1 \ldots\ \texttt{and}\ x_n:\sigma_n = tf_n\ \texttt{in}\ d' \rceil\ : \\
& & \quad (\forall i.\mathrm{c_{tf}}(tf_i); \mathrm{c_d}(d')) \\
& & \lceil \texttt{let}\ x_1:\sigma = d_1\ \texttt{handle}\ f\ \texttt{in}\ d_2 \rceil\ : \\
& & \quad (\mathrm{c_d}(d_1); \mathrm{c_f}(f); \mathrm{c_d}(d_2)) \\
& & \lceil \texttt{raise}\ (v,\sigma) \rceil\ : () \\
\\
\mathrm{c_e}(e) & = & \text{case } e \text{ of} \\
& & \lceil \texttt{switch}\ (st)\, v\ \texttt{of}\ (i:f_i)^*\ \texttt{default:}\ \ od \rceil\ : \\
& & \quad (\forall i.\mathrm{c_f}(f_i); \mathrm{c_d}(od)) \\
& & \lceil \texttt{typecase}\ cv\ \texttt{of}\ [t.\sigma]ta_i^*\ \texttt{rectype}\ :otf\ \texttt{default}:od \rceil\ : \\
& & \quad (\forall i.\mathrm{c_{ta}}(ta_i); \mathrm{c_{tf}}(otf); \mathrm{c_d}(od)) \\
& & \lceil \texttt{tlistcase}\ cv\ \texttt{of}\ [t.\sigma]\ \texttt{nil}\ :d\ \texttt{cons}:tf \rceil\ : \\
& & \quad (\mathrm{c_d}(d); \mathrm{c_{tf}}(tf)) \\
\\
\mathrm{c_f}(f) & = & \text{case } f \text{ of} \\
& & \lceil \lambda(x_1 \ldots x_n).d \rceil\ : \mathrm{c_d}(d) \\
\mathrm{c_{tf}}(tf) & = & \text{case } tf \text{ of} \\
& & \lceil \Lambda(t_1 \ldots t_n).d \rceil\ : \mathrm{c_d}(d) \\
\mathrm{c_{ta}}(a) & = & \text{case } a \text{ of} \\
& & \lceil c_0 : d \rceil : \mathrm{c_d}(d) \\
& & \lceil c_1 : tf \rceil : \mathrm{c_{tf}}(tf)
\end{array}
$$

Figure 6.5: Algorithm 8: identify constant expressions and constructors

$B \quad = \quad \texttt{ref nil}$

$\mathrm{r_p}(d) \quad = \quad$ let $d' = \mathrm{r_d}(d)$
in prefix(rev($!B$)$, d'$)
end

$\mathrm{r_d}(d) \quad = \quad$ case d of
$\lceil v : \sigma \rceil : \lceil v : \sigma \rceil$
$\lceil \texttt{let}\ x : \sigma = e\ \texttt{in}\ d' \rceil\ :$
  let $e' = \mathrm{r_e}(e)$ and $d'' = \mathrm{r_d}(d')$
  in if $x \in (!E)$
    then $(B\ :=\ (x, \sigma, e') :: (!B); d'')$
    else $\lceil \texttt{let}\ x : \sigma = e'\ \texttt{in}\ d'' \rceil$
  end
$\lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil\ :$
  let $d'' = \mathrm{r_d}(d')$
  in if $x \in (!C)$
    then $(B\ :=\ cb :: (!B); d'')$
    else $\lceil \texttt{let}\ cb\ \texttt{in}\ d'' \rceil$
  end
$\lceil \texttt{let fix}\ x_1 : \sigma_1 = f_1 \ldots\ \texttt{and}\ x_n : \sigma_n = f_n\ \texttt{in}\ d' \rceil\ :$
  $\lceil \texttt{let fix}\ x_1 : \sigma_1 = \mathrm{r_f}(f_1) \ldots\ \texttt{and}\ x_n : \sigma_n = \mathrm{r_f}(f_n)\ \texttt{in}\ \mathrm{r_d}(d') \rceil$
$\lceil \texttt{let fixtype}\ x_1 : \sigma_1 = tf_1 \ldots\ \texttt{and}\ x_n : \sigma_n = tf_n\ \texttt{in}\ d' \rceil\ :$
  $\lceil \texttt{let fixtype}\ x_1 : \sigma_1 = \mathrm{r_{tf}}(tf_1) \ldots\ \texttt{and}\ x_n : \sigma_n = \mathrm{r_{tf}}(tf_n)\ \texttt{in}\ \mathrm{r_d}(d') \rceil$
$\lceil \texttt{let}\ x_1 : \sigma = d_1\ \texttt{handle}\ f\ \texttt{in}\ d_2 \rceil\ :$
  $\lceil \texttt{let}\ x_1 : \sigma = \mathrm{r_d}(d_1)\ \texttt{handle}\ \mathrm{r_f}(f)\ \texttt{in}\ \mathrm{r_d}(d_2) \rceil$
$\lceil \texttt{raise}\ (v, \sigma) \rceil\ :$
  $\lceil \texttt{raise}\ (v, \sigma) \rceil$

$\mathrm{r_e}(e) \quad = \quad$ case e of
$\lceil \texttt{switch}\ (st)\, v\ \texttt{of}\ (i : f_i)^*\ \texttt{default:}\ \ od \rceil\ :$
  $\lceil \texttt{switch}\ (st)\, v\ \texttt{of}\ (i : \mathrm{r_f}(f_i))^*\ \texttt{default:}\ \ \mathrm{r_d}(od) \rceil$
$\lceil \texttt{typecase}\ cv\ \texttt{of}\ [t.\sigma] ta_i^*\ \texttt{rectype}\ : otf\ \texttt{default} : od \rceil\ :$
  $\lceil \texttt{typecase}\ cv\ \texttt{of}\ [t.\sigma] \mathrm{r_{ta}}(ta_i)^*\ \texttt{rectype}\ : \mathrm{r_{tf}}(otf)\ \texttt{default} : \mathrm{r_d}(od) \rceil$
$\lceil \texttt{tlistcase}\ cv\ \texttt{of}\ [t.\sigma]\ \texttt{nil}\ : d\ \texttt{cons} : tf \rceil\ :$
  $\lceil \texttt{tlistcase}\ cv\ \texttt{of}\ [t.\sigma]\ \texttt{nil}\ : \mathrm{r_d}(d)\ \texttt{cons} : \mathrm{r_{tf}}(tf) \rceil$
otherwise : e

Figure 6.6: Algorithm 8, part 1: move constant expressions and constructors

$$
\begin{array}{lcl}
\mathrm{r_f}(f) & = & \text{case } f \text{ of} \\
 & & \lceil \lambda(x_1 \ldots x_n).d \rceil \ : \lceil \lambda(x_1 \ldots x_n).\mathrm{r_d}(d) \rceil \\
 & & \\
\mathrm{r_{tf}}(tf) & = & \text{case } tf \text{ of} \\
 & & \lceil \Lambda(t_1 \ldots t_n).d \rceil \ : \lceil \Lambda(t_1 \ldots t_n).\mathrm{r_d}(d) \rceil \\
 & & \\
\mathrm{r_{ta}}(a) & = & \text{case } a \text{ of} \\
 & & \lceil c_0 : d \rceil : \lceil c_0 : \mathrm{r_d}(d) \rceil \\
 & & \lceil c_1 : tf \rceil : \lceil c_1 : \mathrm{r_d}(tf) \rceil
\end{array}
$$

Figure 6.7: Algorithm 8, continued: move constant expressions and constructors

To prove the correctness of hoisting other kinds of expressions, we can again use the β_v rule with an extended syntactic class of values. For example, we can use this version of the β_v rule to show the correctness of hoisting the expression `(3,x)` to the top of the following program:

```
let x = 5
    fix f y =
       let z = (3,x)
       in z
       end
in f x
end
```

We can do a β-expansion that binds `z'` to `(3,x)`:

```
let x = 5
    z' = (3,x)
    fix f y =
       let z = z'
       in z
       end
in f x
end
```

followed by a β-reduction:

```
let x = 5
    z' = (3,x)
    fix f y = z'
in f x
end
```

If we wish to preserve variable names, we can rename `z'` to `z`.

Note that this version of the β_v suffices to show the correctness of the transformations done to hoist constant expressions. Thus, showing the correctness of hoisting constant expressions is even easier than showing the correctness of CSE.

6.3.3 Asymptotic complexity

The running time of Algorithm 8 depends on what kind of expressions you hoist. If you hoist all kinds of expressions, the running time of Algorithm 8 is $O(N^2)$ (I conjecture below that this can be improved with a different algorithm). If you hoist only expressions and constructors that do *not* contain declarations within them, the running time is $O(N)$.

The algorithm can compute the effects of all expressions in one pre-pass over the program in $O(N)$ time. It can look up the effect of an expression in average constant time, if you implement the map from variables to effects as a hash table.

You can implement the sets of constant variables and constant constructor variables as hash tables, so operations on them can be done in average constant time.

If you can hoist all expression and constructors, Algorithm 8 takes $O(N^2)$ time. There are N syntax nodes, and it takes $O(N)$ time to process `let`-bound expressions and `let`-bound constructors. For `let`-bound expressions, first the algorithm computes the set F of free variables. Because N bounds the size of an expression, this produces a set of size $O(N)$ and takes $O(N)$ time, assuming that you implement F as a hash table. Next, the algorithm must check whether all the free variables are constant. Because the size of F is $O(N)$ and looking up each free variable in C takes average constant time, this takes $O(N)$ time. Finally, the algorithm must look up the effect of the expression, which takes average constant time. Thus, the total time to process a `let`-bound expression is $O(N)$. Similarly, it takes $O(N)$ time to process a `let`-bound constructor.

If you can hoist only expressions and constructors that do not contain nested declarations, then Algorithm 8 takes $O(N)$ time. Let the size of each expression or constructor be s_i. It takes $O(s_i)$ time to process each expression or constructor, because s_i bounds the size of the set of free variables for an expression or constructor. Now, by definition, $\Sigma s_i < N$, where N is the size of the program. Thus, $\Sigma(O(s_i)) = O(N)$.

I conjecture that we can reduce the time for hoisting any kind of expression from $O(N^2)$ time to $O(N)$ time by improving the computation of whether an expression or constructor is constant. Algorithm 8 does this computation by checking whether the free variables of each expression or constructor are constant, which results in an $O(N^2)$ running time. We can improve upon this by doing a single, memoized bottom-up pass over the program. The pass must carefully traverse nodes in the proper order. For example, for a `let`-bound expression, it needs to process the expression bound by the `let` before it processes the body of the `let`. For a λ-expression or a Λ-expression, it needs to mark the variables bound by the expression as non-constant before it processes the body of the expression.

6.4 Invariant removal

An expression is invariant in a loop if the expression evaluates to the same value on every iteration of the loop. Similarly, an expression is invariant in a recursive function if the expression evaluates to the same value every time the body of the recursive function is evaluated. For example, in the trivial SML function

```
fun r nil = nil
  | r (h ::t) = ((a,a),h) :: r t
```

the expression `(a,a)` is invariant.

Invariant expressions arise naturally during compilation from array address computations for multi-dimensional arrays. In type-safe languages, these computations typically are not exposed at the user level, so the compiler must optimize them. For example, consider a simplified version of the dot product function from integer matrix multiply where the 2-d array address computations are made explicit:

```
fix dot=
    λcnt,sum.
        let val test = lti(cnt,bound)
        in  Switch_tag test
            of 1 => let val v1 = sub1(a1,j*columns_a1+cnt)
                        val v2 = sub1(a2,cnt*columns_a2+k)
                        val sum' = c + v1 * v2
                    in dot (cnt+1,sum')
                    end
             |  0 => sum
        end
```

The `dot` function computes the product of some row j of matrix `a1` with some column k of matrix `a2`. Because the variable j does not change during this function, the expression `j*columns_a1` is invariant.

Normally, you cannot move multiplications upward in scope in SML because they may overflow. However, the 2-d array address computations, which are introduced by the compiler, use arithmetic that ignores overflow. Array bounds checking ensures that ignoring overflow is correct in this situation. Let r be the result of a 2-d address computation that is used to index a 1-dimensional array. if array bounds checking succeeds, then the 2-d array indices are within bounds and r must be smaller than the size of the 1-d array. This implies that no overflow occurred when computing r. If array bounds checking fails, then any overflow that occurs when computing r can be ignored because r is never used.

Thus the array-address multiplications are pure, and the expression `j*columns_a1` can be moved out of the loop.

6.4.1 An algorithm for invariant removal

The general idea behind invariant removal is first to estimate the number of times each point in the program is executed, and then to move expressions that may be executed frequently to points where they may be executed less frequently. I have divided the algorithm for invariant removal into into three phases. The first phase estimates how frequently each program point is executed. It computes the the *maximal recursive-function nesting depth* of each program point and assumes that program points with higher maximal nesting depths are executed more frequently. Recursive-function nesting depths are analogous to loop-nesting depths in conventional languages like C or Pascal. The second phase decides which expressions to move and where to move them. It bases this decision on the estimates of program point execution frequencies computed by the first phase. The third phase is a transformation phase that actually moves expressions.

Recursive-function nesting depth is a *dynamic* estimate of how deeply nested a function is within recursive functions. For example, consider the following SML program:

```
fun f x = if x>0 then x+f(x-1) else 0
fun g y = if y>0 then f x + g(y-1) else 0
val a = g 10
```

The recursive-function nesting depth of `g` is 1; the recursive-function nesting depth of `f` is 2. This example also illustrates that recursive-function nesting depth is different from lexical nesting depth.

I use recursive-function nesting depths to construct a profile at compile time of where a program may be spending its time. In informal measurements, which I do not present in this thesis, I have found that profiles constructed from recursive-function nesting depths are very good predictors of where programs actually spend their time.

To compute recursive-function nesting depth, we analyze the call graph [76] of a program. A call graph tells for each function in a program what functions it may call. A call graph is a directed graph G whose nodes are functions. The edges represent potential function calls: there is an edge from a function f to a function g if f may call g. Calls from functions lexically nested within f are not considered to be calls from f. If h is nested within f and it calls j, then there is an edge from h to j but not from f to j.

Here is an example that illustrates this point:

```
fun f x = if x>0 then x+f(x-1) else
                      let fun h () = j()
                      in h ()
                      end
fun g y = if y>0 then f x + g(y-1) else 0
val a = g 10
```

Figure 6.8 shows the call graph for this program fragment.

In languages with higher-order functions, the call graphs of programs may need to be approximated. Consider the following definition of the `map` function:

```
fun map f =
  let fun loop nil = nil
        | loop (h::t) = f h :: loop t
  in loop
  end
```

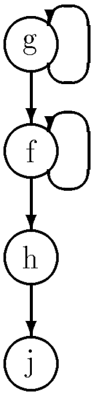

Figure 6.8: A call graph for a program fragment with a nested function. The function h is nested within f, but the call from h to j does not result in an edge from f to j.

The functions called by `map` are not syntactically apparent and the set of possible functions that may be bound to f must be approximated. This problem has been studied extensively and numerous analyses have been proposed for constructing approximations of call graphs and control-flow graphs [36, 39, 44, 76, 82, 83, 95].

It is not clear, however, that these analyses are useful or needed. First, none of the previous works demonstrated that their analyses are *useful* in an actual optimizing compiler, or that the approximations they chose improve existing optimizations in practice. Second, inlining can eliminate many higher-order functions, reducing the importance of approximating the control-flow graph.

Because of this, I chose not to use these analyses for approximating call graphs. Instead, I rely on inlining to eliminate higher-order functions and use just the *known fragment* of the call graph. The idea is to see if a simple approach yields improvements, and leave the use of more sophisticated analyses for future work. To construct the *known fragment* of the call graph, consider only known function applications, and ignore all other applications. A *known function application* is an application of a variable f where f is bound to a function definition. For example, in the expression

```
fun inc x = x+1
fun compose f g x = f (g x)
val a = inc 5
```

the application `inc 5` is a known function application and the application of `g x` is not a known function application.

Figure 6.9 gives Algorithm 9, which calculates the known fragment of the call graph. The graph is represented as a set of edges stored in the set G. The set of variables bound to functions is stored in the set F. I have defined the algorithm as a set of mutually-recursive functions that traverse the program from top to bottom. I omit the functions for traversing constructors, because these are similar to those for traversing expressions and declarations. Each function has two arguments: a program term, such as an expression or declaration, and g, the name of the function whose body it is processing. When the algorithm encounters a function definition, it adds the variable to which the function is bound to the set F. At an application $v(...)$, it checks to see whether v is in F. If so, it adds an edge in G from g (the function that it is processing) to v.

Algorithm 9 takes $O(N)$ time, where N is the size of the program syntax tree. In the following discussion (and throughout the remainder of this section), I will assume that variables are integers between 1 and N. In practice, variables are actually unique arbitrary integers, but it is straightforward to use a hash table to map them to integers between 1 and N in average constant time.

The running time is $O(N)$ because the algorithm traverses each node in the syntax tree and does a constant amount of work at each node. In most cases, it simply recurses. At function definitions, it also adds variables to the set F. This takes constant time if F is implemented as a boolean array. At function applications, the algorithm takes constant time to check for membership in F. If the set G is implemented as an array of N hash tables, where the hash table for a variable f represents the set of variables g such that there is an edge from f to g, then adding an edge $f \rightarrow g$ takes constant time: index the array by f to find the hash table for f in constant time and then insert g into the hash table in average constant time.

Given the call graph, mutually-recursive functions can be found by first computing the strongly connected components of the graph. A strongly-connected component of a graph is a set of nodes such that every node in the component is reachable by a path with 0 or more edges from every other node in the component. In other words, for components with two or more nodes, there is a a cyclic path (loop) connecting all nodes in the component.

Algorithms for identifying maximal strongly-connected components are well-known [80]. It takes $O(M)$ time to find strongly-connected components, where M is the number of vertices and edges in the graph.

The following definition uses strongly-connected components to identify recursive functions. Note that a component with a single node must be treated specially because a single node is by definition always a strongly-connected component: there is a path of length 0 from the node to itself.

Definition 2 (Recursive function) *Given the strongly-connected components of a call graph G, a function $f \in G$ is* recursive *if*

- *it is a member of a strongly-connected component with two or more nodes*
- *or it is in a strongly-connected component with just one node, and there is an edge from the node to itself in the call graph.*

Definition 3 (Mutually-recursive functions) *A set of functions F is* mutually recursive *if all the functions are recursive and they are all members of the same strongly-connected component.*

$$
\begin{array}{lcl}
F & = & \texttt{ref}\ \emptyset \\[1em]
G & = & \texttt{ref}\ \emptyset \\[1em]
\mathrm{cg}_{\mathrm{f}}(f,g) & = & \text{case } f \text{ of} \\
 & & \lceil \lambda(x_1 \ldots x_n).d \rceil \ : \mathrm{cg}_{\mathrm{d}}(d,g) \\[1em]
\mathrm{cg}_{\mathrm{tf}}(tf,g) & = & \text{case } tf \text{ of} \\
 & & \lceil \Lambda(t_1 \ldots t_n).d \rceil \ : \mathrm{cg}_{\mathrm{d}}(d,g) \\[1em]
\mathrm{cg}_{\mathrm{d}}(d,g) & = & \text{case } d \text{ of} \\
 & & \lceil v : \sigma \rceil : () \\
 & & \lceil \texttt{let}\ x : \sigma = e\ \texttt{in}\ d' \rceil \ : (\mathrm{cg}_{\mathrm{e}}(e,g); \mathrm{cg}_{\mathrm{d}}(d',g)) \\
 & & \lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil \ : (\mathrm{cg}_{\mathrm{cb}}(cb,g); \mathrm{cg}_{\mathrm{d}}(d',g)) \\
 & & \lceil \texttt{let fix}\ x_1 : \sigma_1 = f_1 \ldots\ \texttt{and}\ x_n : \sigma_n = f_n\ \texttt{in}\ d' \rceil \ : \\
 & & \quad (F := !F \cup \{x_i | 1 \leq i \leq n\}; \forall i.\mathrm{cg}_{\mathrm{f}}(f_i, x_i); \mathrm{cg}_{\mathrm{d}}(d',g)) \\
 & & \lceil \texttt{let fixtype}\ x_1 : \sigma_1 = tf_1 \ldots\ \texttt{and}\ x_n : \sigma_n = tf_n\ \texttt{in}\ d' \rceil \ : \\
 & & \quad (F := !F \cup \{x_i | 1 \leq i \leq n\}; \forall i.\mathrm{cg}_{\mathrm{tf}}(tf_i, x_i); \mathrm{cg}_{\mathrm{d}}(d',)) \\
 & & \lceil \texttt{raise}\ (v,\sigma) \rceil \ : () \\
 & & \lceil \texttt{let}\ v : \sigma = d_1\ \texttt{handle}\ f\ \texttt{in}\ d_2 \rceil \ : \\
 & & \quad (\mathrm{cg}_{\mathrm{d}}(d_1,g); \mathrm{cg}_{\mathrm{f}}(f,g); \mathrm{cg}_{\mathrm{d}}(d_2,g)) \\[1em]
\mathrm{cg}_{\mathrm{e}}(e,g) & = & \text{case } e \text{ of} \\
 & & \lceil \texttt{switch}\ (st)\, v\ \texttt{of}\ (i : f_i)^*\ \texttt{default:}\ \ od \rceil \ : \\
 & & \quad (\mathrm{cg}_{\mathrm{f}}(f_i,g); \mathrm{cg}_{\mathrm{d}}(od,g)) \\
 & & \lceil \texttt{typecase}\ cv\ \texttt{of}\ [t.\sigma] ta_i^*\ \texttt{rectype}\ : otf\ \texttt{default} : od \rceil \ : \\
 & & \quad (\mathrm{cg}_{\mathrm{ta}}(ta_i,g); \mathrm{cg}_{\mathrm{tf}}(otf,g); \mathrm{cg}_{\mathrm{d}}(od,g)) \\
 & & \lceil \texttt{tlistcase}\ cv\ \texttt{of}\ [t.\sigma]\ \texttt{nil}\ : d\ \texttt{cons} : tf \rceil \ : \\
 & & \quad (\mathrm{cg}_{\mathrm{d}}(d,g); \mathrm{cg}_{\mathrm{tf}}(tf,g)) \\
 & & \lceil v(v_1 \ldots v_n) \rceil : \\
 & & \quad \text{if } v \in F \text{ then } G := G \cup \{(g,v)\} \text{ else } () \\
 & & \lceil v[cv_1 \ldots cv_n) \rceil : \\
 & & \quad \text{if } v \in F \text{ then } G := G \cup \{(g,v)\} \text{ else } () \\
 & & \text{otherwise} : () \\[1em]
\mathrm{cg}_{\mathrm{ta}}(a,g) & = & \text{case } a \text{ of} \\
 & & \lceil c_0 : d \rceil : \mathrm{cg}_{\mathrm{d}}(d,g) \\
 & & \lceil c_1 : tf \rceil : \mathrm{cg}_{\mathrm{tf}}(tf,g)
\end{array}
$$

Figure 6.9: Algorithm 9: calculate the known fragment of the call graph

Definition 4 (Recursive-function nesting) *A function f is nested within a set of mutually recursive functions C if f is reachable from some node in C.*

Based on this notion of nesting, we can define the notion of recursive-function nesting depth: how deeply a function is nested within recursive functions. For example, consider the program

```
fun inc i = i+1
fun g i = if i<10 then g(inc i) else g ()
fun f j = if i<10 then (g i; f(j+1)) else ()
```

Here the function `f` has a nesting depth of 1, because `f` is not nested within any other function. Because `g` is nested with `f`, it has a nesting depth of 2. Finally, because `inc` is nested within `g`, it also has a nesting depth of 2.

A function actually has multiple nesting depths, because it can be nested within multiple recursive functions. Consider the slightly different program:

```
fun inc i = i+1
fun g i = if i<10 then g(inc i) else g ()
fun f j = if i<10 then (g i; f(inc j)) else ()
```

where `inc` is called from by `f` and `j`. Here the function `inc` has nesting depths 1 and 2.

Definition 5 (Recursive-function nesting depths) *The set of recursive-function nesting depths (ND) of some function f is the smallest set $ND(f)$ such that*

- $0 \in ND(f)$
- *If f is not recursive, then*

$$\forall g \in pred(f).ND(f) \supseteq ND(g)$$

- *If f is recursive, then*

  Let C be the strongly-connected component of which f is a member. Let $P = \cup\{pred(g) | g \in C\} - C$.

$$\forall g \in P.ND(f) \supseteq \{x+1 | x \in ND(g)\}$$

The nesting depths of a non-recursive function are simply the nesting depths of its predecessors in the call graph. The definition of nesting depths for a recursive function f is more complicated, because it is phrased in terms of the component of which the function is a member. First, the predecessors of the component are computed. The predecessors P of a component C include the predecessors of each node in C, but exclude nodes in C. Then, one is added to the nesting depths for P.

Because we want to generate excellent code for inner loops and deeply-nested loops, we will choose to use the maximal nesting depth of a function as the estimate of how frequently it is executed.

Definition 6 (Maximal nesting depth) *The maximal nesting depth (MND) of a function f is MND(f) = $\max$(ND(f))*

The following linear-time algorithm computes the MND for every function in a call graph G. The algorithm takes $O(M)$ time, where M is the number of edges and vertices in the call graph. Because M is bounded by program size, the algorithm also takes $O(N)$, where N is program size. (Note that this is a subtle point: M is bounded by N only because we are using the known fragment of the call graph. If we were not using the known fragment of the call graph, M would be bounded only by N^2.)

Algorithm 10 *Compute maximal nesting depths of functions*

1. Compute the strongly-connected components of the call graph. It is straightforward to return the list L of components so that it is *topologically sorted*. A list of components L is topologically sorted if a component C precedes any components $D \in L$ that it calls (that is, depends on).

   Each component can be represented as a list.

2. Assign each component a number and create a mapping COMP from function names to component numbers. This mapping can be implemented as an array of integers, and can be created in $O(M)$ time.

3. Also create a mapping PRED that maps each function name to a list of its predecessors. PRED is an inverted version of the call graph. The mapping can be implemented as an array of lists, and can be created in $O(M)$ time by traversing all the edges in the call graph.

4. Create a mapping MND from components to maximal nesting depths. Initialize the mapping to 0. This mapping can be implemented as an array and can be created in $O(M)$ time.

5. Traverse the list L of strongly-connected components. For each component C:

   - If C is a set of mutually-recursive functions, traverse each node in C to compute max(MND(COMP($pred(C)$))). Initialize a variable `m` to 0. For each node, look up the set of predecessors using PRED. For each predecessor p, use COMP to find the component C' that the predecessor is in. Next, use MND to find the nesting depth d of C'. If $d > m$, then set m to d.

     After traversing every predecessor of every node in C, set MND(C) $= m + 1$.

   - Otherwise C is a singleton set $\{f\}$. Find the MND of each predecessor of f, and take the maximum m. Set MND(C) $= m$.

   L can be processed in $O(M)$ time. The number of predecessors to be processed is the same as the number of edges in the call graph. Each predecessor can be processed in constant time. The look ups in COMP and MND can both be done in constant time, and so can the comparison against the current maximum.

6. Finally, create a map M that computes the MND of each function. For each function f, set $M(f) = MND(COMP(f))$. This mapping can be created in $O(M)$ time.

Because each individual step of the algorithm has a running time of $O(M)$, the algorithm has an $O(M)$ running time.

6.4.2 Deciding which expressions and constructors to move

The algorithm for deciding which expressions and constructors to move is divided into two phases. The first phase traverses the program from top to bottom and records for each bound variable:

- its lexical nesting depth
- its maximal recursive-function nesting depth

The lexical-nesting depth of a variable v is simply the number of variable bindings within which v's binding is nested. The recursive-function nesting depth of a variable v is the recursive-function nesting depth of the nearest function definition f enclosing v's binding.

With this information, it is quite easy to decide which expressions and constructors to move and where to move them. I will discuss only deciding how to move expressions, because the treatment of constructors is similar. Given an expression e with a recursive-function nesting depth d [3], let the set of variables it uses be U. The set of variables it uses is simply its free variables. The expression e is invariant and can be moved if

- $\forall v \in U,$ recursive-function nesting depth$(v) < d$
- e is pure according to the effects analysis presented in Section 5.3.1

and a side-condition related to preserving scoping is met:

- $\forall v \in U$, v is not the name of a recursive function within whose body e is nested.

The expression should be moved to the point immediately after all its free variables have been bound, because this point is estimated to be executed less frequently then the current location of the expression.

The side-condition is needed when moving expressions whose free variables include names of recursive functions. It is not possible to move the invariant expression "after" the binding of such a function name, because that places them outside the scope of the `fix` or `fixtype`. For example, consider the simple function f:

```
fix f = λx.if x>0 then NONE else SOME f
```

The expression `SOME f` cannot be moved after the binding of f because that places the expression outside of the scope of f:

[3] More properly, which is bound to a variable v with a nesting depth d.

```
fix f = λx.if x>0 then NONE else a
val a = SOME f
```

For constructors, the implementation moves only constructors bound at the level of expressions. It does not move individual constructors bound at the level of constructors because that would complicate the implementation even further. It is straightforward to extend the implementation to do so, and it would be interesting to measure its benefit.

Figure 6.10 presents Algorithm 11, which implements the first phase of deciding which expressions and constructors to move. It produces a mapping M from bound variables to their properties. Each function takes three arguments: a term, the lexical-nesting depth of the term, and the recursive-function nesting depth of the term.

For constructor variables, Algorithm 11 computes information only for constructor variables bound at the level of expressions. The function getv(cb) retrieves the variable bound by a constructor binding.

Figure 6.11 and Figure 6.12 present Algorithm 12, which actually decides which expressions and constructors to move. It produces a set MV and a mapping A. MV tells which `let`-bound expressions or constructors to move, while A tells how to move them. A maps a variable x to the sequence of expressions and constructors S that should be moved immediately after the binding of x. Each expression `let` $y : \sigma = e$ to be moved is represented as a triple (y, σ, e) and each constructor binding cb is represented as itself.

The algorithm directly implements the conditions described earlier. The set F is the set of recursive functions whose bodies are being processed; it is used to check for the side-condition related to preserving scoping for recursive functions. When the algorithm encounters an expression, it checks that none of the free variables of the expression is in the set F and that the recursive-function nesting depth of each free variable is less than the recursive-function nesting depth of the expression. If so, and if the expression is pure, then it should be moved. To decide where to move the expression, the algorithm finds the free variable dst with the greatest lexical-nesting depth, that is, the last free variable of the expression to be bound. The algorithm updates the mapping A to record that the expression should be moved after the binding of variable dst.

The treatment of constructors is simpler than the treatment of expressions. First, the algorithm does not need to check that constructors are pure. Second, the algorithm does not need to check whether any of the free variables of the constructor are in the set F, because constructors cannot refer to recursive functions. Recall that getv(cb) retrieves the variable bound by a constructor binding.

Finally, Figure 6.13 presents Algorithm 13, which moves expressions and constructors. It traverses the program syntax tree from top to bottom. Whenever a variable is bound, it consults the mapping A to find which expressions and constructors to move after the variable binding. Note that it moves expressions and constructors after the point at which the variable was *originally* bound, and not the new point at which the variable is bound. The auxilary function `insert(S,d)` takes a sequence of expressions and constructors $S = \{(x_1, \sigma_1, e_1), cb_2, \ldots (x_n, \sigma_n, e_n)\}$ to move and a declaration d to insert the expressions and constructors before and creates a declaration

$$
\begin{array}{ll}
M & = \texttt{ref}\ \emptyset \\
\mathrm{p_f}(f, ld, rd) & = \text{case } f \text{ of} \\
& \quad \lceil \lambda(x_1 \ldots x_n).d \rceil \ : \\
& \qquad (\forall i : 1 \leq i \leq n. M := !M + \{v_i \mapsto \{\text{lex} = ld + i - 1, \text{rec} = rd\}\}; \\
& \qquad \mathrm{p_d}(d, ld + n, rd)) \\
\\
\mathrm{p_{tf}}(tf, ld, rd) & = \text{case } tf \text{ of} \\
& \quad \lceil \Lambda(t_1 \ldots t_n).d \rceil \ : \\
& \qquad (\forall i : 1 \leq i \leq n. M := !M + \{t_i \mapsto \{\text{lex} = ld + i - 1, \text{rec} = rd\}\}; \\
& \qquad \mathrm{p_d}(d, ld + n, rd)) \\
\\
\mathrm{p_d}(d, ld, rd) & = \text{case } d \text{ of} \\
& \quad \lceil \texttt{let}\ x : \sigma = e\ \texttt{in}\ d' \rceil \ : \\
& \qquad (M := !M + \{x \mapsto \{\text{lex} = ld, \text{rec} = rd\}\}; \mathrm{p_e}(e, ld, rd); \mathrm{p_d}(d', ld + 1, g)) \\
& \quad \lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil \ : \\
& \qquad (M := !M + \{\text{getv(cb)} \mapsto \{\text{lex} = ld, \text{rec} = rd\}\}; \mathrm{p_d}(d', ld + 1, rd)) \\
& \quad \lceil \texttt{let fix}\ x_1 : \sigma_1 = f_1 \ldots\ \texttt{and}\ x_n : \sigma_n = f_n\ \texttt{in}\ d' \rceil \ : \\
& \qquad (\forall i : 1 \leq i \leq n. M := !M + \{x_i \mapsto \{\text{lex} = ld, \text{rec} = rd\}\}; \\
& \qquad \forall i : 1 \leq i \leq n. \mathrm{p_f}(f_i, ld + 1, MND(x_i)); \\
& \qquad \mathrm{p_d}(d', ld + 1, rd)) \\
& \quad \lceil \texttt{let fixtype}\ x_1 : \sigma_1 = tf_1 \ldots\ \texttt{and}\ x_n : \sigma_n = tf_n\ \texttt{in}\ d' \rceil \ : \\
& \qquad (\forall i : 1 \leq i \leq n. M := !M + \{x_i \mapsto \{\text{lex} = ld, \text{rec} = rd\}\}; \\
& \qquad \forall i : 1 \leq i \leq n. \mathrm{p_{tf}}(tf_i, ld + 1, MND(x_i)); \\
& \qquad \mathrm{p_d}(d', ld + 1, rd)) \\
& \quad \lceil \texttt{let}\ x : \sigma = d_1\ \texttt{handle}\ f\ \texttt{in}\ d_2 \rceil \ : \\
& \qquad (M := !M + \{x \mapsto \{\text{lex} = ld, \text{rec} = rd\}\}; \mathrm{p_d}(d_1, ld, rd); \\
& \qquad \mathrm{p_f}(f, ld, rd); \mathrm{p_d}(d_2, ld + 1, rd)) \\
& \quad \text{otherwise} : () \\
\\
\mathrm{p_e}(e, ld, rd) & = \text{case } e \text{ of} \\
& \quad \lceil \texttt{switch}\ (st)\, v\ \texttt{of}\ (i : f_i)^*\ \texttt{default:}\ \ od \rceil \ : \\
& \qquad (\mathrm{p_f}(f_i, ld, rd); \mathrm{p_d}(od, ld, rd)) \\
& \quad \lceil \texttt{typecase}\ cv\ \texttt{of}\ [t.\sigma] ta_i^*\ \texttt{rectype}\ : otf\ \texttt{default} : od \rceil \ : \\
& \qquad (\mathrm{p_{ta}}(ta_i, ld, rd); \mathrm{p_{tf}}(tf, ld, rd); \mathrm{p_d}(od, ld, rd)) \\
& \quad \lceil \texttt{tlistcase}\ cv\ \texttt{of}\ [t.\sigma]\ \texttt{nil}\ : d\ \texttt{cons} : tf \rceil \ : \\
& \qquad (\mathrm{p_d}(d, ld, rd); \mathrm{p_{tf}}(tf, ld, rd)) \\
& \quad \text{otherwise} : () \\
\\
\mathrm{p_{ta}}(a, ld, rd) & = \text{case } a \text{ of} \\
& \quad \lceil c_0 : d \rceil : \mathrm{p_d}(d, ld, rd) \\
& \quad \lceil c_1 : tf \rceil : \mathrm{p_{tf}}(tf, ld, rd)
\end{array}
$$

Figure 6.10: Algorithm 11: calculate properties of bound variables

$$
\begin{array}{lll}
A & = & \texttt{ref}\ \emptyset \\[1em]
MV & = & \texttt{ref}\ \emptyset \\[1em]
F & = & \texttt{ref}\ \emptyset \\[1em]
\mathrm{mv}_{\mathrm{f}}(f) & = & \text{case } f \text{ of} \\
 & & \lceil \lambda(x_1 \ldots x_n).d \rceil \ : \mathrm{mv}_{\mathrm{d}}(d) \\[1em]
\mathrm{mv}_{\mathrm{tf}}(tf) & = & \text{case } tf \text{ of} \\
 & & \lceil \Lambda(t_1 \ldots t_n).d \rceil \ : \mathrm{mv}_{\mathrm{d}}(d) \\[1em]
\mathrm{mv}_{\mathrm{d}}(d) & = & \\
\text{case } d \text{ of} & &
\end{array}
$$

$$
\begin{array}{l}
\lceil \texttt{let}\ x : \sigma = e\ \texttt{in}\ d' \rceil \\
\quad (\mathrm{mv}_{\mathrm{e}}(e); \\
\quad \text{let } U = \mathrm{fv}(e) \text{ and } d = \text{rec. depth}(x) \\
\quad \text{in if pure}(e) \text{ and } \forall y \in U.\text{rec. depth}(y) < d \text{ and } y \notin F \\
\qquad \text{then let } dst = y \text{ where } y \in U \text{ with the max. lex. nesting depth} \\
\qquad\qquad \text{in } (MV := MV \cup \{x\}; A := A + \{dst \mapsto A(dst) \cup \{(x, \sigma, e)\}\}) \\
\qquad\qquad \text{end} \\
\qquad \text{else } (); \\
\quad \mathrm{mv}_{\mathrm{d}}(d')) \\
\quad \text{end} \\
\lceil \texttt{let}\ cb\ \texttt{in}\ d' \rceil \ : \\
\quad (\text{let } U = \mathrm{fv}(cb) \text{ and } d = \text{rec. depth}(\mathrm{getv}(cb)) \\
\quad \text{in if } \forall y \in U.\text{rec. depth}(y) < d \\
\qquad \text{then let } dst = y \text{ where } y \in U \text{ with the max. lex. nesting depth} \\
\qquad\qquad \text{in } (MV := MV \cup \{\mathrm{getv}(x)\}; A := A + \{dst \mapsto A(dst) \cup \{cb\}\}) \\
\qquad\qquad \text{end} \\
\qquad \text{else } (); \\
\quad \mathrm{mv}_{\mathrm{d}}(d')) \\
\quad \text{end} \\
\lceil \texttt{let fix}\ x_1 : \sigma_1 = f_1 \ldots\ \texttt{and}\ x_n : \sigma_n = f_n\ \texttt{in}\ d' \rceil \ : \\
\quad (F := F \cup \{x_i\}; \forall i : 1 \le i \le n.\mathrm{mv}_{\mathrm{f}}(f_i); F := F - \{x_i\}; \mathrm{mv}_{\mathrm{d}}(d')) \\
\lceil \texttt{let fixtype}\ x_1 : \sigma_1 = tf_1 \ldots\ \texttt{and}\ x_n : \sigma_n = tf_n\ \texttt{in}\ d' \rceil \ : \\
\quad (F := F \cup \{x_i\}; \forall i : 1 \le i \le n.\mathrm{mv}_{\mathrm{tf}}(f_i); F := F - \{x_i\}; \mathrm{mv}_{\mathrm{d}}(d')) \\
\lceil \texttt{let}\ v : \sigma = d_1\ \texttt{handle}\ f\ \texttt{in}\ d_2 \rceil \\
\quad (\mathrm{mv}_{\mathrm{d}}(d_1); \mathrm{mv}_{\mathrm{f}}(f); \mathrm{mv}_{\mathrm{d}}(d_2)) \\
\text{otherwise} : ()
\end{array}
$$

Figure 6.11: Algorithm 12, part 1: decide which expressions to move.

$$
\begin{array}{lcl}
\mathrm{mv}_{\mathrm{e}}(e) & = & \text{case } e \text{ of} \\
 & & \lceil \texttt{switch } (st)\, v \texttt{ of } (i : f_i)^* \texttt{ default: } od \rceil : \\
 & & \quad (\mathrm{mv}_{\mathrm{f}}(f_i); \mathrm{mv}_{\mathrm{d}}(od)) \\
 & & \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma] ta_i^* \texttt{ rectype } : otf \texttt{ default} : od \rceil : \\
 & & \quad (\mathrm{mv}_{\mathrm{ta}}(ta_i); \mathrm{mv}_{\mathrm{tf}}(otf); \mathrm{mv}_{\mathrm{d}}(od)) \\
 & & \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : d \texttt{ cons} : tf \rceil : \\
 & & \quad (\mathrm{mv}_{\mathrm{d}}(d); \mathrm{mv}_{\mathrm{tf}}(tf)) \\
 & & \text{otherwise} : () \\
 & & \\
\mathrm{mv}_{\mathrm{ta}}(a) & = & \text{case } a \text{ of} \\
 & & \lceil c_0 : d \rceil : \mathrm{mv}_{\mathrm{d}}(d) \\
 & & \lceil c_1 : tf \rceil : \mathrm{mv}_{\mathrm{tf}}(tf)
\end{array}
$$

Figure 6.12: Algorithm 12, continued: decide which expressions to move.

```
let x1 : σ1 = re(e1)
    cb2
    ...
    xn : σn = re(en)
in d
 end
```

When the algorithm processes a `let`-bound expression, it checks whether the variable being bound is in the set MV. If it is, then this occurrence of the expression is deleted because the expression is being moved elsewhere. The algorithm processes constructors bound at the level of expressions in a similar manner.

Algorithms 12 and 13 preserve variable scoping. First, consider the case of an expression that does not contain any expressions nested within it or a constructor. Variable scoping is preserved if all free variables in the expression or constructor are still bound before the occurrence of the expression or constructor. The algorithms broaden the scope of variables, and they do not move any expression or constructor before the original point at which its last free variable was bound, so this is true.

Next, consider the more complicated case of an expression that contains expressions nested within it. The nested expressions may be moved, introducing free variables into the expression, which may also be moved. For example, in the following program:

```
let v = if true
             then let y = (a,a)
                  in y
                  end
             else let z = (1,1)
                   in z
                  end
```

$$
\begin{array}{lll}
\mathrm{r}_{\mathrm{f}}(\lceil \lambda(v_1 \ldots v_n).d\rceil\ ) & = & \texttt{insert}(\cup A(v_i), \mathrm{r}_{\mathrm{d}}(d)) \\[1ex]
\mathrm{r}_{\mathrm{tf}}(\lceil \Lambda(t_1 \ldots t_n).d\rceil\ ) & = & \texttt{insert}(\cup A(t_i), \mathrm{r}_{\mathrm{d}}(d)) \\[1ex]
\mathrm{r}_{\mathrm{d}}(d) & = & \text{case } d \text{ of} \\
 & & \lceil \texttt{let } x : \sigma = e \texttt{ in } d'\rceil\ : \\
 & & \quad \text{if } x \in MV \\
 & & \quad \text{then } \texttt{insert}(A(x), \mathrm{r}_{\mathrm{d}}(d')) \\
 & & \quad \text{else } \lceil \texttt{let } x : \sigma = \mathrm{r}_{\mathrm{e}}(e) \texttt{ in } \texttt{insert}(A(x), \mathrm{r}_{\mathrm{d}}(d'))\rceil \\
 & & \lceil \texttt{let } cb \texttt{ in } d'\rceil\ : \\
 & & \quad \text{if getv}(cb) \in MV \\
 & & \quad \text{then } \texttt{insert}(A(\text{getv(cb)}), \mathrm{r}_{\mathrm{d}}(d')) \\
 & & \quad \text{else } \lceil \texttt{let } cb \texttt{ in } \texttt{insert}(A(x), \mathrm{r}_{\mathrm{d}}(d'))\rceil \\
 & & \lceil \texttt{let fix } x_1 : \sigma_1 = f_1 \ldots \texttt{ and } x_n : \sigma_n = f_n \texttt{ in } d'\rceil\ : \\
 & & \quad \lceil \texttt{let fix } x_1 : \sigma_1 = \mathrm{r}_{\mathrm{f}}(f_1) \ldots \texttt{ and } x_n : \sigma_n = \mathrm{r}_{\mathrm{f}}(f_n) \\
 & & \quad\ \ \texttt{in } \texttt{insert}(\cup A(x_i), \mathrm{r}_{\mathrm{d}}(d'))\rceil \\
 & & \lceil \texttt{let fixtype } x_1 : \sigma_1 = tf_1 \ldots \texttt{ and } x_n : \sigma_n = tf_n \texttt{ in } d'\rceil \\
 & & \quad \lceil \texttt{let fixtype } x_1 : \sigma_1 = \mathrm{r}_{\mathrm{f}}(tf_1) \ldots \texttt{ and } x_n : \sigma_n = \mathrm{r}_{\mathrm{tf}}(tf_n) \\
 & & \quad\ \ \texttt{in } \texttt{insert}(\cup A(x_i), \mathrm{r}_{\mathrm{d}}(d'))\rceil \\
 & & \lceil \texttt{let } x : \sigma = d_1 \texttt{ handle } f \texttt{ in } d_2\rceil\ : \\
 & & \lceil \texttt{let } x : \sigma = \mathrm{r}_{\mathrm{d}}(d_1) \texttt{ handle } \mathrm{r}_{\mathrm{f}}(f) \\
 & & \ \ \texttt{in } \mathrm{r}_{\mathrm{d}}(d_2)\rceil \\
 & & \text{otherwise} : d \\[2ex]
\mathrm{r}_{\mathrm{e}}(e) & = & \text{case } e \text{ of} \\
 & & \lceil \texttt{switch } (st)\, v \texttt{ of } (i : f_i)^* \texttt{ default: } od\rceil\ : \\
 & & \quad \lceil \texttt{switch } (st)\, v \texttt{ of } (i : \mathrm{r}_{\mathrm{f}}(f_i))^* \texttt{ default: } \mathrm{r}_{\mathrm{d}}(od)\rceil \\
 & & \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma]ta_i^* \texttt{ rectype } : otf \texttt{ default} : od\rceil\ : \\
 & & \quad \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma]\mathrm{r}_{\mathrm{tf}}(ta_i)^* \\
 & & \quad\ \ \texttt{rectype } : \mathrm{r}_{\mathrm{tf}}(otf) \\
 & & \quad\ \ \texttt{default} : \mathrm{r}_{\mathrm{d}}(od)\rceil \\
 & & \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : d \texttt{ cons} : tf\rceil\ : \\
 & & \quad \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : \mathrm{r}_{\mathrm{d}}(d) \texttt{ cons} : \mathrm{r}_{\mathrm{tf}}(tf)\rceil \\
 & & \text{otherwise} : e \\[2ex]
\mathrm{r}_{\mathrm{ta}}(a) & = & \text{case } a \text{ of} \\
 & & \lceil c_0 : d\rceil : \mathrm{r}_{\mathrm{d}}(d) \\
 & & \lceil c_1 : tf\rceil : \mathrm{r}_{\mathrm{tf}}(tf)
\end{array}
$$

Figure 6.13: Algorithm 13: move expressions.

the `if`-statement and the expression (a, a) may be moved. Consider the general case of an expression e_1 with an expression e_2 nested within it, where e_1 and e_2 are both being moved. If $\mathrm{fv}(e_1) \supset \mathrm{fv}(e_2)$, then e_1 will be moved so that is bound after e_2, because there is some free variable of e_1 that is bound after all the free variables of e_2 have been bound. If $\mathrm{fv}(e_1) = \mathrm{fv}(e_2)$, then e_1 and e_2 are being moved after the same variable. Let that variable be x. Because e_1 was traversed before being considered as a candidate for moving, e_2 must precede e_1 in the sequence $A(x)$.

6.4.3 Correctness

We can show the correctness of the transformations done by Algorithm 13 by using the call-by-value λ-calculus, just as we can show the correctness of transformations done by the other code motion optimizations (CSE and hoisting constant expressions).

Like we do for the hoisting of constant expressions, we only need to rely on the call-by-value λ-calculus with a larger syntactic class of values. Specifically, the syntactic class of values must be extended beyond constants, variables, and λ-expressions to include other effect-free expressions, such as creating a record from values.

To remove an invariant expression with this extended version of the β_v rule, we simply do a β expansion. For example, to move `len A` out of the function `loop` in the following program:

```
let fix loop (cnt,sum) =
      if cnt<len A then loop(cnt+1,sum+sub1(A,cnt))
      else sum
in loop
end
```

we simply do a β expansion that binds `len A` to the variable `bound`:

```
let bound = len A
    fix loop (cnt, sum)
      if cnt<bound then loop(cnt+1,sum+sub1(A,cnt))
      else sum
in loop
end
```

6.4.4 Asymptotic complexity

Let N be the size of the program. Algorithm 11, which records lexical nesting depth and recursive-function nesting depth, has a running time of $O(N)$. For each bound variable, the algorithm adds an entry to the map M. This can be done in constant time if M is implemented as a hash table. When a function bound by `fix` or `fixtype` is processed, the algorithm must look up its maximal-recursive function nesting depth, which can also be done in constant time.

Algorithm 12, which decides which expressions to move, has a running time of $O(N)$ or $O(N^2)$, depending on what kinds of expressions are moved. The running time is $O(N)$ if only expressions without declarations nested within them are moved; otherwise it is $O(N^2)$ (it may be possible to reduce this running time by using an improved algorithm). The argument is similar to that presented in Section 6.3.3 for hoisting constant expressions. The important step is processing `let`-bound expressions, which takes time proportional to the free variables of the expression. In turn, the number of free variables is proportional to the size of the expression.

Let s_i be the size of each expression. If only expressions not containing declarations within are moved, then $O(\Sigma s_i) = O(N)$, so processing takes $O(N)$ time. Otherwise, s_i is bounded only by N, so processing takes $O(\Sigma s_i) = O(N^2)$ time.

Algorithm 13 has a running time of $O(N)$. At each point in the program where a variable is bound, the algorithm looks up in the map A what expressions have to be moved after the variable. This takes average constant time, if A is implemented as a hash table.

If there are any expressions that have to be moved, the algorithm inserts them into the program. Because at most N expressions can be moved, the total time to insert expressions in $O(N)$.

The total running time for invariant removal is $O(N)$ if only expressions not containing declarations nested within them are moved. It is $O(N^2)$ if expressions containing declarations nested within them are moved. Each step of invariant removal takes only $O(N)$ time, except for Algorithm 12 which takes $O(N)$ or $O(N^2)$ time. Thus, Algorithm 12 dominates the running time of invariant removal.

6.4.5 Converging in one pass

The algorithm that I just presented for invariant removal requires multiple passes to move all invariant expressions and constructors. For example, given an expression of the form

```
let v = (a,a)
    w = (v,v)
in ...
end
```

where `(a,a)` is invariant, the binding of `v` will be hoisted on the first pass, but the binding of `w` will be hoisted only on a second pass. The problem is that when the algorithm moves `v`, it needs to update the lexical nesting depth and recursive-function nesting depth of v.

The actual implementation that I use in TIL converges in one pass. The implementation combines Algorithm 11 and Algorithm 12 into one pass. When it moves an expression or constructor, it updates the lexical nesting depth and the recursive-function nesting to reflect the new location of the expression or constructor.

Maintaining the mapping A used in Algorithm 12 is subtle. A maps each variable to the sequence of expressions and constructors that should be moved immediately after the binding of that variable. If an expression `let` $x : \sigma = e$ `in` ... is supposed to be moved after some variable y, but y is supposed to be moved after the binding of some variable z, then we add (x, σ, e) to the sequence of expressions for z, not y. To preserve scoping, it is important to keep expressions and constructors in the sequence in the same order in which they are added to the sequence.

To illustrate this last point, consider the previous example. Suppose that v is being moved after the variable a, that is $A(a) = [(v, \texttt{int}, (\texttt{a}, \texttt{a}))]$. When w is added to $A(a)$, $A(a)$ must equal $[(v, \texttt{int}, (\texttt{a}, \texttt{a})), (w, \texttt{int}, (\texttt{v}, \texttt{v}))]$. This ensures that when $A(a)$ is appended after the binding of a, the scoping constraints for v and w are preserved.

6.5 Eliminating redundant comparisons

The reader may have noticed that eliminating redundant switches misses some obvious cases. For example, given an SML program of the form,

```
let val x = if a>b then
                 if a>=b then 0 else 1
            else 1
in ...
end
```

redundant switch elimination will not eliminate the inner `if`-expression. Also, it does not eliminate all the cost of bounds checking even for the simple example of a loop summing the contents of the array; the comparison that checks that the loop count is greater than 0 remains.

These problems arise because redundant switch elimination has no knowledge of the semantics of arithmetic comparisons. It only eliminates switches on the same variable. The B-form version of the previous example is:

```
let t1 = a>b
    x = switch (enum) t1
        0 : λ().1
        1 : let t2 = a>=b
                t3 = switch (enum) t2
                     0 : λ().1
                     1 : λ().0
            in t3
in ...
```

CSE cannot eliminate `t2`, so redundant switch elimination cannot eliminate the switch on `t1`.

Redundant comparison elimination is a generalization of redundant switch elimination that eliminates comparisons. I divided the algorithm into two phases: an analysis phase and a transformation phase. The analysis phase determines for each point in a program a set of propositions about relations between integer variables that are true at that point.

Definition 7 *A proposition has the form*

$$bv_1 = bv_2, bv_1 > bv_2, bv_1 \neq bv_2, \;\; or\; bv_1 >= bv_2$$

where bv_i ranges over variables and integers.

$$
\begin{aligned}
a = b &\Rightarrow b = a \\
a = b &\Rightarrow b \geq a \\
a \neq b &\Rightarrow b \neq a \\
a > b &\Rightarrow a \geq b \\
a > b &\Rightarrow a \neq b \\
a = b \text{ and } b \text{ op } c &\Rightarrow a \text{ op } c \\
a = b \text{ and } c \text{ op } b &\Rightarrow c \text{ op } a \\
a \neq b \text{ and } a \geq b &\Rightarrow a > b \\
a > b \text{ and } b > c &\Rightarrow a > c \\
a \geq b \text{ and } b \geq c &\Rightarrow a \geq c \\
a = b \text{ and } b = c &\Rightarrow a = c
\end{aligned}
$$

Figure 6.14: Axioms used by redundant comparison elimination

There are numerous analyses possible that can produce results in such a form. I use two simple analyses. The first analysis scans a program top-down, examining switches. If a switch is on a boolean variable that is bound to an arithmetic comparison, the analysis infers that the arithmetic comparison is true in the true branch of the switch, and false in the false branch of the switch. In the previous example, when the analysis processes the `switch` on `t2`, it infers that `a` $\geq$ `b` in the true arm of the switch (the 1 branch), and that `a` $\not\geq$ `b` (that is `b` $>$ `a`) in the false branch of the switch (the 0 branch). The second analysis uses a simple abstract interpretation based on the rule-of-signs for arithmetic operations (if $a > 0$ and $b > 0$ then $ab > 0$ and so on).

The transformation phase traverses the program. When it sees a comparison, it converts the comparison to a proposition P. It then looks up the set of propositions S which are known to be true. It checks to see if $P \in S$ or $\neg P \in S$. If this fails, it computes the transitive closure TS of S, according to the axioms given in Figure 6.14. It then checks whether P or $\neg P \in S$. If P is a proposition involving a constant, it checks to see if there is some $P' \in S$ involving a constant which implies P or $\neg P$. For example, if P is $a > 5$, then $a > 7 \in S$ implies the truth of P.

6.5.1 Inferring propositions from `switch`es

The first analysis, Algorithm 14, is presented in Figure 6.15 and Figure 6.16. The result of the algorithm is a map M that maps each program point to a set of propositions that are true at that program point. Each function in the analysis takes three arguments: a term to analyze, an environment env mapping variables of boolean type to propositions that are true if the variable is true, and a set S of propositions that are known to be true. For each `let`-bound variable, the analysis records the set S in the map M. When the analysis processes a `let`-bound comparison expression of the form `let` $x = op(y, z)$ `in` $\ldots$, it infers a proposition P from the comparison and extends env to map x to P. When it processes a `switch` on a variable x, it checks to see if x is in the environment. If so, it adds $env(x)$ to the set S of true propositions when processing the `true` branch of the switch, and adds the negation of $env(x)$ to the set S when processing the `false` branch.

This analysis has an asymptotic complexity of $O(N \log N)$. On processing each `let`-bound expression, it needs to add the current set of propositions to the map M. This can be done in average constant time if M is implemented as a hash table. If the expression is a boolean expression, it also needs to extend the environment mapping variables to propositions. This can be done in $O(\log N)$ if environments are implemented as balanced trees. Thus the time to process a `let`-bound expression is $O(\log N)$.

On processing a `switch`, the analysis needs to check the environment in $O(\log N)$ time to see if the variable is bound to a boolean expression. If so, the analysis needs to add a proposition to the set of propositions when processing the true and false arms of the switch. The set of propositions can be implemented as a balanced tree, with a lexicographic ordering imposed on propositions, and there are at most N propositions in the set, so this can be done in $O(\log N)$ time. Thus, the total time to process a `switch` is $O(\log N)$ time.

Because there are at most N steps in the analysis, the entire analysis takes $O(N \log N)$ time.

6.5.2 Rule-of-signs abstract interpretation

The rule-of-signs abstract interpretation uses a lattice whose elements are $\{\top, +, -, i, \bot\}$, where $\top$ denotes any integer, $+$ denotes the set $\{i|i \geq 0\}$, $-$ denotes the set $\{i|i \leq 0\}$, i denotes integer constants, and $\bot$ denotes absence of information.

Figures 6.17 through Figures 6.19 present the abstract interpreter. Figures 6.17 and 6.18 contain the main body of the interpreter, while Figure 6.19 defines the abstract primitive operations. The definitions of multiplication and division are elided to save space.

The analysis uses two data structures. The first is a map F from function names to their approximated domain and range values, where the approximated domains and ranges are stored in ref cells. F should be initialized so that all escaping functions[4] are mapped to domains of $\top$ values, while all known functions are mapped to domains of $\bot$ values. All ranges should be initialized to $\bot$. The second data structure is an updateable map A from variables to their approximated values. This map A approximates all the possible values of environments. A should be initialized so that all bound variables are mapped $\bot$ values, except for variables bound by the arms of branching expressions. Those variables represent values extracted from program data structures, so they should be initialized to $\top$.

After the data structures have been initialized, the function $\mathrm{a_d}$ should be applied to a program repeatedly until the ranges of F and A do not change. That is,

1. $\mathrm{a_d}$ is applied to the program.

2. If that changes the ranges of F or A, we go back to step (1).

[4] A function escapes if some of its call sites are unknown.

$$
\begin{array}{lcl}
M & = & \text{ref } \emptyset \\
\\
\text{neg}(p) & = & \text{case } p \text{ of} \\
& & \lceil x = y \rceil : \lceil x \neq y \rceil \\
& & \lceil x \neq y \rceil : \lceil x = y \rceil \\
& & \lceil x \geq y \rceil : \lceil y > x \rceil \\
& & \lceil x > y \rceil : \lceil y \geq x \rceil \\
\\
\text{a}_{\text{d}}(d, env, s) & = & \text{case } d \text{ of} \\
& & \lceil \texttt{let } x : \sigma = e \texttt{ in } d' \rceil \; : \\
& & \text{let } env' = \;\; \text{case } e \text{ of} \\
& & \qquad\qquad \lceil \texttt{eqi}(bv_1, bv_2) \rceil : env[x \mapsto \lceil bv_1 = bv_2 \rceil] \\
& & \qquad\qquad \lceil \texttt{lti}(bv_1, bv_2) \rceil : env[x \mapsto \lceil bv_2 > bv_1 \rceil] \\
& & \qquad\qquad \lceil \texttt{gti}(bv_1, bv_2) \rceil : env[x \mapsto \lceil bv_1 > bv_2 \rceil] \\
& & \qquad\qquad \lceil \texttt{ltei}(bv_1, bv_2) \rceil : env[x \mapsto \lceil bv_2 \geq bv_1 \rceil] \\
& & \qquad\qquad \lceil \texttt{gtei}(bv_1, bv_2) \rceil : env[x \mapsto \lceil bv_1 \geq bv_2 \rceil] \\
& & \qquad\qquad \lceil \texttt{switch } (enum)\, x' \texttt{ of } 0 : \texttt{enum } 0, f_1 : \texttt{enum } 1 \texttt{ default: } \;\; \epsilon \rceil \; : \\
& & \qquad\qquad\quad \text{if } x' \in \text{dom}(env) \text{ then } env[x \mapsto env(x')] \text{ else } () \\
& & \qquad\qquad \lceil \texttt{switch } (enum)\, x' \texttt{ of } 0 : \texttt{enum } 1, f_1 : \texttt{enum } 0 \texttt{ default: } \;\; \epsilon \rceil \; : \\
& & \qquad\qquad\quad \text{if } x' \in \text{dom}(env) \text{ then } env[x \mapsto \text{neg}(env(x'))] \text{ else } () \\
& & \text{in } (M \; := \; (!M)[x \mapsto s]; \text{a}_{\text{e}}(e, env, s); \text{a}_{\text{d}}(d', env', s)) \\
& & \text{end} \\
& & \lceil \texttt{let } cb \texttt{ in } d' \rceil \; : \\
& & \quad \text{a}_{\text{d}}(d', env, s) \\
& & \lceil \texttt{let fix } x_1 : \sigma = f_1 \ldots \texttt{ and } x_n : \sigma = f_n \texttt{ in } d' \rceil \; : \\
& & \quad (\forall i . \text{a}_{\text{f}}(f_i, env, s); \text{a}_{\text{d}}(d', env, s)) \\
& & \lceil \texttt{let fixtype } x_1 : \sigma = f_1 \ldots \texttt{ and } x_n : \sigma = f_n \texttt{ in } d' \rceil \; : \\
& & \quad (\forall i . \text{a}_{\text{tf}}(f_i, env, s); \text{a}_{\text{d}}(d', env, s)) \\
& & \lceil \texttt{let } x : \sigma = d_1 \texttt{ handle } f \texttt{ in } d_2 \rceil \; : \\
& & \quad (\text{a}_{\text{d}}(d_1, env, s); \text{a}_{\text{f}}(f, env, s); \text{a}_{\text{d}}(d_2, env, s)) \\
& & \text{otherwise:}()
\end{array}
$$

Figure 6.15: Algorithm 14, part 1: infer truth or falsity of comparisons from `switch`es.

$$
\begin{array}{lcl}
\mathrm{a}_{\mathrm{e}}(e, env, s) & = & \text{case } e \text{ of} \\
 & & \lceil \texttt{switch}\ (st)\, x\ \texttt{of}\ (i : f_i)^*\ \texttt{default:}\ \ od \rceil\ : \\
 & & \quad \text{if } x \in \mathrm{dom}(env) \\
 & & \quad \text{then let } true\_s = S \cup \{env(x)\} \text{ and } false\_s = S \cup \{\mathrm{neg}(env(x))\} \\
 & & \qquad \text{in } \forall i.\text{if } i = 0 \\
 & & \qquad\quad \text{then } \mathrm{a}_{\mathrm{f}}(f_0, env, false\_s) \\
 & & \qquad\quad \text{else if } i = 1 \\
 & & \qquad\qquad \text{then } \mathrm{a}_{\mathrm{f}}(f_1, env, true\_s) \\
 & & \qquad\qquad \text{else impossible} \\
 & & \qquad \text{end} \\
 & & \quad \text{else } (\mathrm{a}_{\mathrm{f}}(f_i, env, s); \mathrm{a}_{\mathrm{d}}(od, env, s)) \\
 & & \lceil \texttt{switch}\ (st)\, v\ \texttt{of}\ (i : f_i)^*\ \texttt{default:}\ \ od \rceil\ : \\
 & & \quad (\mathrm{a}_{\mathrm{f}}(f_i, env, s); \mathrm{a}_{\mathrm{d}}(od, env, s)) \\
 & & \lceil \texttt{typecase}\ cv\ \texttt{of}\ [t.\sigma] ta_i^*\ \texttt{rectype}\ : otf\ \texttt{default} : od \rceil\ : \\
 & & \quad (\mathrm{a}_{\mathrm{ta}}(ta_i, env, s); \mathrm{a}_{\mathrm{tf}}(otf, env, s); \mathrm{a}_{\mathrm{d}}(od, env, s)) \\
 & & \lceil \texttt{tlistcase}\ cv\ \texttt{of}\ [t.\sigma]\ \texttt{nil}\ : d\ \texttt{cons} : tf \rceil\ : \\
 & & \quad (\mathrm{a}_{\mathrm{d}}(d, env, s); \mathrm{a}_{\mathrm{tf}}(otf, env, s)) \\
 & & \\
\mathrm{a}_{\mathrm{f}}(f, env, s) & = & \text{case } f \text{ of} \\
 & & \lceil \lambda(t_1 \ldots t_n).d \rceil\ : \mathrm{a}_{\mathrm{d}}(d, env, s) \\
 & & \\
\mathrm{a}_{\mathrm{tf}}(tf, env, s) & = & \text{case } tf \text{ of} \\
 & & \lceil \Lambda(t_1 \ldots t_n).d \rceil\ : \mathrm{a}_{\mathrm{d}}(d, env, s) \\
 & & \\
\mathrm{a}_{\mathrm{ta}}(a, env, s) & = & \text{case } a \text{ of} \\
 & & \lceil c_0 : d \rceil : \mathrm{a}_{\mathrm{d}}(d, env, s) \\
 & & \lceil c_1 : tf \rceil : \mathrm{a}_{\mathrm{d}}(tf, env, s)
\end{array}
$$

Figure 6.16: Algorithm 14, continued: infer truth or falsity of comparisons from `switch`es.

When the function $\mathrm{a_d}$ traverses a `let`-bound expression (see Figure 6.17), it evaluates the expression, updates A, and traverses the body of the `let`. When it traverses a function definition, it evaluates the function in its approximated domain, and updates the approximated range of the function. When it encounters a `raise` declaration, it returns $\perp$, since `raise` declarations never return a value.

The function $\mathrm{a_e}$ evaluates expressions. For one and two-argument primitives operators, $\mathrm{a_e}$ evaluates the arguments and then applies versions of the primitives which work on approximated values. For expressions which branch, $\mathrm{a_e}$ evaluates each of their branches and return the least upper bound of their branches. For function applications where the function being is applied is known, $\mathrm{a_e}$ adds their arguments to the domain of the function, and returns the range. When the function is unknown $\mathrm{a_e}$ returns $\top$. For polymorphic function applications, the function $\mathrm{a_e}$ does not update the domains of polymorphic functions because the arguments are known to be types. All other expressions return $\top$.

The interpretation has a running time of $O(N^2)$, where N is the size of the program. It takes $O(N)$ time to make one pass over the program. There are at most N variables, and each variable can have at most 4 abstract values (the height of the lattice). Since the values of variables only move upward in the lattice, at most 4N passes can be made before the data structures converge. Thus, the total running time is $4N * O(N)$, or $O(N^2)$.

I speculate that using dependency information to recalculate the abstract values of variables only when necessary can significantly improve the actual running time of the algorithm, perhaps to the point where the actual running time of the algorithm is linear in the size of the program.

The result of the interpretation is the map A, which maps variables to their approximated values. It is straightforward to convert A to a set of propositions P about variables. P is true for all program points, so converting P to be the mapping M needed by the following transformation phase is easy. M has to be a mapping from program points to propositions that are true at that point, so $M = \lambda x.P$.

6.5.3 Eliminating comparisons

Figures 6.20 and 6.21 show Algorithm 15, which does the actual elimination of comparisons. The algorithm is parameterized by a mapping M from program points (let-bound variable names) to sets of propositions that are true at that point. It traverses the program from top to bottom. At each `let`-bound boolean expression, it looks up the set of propositions that are known to be true. It then checks to see if the boolean expression is true or if the negation of the boolean expression is true in this set. If so, it replaces the expression with the appropriate constant.

The running time of this algorithm depends on the implementation of the function *provable*. If this function is the identity function (that is, uses only the set of syntactic propositions), then the time to process each boolean expression is $O(\log N)$, where N is program size. It takes constant time to look up the set of propositions in M. Because there are at most N propositions, it takes $O(\log N)$ time to check whether an expression is true or false. Because there are at most N steps, the total running time of this phase is $O(N \log N)$.

If, however, the function *provable* computes all provable propositions using transitive

$$
\begin{array}{lcl}
F & = & \ldots \\[1em]
A & = & \text{ref } \ldots \\[1em]
\mathrm{a_d}(d) & = & \text{case } d \text{ of} \\
& & \lceil v \rceil : \mathrm{a_V}(v) \\
& & \lceil \texttt{let } x : \sigma = e \texttt{ in } d' \rceil \; : \\
& & \quad \text{let } x' = \mathrm{a_e}(e) \\
& & \quad \text{in } A := !A[x \mapsto (!A)(x) \sqcup x']; \mathrm{a_d}(d') \\
& & \quad \text{end} \\
& & \lceil \texttt{let } cb \texttt{ in } d' \rceil \; : \\
& & \quad \mathrm{a_d}(d') \\
& & \lceil \texttt{let fix } x_1 : \sigma = f_1 \ldots \texttt{ and } x_n : \sigma = f_n \texttt{ in } d \rceil \; : \\
& & \quad (\forall i.\text{let } \{\text{dom} = d, \text{range} = r\} = F(x_i) \text{ and } s = \mathrm{a_f}(f_i, !d) \\
& & \qquad \text{in } r := (!r) \sqcup s \\
& & \qquad \text{end} \\
& & \quad\; \mathrm{a_d}(d)) \\
& & \lceil \texttt{let fixtype } x_1 : \sigma = f_1 \ldots \texttt{ and } x_n : \sigma = f_n \texttt{ in } d \rceil \; : \\
& & \quad (\forall i.\text{let } \{\text{dom} = d, \text{range} = r\} = F(x_i) \text{ and } s = \mathrm{a_{tf}}(f_i) \\
& & \qquad \text{in } r := (!r) \sqcup s \\
& & \qquad \text{end} \\
& & \quad\; \mathrm{a_d}(d)) \\
& & \lceil \texttt{raise } (v, \sigma) \rceil \; : \bot \\
& & \lceil \texttt{let } x : \sigma = d_1 \texttt{ handle } f \texttt{ in } d_2 \rceil \; : \\
& & \quad \text{let } x_1 : \sigma_1 = \mathrm{a_d}(d_1) \text{ and } x_2 = \mathrm{a_{ef}}(f) \\
& & \quad \text{in } A := (!A)[x \mapsto x_1 \sqcup x_2]; \mathrm{a_d}(d_2) \\
& & \quad \text{end} \\[1em]
\mathrm{a_V}(v) & = & \text{case } v \text{ of} \\
& & \lceil x \rceil : (!A)(x) \\
& & \lceil i \rceil : i \\
& & \lceil cp(v) \rceil : \sigma : \mathrm{a_V}(v) \\
& & \text{otherwise: } \top
\end{array}
$$

Figure 6.17: Rule-of-signs abstract interpretation, part 1: the function $\mathrm{a_d}$ is repeatedly applied to a program until the sets F and M do not change.

$$
\begin{array}{lcl}
\mathrm{a_e}(e) & = & \text{case } e \text{ of} \\
& & v : \mathrm{a_V}(v) \\
& & \lceil p_1(v) \rceil : \mathrm{a_{op1}}(p_1, \mathrm{a_V}(v)) \\
& & \lceil p_2(v_1, v_2) \rceil : \mathrm{a_{op2}}(p_2, \mathrm{a_V}(v_1), \mathrm{a_V}(v_2)) \\
& & \lceil \texttt{switch } (st)\, v \texttt{ of } (i : f_i)^* \texttt{ default: } od \rceil \; : \\
& & \quad \sqcup \mathrm{a_{ef}}(f_i) \sqcup \mathrm{a_d}(od) \\
& & \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma] ta_i^* \texttt{ rectype } : otf \texttt{ default} : od \rceil \; : \\
& & \quad \sqcup \mathrm{a_{ta}}(ta_i) \sqcup \mathrm{a_{tf}}(otf) \sqcup \mathrm{a_d}(od) \\
& & \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : d \texttt{ cons} : tf \rceil \; : \\
& & \quad \mathrm{a_d}(d) \sqcup \mathrm{a_{tf}}(otf) \\
& & \lceil v(v_1 \ldots v_n) \rceil : \\
& & \quad \text{if } v \text{ is a variable } x \text{ and } x \in \mathrm{dom}(!F) \\
& & \quad \text{then let } \{dom = d, range = r\} = F(x) \text{ and } (a_1, \ldots, a_n) = (!d) \\
& & \qquad \text{in } (d \; := \; (a_1 \sqcup \mathrm{a_V}(v_1), \ldots a_m \sqcup \mathrm{a_V}(v_n)); !r) \\
& & \qquad \text{end} \\
& & \quad \text{else } \top \\
& & \lceil v[v_1 \ldots v_n] \rceil : \\
& & \quad \text{if } v \text{ is a variable } x \text{ and } x \in \mathrm{dom}(!F) \\
& & \quad \text{then let } \{dom = d, range = r\} = (!F)(x) \\
& & \qquad \text{in } (!r) \\
& & \qquad \text{end} \\
& & \quad \text{else } \top \\
& & \text{otherwise: } \top \\
\\
\mathrm{a_f}(f, args) & = & \text{case } (f, args) \text{ of} \\
& & (\lceil \lambda(x_1 : \sigma_1 \ldots x_n : \sigma_n).d \rceil \; , a_1 \ldots a_n) : \\
& & \quad (\forall i.A \; := !A[x_i \mapsto a_i]; \mathrm{a_d}(d)) \\
\\
\mathrm{a_{ef}}(f) & = & \text{case } f \text{ of} \\
& & \lceil \lambda(x_1 : \sigma_1 \ldots x_n : \sigma_n).d \rceil \; : \\
& & \quad (\forall i.A \; := !A[x_i \mapsto \top]; \mathrm{a_d}(d)) \\
\\
\mathrm{a_{tf}}(tf) & = & \text{case } tf \text{ of} \\
& & \lceil \Lambda(t_1 \ldots t_n).d \rceil \; : \mathrm{a_d}(d) \\
\\
\mathrm{a_{ta}}(a) & = & \text{case } a \text{ of} \\
& & \lceil c_0 : d \rceil : \mathrm{a_d}(d) \\
& & \lceil c_1 : tf \rceil : \mathrm{a_{tf}}(tf)
\end{array}
$$

Figure 6.18: Rule-of-signs abstract interpretation, continued.

$$
\begin{array}{lcl}
\mathrm{a_{op1}}(op1, v) & = & \text{case } op1 \text{ of} \\
& & \texttt{size} : + \\
& & \text{otherwise} : \top \\
\\
\mathrm{a_{op2}}(op2, v_1, v_2) & = & \text{case } op2 \text{ of} \\
& & \texttt{plusi} : \\
& & \qquad \text{case } (v_1, v_2) \text{ of} \\
& & \qquad (\top, \_) : \top \\
& & \qquad (\_, \top) : \top \\
& & \qquad (\bot, \_) : \bot \\
& & \qquad (\_, \bot) : \bot \\
& & \qquad (+, +) : + \\
& & \qquad (-, -) : - \\
& & \qquad (+, -) : \top \\
& & \qquad (-, +) : \top \\
& & \qquad (+, i) : \text{if } i \geq 0 \text{ then } + \text{ else } \top \\
& & \qquad (i, +) : \text{if } i \geq 0 \text{ then } + \text{ else } \top \\
& & \qquad (-, i) : \text{if } i \leq 0 \text{ then } - \text{ else } \top \\
& & \qquad (i, -) : \text{if } i \leq 0 \text{ then } - \text{ else } \top \\
& & \qquad (i, j) : i + j \\
& & \texttt{minusi} : \\
& & \qquad \text{let } v_2' = \begin{array}[t]{l} \text{case } v_2 \text{ of} \\ i : -i \\ + : - \\ - : + \\ \bot : \bot \\ \top : \top \end{array} \\
& & \qquad \text{in } \mathrm{a_{op2}}(\texttt{plus}, v_1, v_2') \\
& & \qquad \text{end} \\
& & \texttt{muli} : \\
& & \qquad \ldots \\
& & \texttt{divi} : \\
& & \qquad \ldots \\
& & \text{otherwise: } \top
\end{array}
$$

Figure 6.19: Definitions of abstract operators for rule-of-signs abstract interpretation.

closure, then the running time of this phase is $O(N^4)$. At each `let`-bound expression point, computing transitive closure may take $O(N^3)$ time, using an N by N boolean matrix representing propositions which are true between variables. It takes constant time to check the matrix to see whether the boolean expression is true or false. Because there are at most N steps, the total running time of this phase is $O(N^3) * N = O(N^4)$.

6.5.4 Correctness

In this section, I sketch how to prove the correctness of redundant comparision elimination by using a variant of the call-by-value λ-calculus that supports pre-and-post conditions.

If we are using Algorithm 14, which infers propositions from `switch`es, we can modify the algorithm to produce programs annotated with pre-and-post conditions. That is, instead of the algorithm producing a mapping from program points to propositions that are true at those program points, the algorithm can annotate the program with those propositions.

A statement and proof that the modified algorithm produces an annotated program could take the following lines. Let $\mathrm{c_d}$ be the main function of the algorithm that takes a program P and a pre-condition A and produces an annotated program. $\mathrm{c_d}(P, A)$ produces properly annotated programs. The proof is by structural induction on P.

We can then modify Algorithm 15 to take annotated programs, and prove that it correctly eliminates comparisons from properly annotated programs. The proof would again be by a structural induction on (properly annotated) programs.

If we are using the rule-of-signs abstract interpretation, we would need to modify the abstract interpretation to produce properly annotated programs. This is not hard to do. Recall that the abstract interpretation produces an abstract environment that can be regarded as a set of propositions about variables. We can simply annotate each program point with the propositions that are true for the variables in scope at that point.

6.5.5 Overall asymptotic complexity

The asymptotic complexity of redundant comparison elimination depends on exactly what analysis we use and what set of propositions infer from the set collected by the analysis. If we infer propositions from `switch`es, and do not try to infer any additional propositions, then the running time is $O(N \log N)$. If we include a rule-of-signs analysis, the running time increases to $O(N^2)$. If we compute the transitive closure of the set of available propositions, the running time increases to $O(N^4)$.

It should be noted that even an $O(N^4)$ running time is feasible in practice, because N is actually bounded by the number of variables with integer types in scope at any particular point in the program.

In practice, TIL uses the $O(N^4)$ version of redundant comparison elimination. This uses the most accurate — and expensive — analysis. Chapter 9 shows that this version of redundant comparison elimination offers little benefit in practice.

6.6 Ordering the optimizations

$$
\begin{array}{lcl}
\mathrm{c_d}(d) & = & \text{case } d \text{ of} \\
& & \lceil v:\sigma\rceil : \lceil v:\sigma\rceil \\
& & \lceil \texttt{let } x:\sigma = e \texttt{ in } b\rceil \; : \\
& & \quad \text{let } e' = \mathrm{c_e}(e) \\
& & \quad \text{in if } e' \in \mathit{provable}((!M)(x)) \\
& & \qquad \text{then } \lceil \texttt{let } x:\sigma = \texttt{true in } \mathrm{c_d}(b)\rceil \\
& & \qquad \text{else if not}(e') \in \mathit{provable}(!M)(x) \\
& & \qquad\quad \text{then } \lceil \texttt{let } x:\sigma = \texttt{false in } \mathrm{c_d}(b)\rceil \\
& & \qquad\quad \text{else } \lceil \texttt{let } x:\sigma = e' \texttt{ in } \mathrm{c_d}(b)\rceil \\
& & \quad \text{end} \\
& & \lceil \texttt{let } cb \texttt{ in } d\rceil \; : \\
& & \quad \lceil \texttt{let } cb \texttt{ in } \mathrm{c_d}(d)\rceil \\
& & \lceil \texttt{let fix } x_1:\sigma_1 = f_1 \ldots \texttt{ and } x_n:\sigma_n = f_n \texttt{ in } d'\rceil \; : \\
& & \quad \lceil \texttt{let fix } x_1:\sigma_1 = \mathrm{c_f}(f_1) \ldots \texttt{ and } x_n:\sigma_n = \mathrm{c_f}(f_n) \\
& & \quad\; \texttt{in } \mathrm{c_d}(d')\rceil \\
& & \lceil \texttt{let fixtype } x_1:\sigma_1 = tf_1 \ldots \texttt{ and } x_n:\sigma_n = tf_n \texttt{ in } d'\rceil \; : \\
& & \quad \lceil \texttt{let fixtype } x_1:\sigma_1 = \mathrm{c_{tf}}(tf_1) \ldots \texttt{ and } x_n:\sigma_n = \mathrm{c_{tf}}(tf_n) \\
& & \quad\; \texttt{in } \mathrm{c_d}(d')\rceil \\
& & \lceil \texttt{let } x_1:\sigma = d_1 \texttt{ handle } f \texttt{ in } d_2\rceil \; : \\
& & \quad \lceil \texttt{let } x_1:\sigma = \mathrm{c_f}(f) \texttt{ handle } \mathrm{c_d}(d_1) \texttt{ in } \mathrm{c_d}(d_2)\rceil \\
& & \lceil \texttt{raise } (v,\sigma)\rceil \; : \\
& & \quad \lceil \texttt{raise } (v,\sigma)\rceil \\
\\
\mathrm{c_e}(e) & = & \text{case } e \text{ of} \\
& & \lceil \texttt{switch } (st)\, v \texttt{ of } (i:f_i)^* \texttt{ default: } od\rceil \; : \\
& & \quad \lceil \texttt{switch } (st)\, v \texttt{ of } i:\mathrm{c_f}(f_i)^* \texttt{ default: } \mathrm{c_d}(od)\rceil \\
& & \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma]ta_i^* \texttt{ rectype } : otf \texttt{ default} : od\rceil \; : \\
& & \quad \lceil \texttt{typecase } cv \texttt{ of } [t.\sigma]\mathrm{c_{ta}}(ta_i)^* \\
& & \quad\; \texttt{rectype } : \mathrm{c_{tf}}(otf) \\
& & \quad\; \texttt{default} : \mathrm{c_d}(od)\rceil \\
& & \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : d \texttt{ cons} : tf\rceil \; : \\
& & \quad \lceil \texttt{tlistcase } cv \texttt{ of } [t.\sigma] \texttt{ nil } : \mathrm{c_d}(d) \texttt{ cons} : \mathrm{c_{tf}}(tf)\rceil \\
& & \text{otherwise} : e
\end{array}
$$

Figure 6.20: Algorithm 15: traversing expressions and declarations.

$$
\begin{array}{lcl}
\mathrm{c_f}(f) & = & \text{case } f \text{ of} \\
 & & \lceil \lambda(x_1 \ldots x_n).d \rceil \;\; : \lceil \lambda(x_1 \ldots x_n).\mathrm{c_d}(d) \rceil \\[1em]
\mathrm{c_{tf}}(tf) & = & \text{case } tf \text{ of} \\
 & & \lceil \Lambda(t_1 \ldots t_n).d \rceil \;\; : \lceil \Lambda(t_1 \ldots t_n).\mathrm{c_d}(d) \rceil \\[1em]
\mathrm{c_{ta}}(a) & = & \text{case } a \text{ of} \\
 & & \lceil c_0 : d \rceil : \lceil c_0 : \mathrm{c_d}(d) \rceil \\
 & & \lceil c_1 : tf \rceil : \lceil c_1 : \mathrm{c_{tf}}(tf) \rceil
\end{array}
$$

Figure 6.21: Algorithm 15, continued: traversing expressions and declarations.

Figure 6.22 shows the order in which optimizations are applied. The order is similar to that used by Appel in the SML/NJ compiler [7]. First, a round of reduction optimizations are done. These optimization include constant-folding, common-subexpression elimination, redundant switch elimination, inlining of functions only called once, invariant removal, hoisting of constant expressions, and dead-code elimination. These optimizations are guaranteed to not increase the size of a program; they should make programs smaller and faster. These optimizations may be iterated until no further reductions occur. Next, after shrinking the program in size, switch-continuation inlining, sinking, uncurrying, comparison elimination, and inline expansion are done. The entire process is iterated two more times.

It is important that some optimizations be done before others. CSE should always be done before inline expansion, because CSE can eliminate applications of polymorphic functions. This makes it less likely that duplicate copies of polymorphic functions will be created by inline expansion. CSE should also be done before redundant switch elimination. Redundant switch elimination looks for switches on the same variable; CSE may show that one variable can be replaced by another.

With no iteration during the "shrink" phase, the asymptotic complexity of the optimizer ranges from $O(N \log N)$ to $O(N^4)$, where N is the size of the program. The asymptotic complexity of the optimizer is determined by the worst-complexity of any particular optimization, because each optimization is done only a constant number of times and each optimization increases program size only by a constant factor. Table 6.1 shows the asymptotic complexities of each optimization. The asymptotic complexity of the optimizer is $O(N \log N)$ if a simple version of redundant comparison elimination is done and only expressions without declarations nested within them are moved by invariant removal and hoisting of constant expressions. It is $O(N^2)$ if invariant removal and hoisting are allowed to move any kinds of expressions and the rule-of-signs analysis is used during redundant comparison elimination. It is $O(N^4)$ if transitive closure is used during redundant comparision elimination.

Iterating optimizations during the "shrink" phase is useful because these optimizations interact, particularly with inlining of functions only called once [9]. For example, inlining of a function only called once may expose the fact that a variable passed to a function is in fact unused, thus presenting an opportunity for dead-code elimination. It may also expose

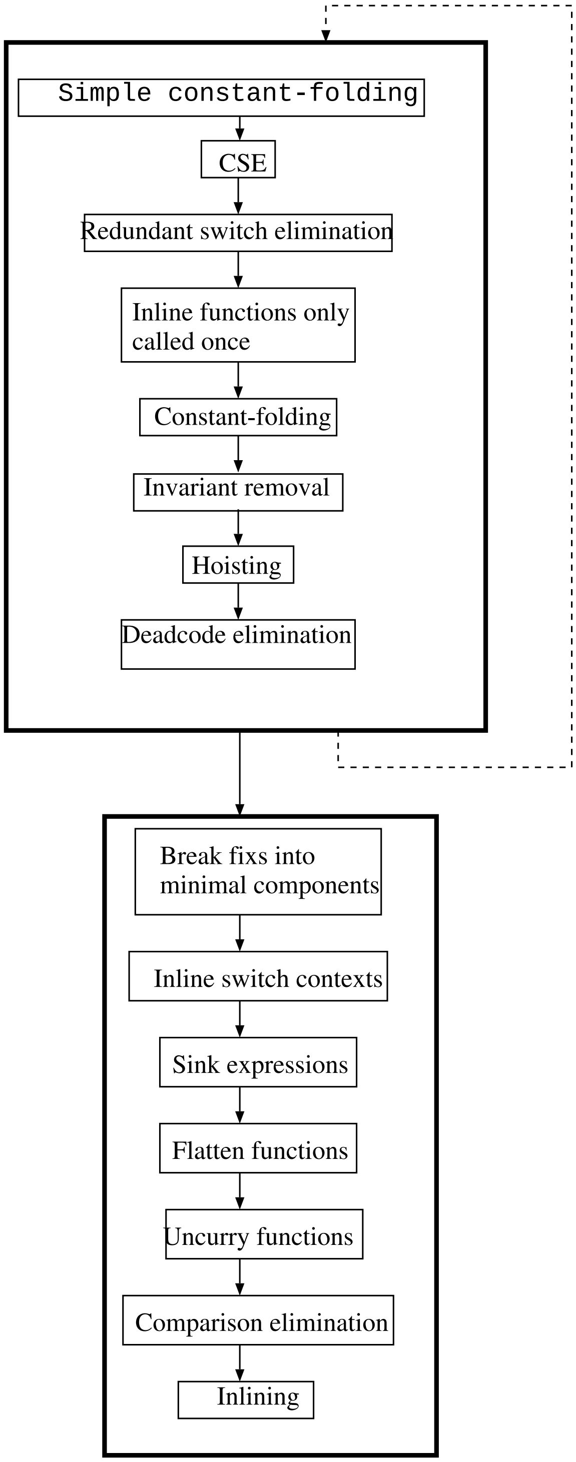

Figure 6.22: Order in which optimizations are applied.

Optimization	Asymptotic running time (N = LMLI program size)
Dead-code elimination	$O(N)$
CSE	$O(N \log N)$
Constant folding	$O(N \log N)$
Inlining of functions only called once	$O(N)$
Inline expansion	$O(N)$
Invariant removal	$O(N)$ or $O(N^2)$
Hoisting of constant-expressions	$O(N)$ or $O(N^2)$
Redundant switch elimination	$O(N \log N)$
Redundant comparison elimination	$O(N \log N)$ to $O(N^4)$
Switch-continuation inlining	$O(N)$
Sinking	$O(N \log N)$
Uncurrying	$O(N \log N)$

Table 6.1: Asymptotic running times of optimizations

the fact that an expression is invariant, and can be moved out of loop.

It is unclear, however, that this iteration is worthwhile. First, iteration can change the asymptotic complexity of this phase from $O(N \log N)$ to $O(N^2 \log N)$, where N is the original size of the program. N iterations may need to done, each taking $O(N \log N)$ time. Second, all the individual optimizations in this phase can be implemented so that each one converges in one pass. For example, CSE, redundant switch elimination, and hoisting as described in this chapter converge in one pass. Invariant removal and inlining as implemented (but not as described) also converge in one pass. Third, the additional optimizations done during iteration may improve performance only a small amount.

6.7 Related work

In this section, I describe related work on optimization of functional languages. That is, this section also covers some related work for the previous chapter.

Steele [85] proposes the use of source-to-source transformations in compiling SCHEME, using a λ-calculus based intermediate language. His optimizer does some simple forms of inlining, eliminates unused arguments to functions, does short-circuiting of if-statements, and does constant-folding of primitive operations. His description is complicated by the fact that he uses a subset of SCHEME for his intermediate language. SCHEME differs from the call-by-value λ-calculus in that it allows variables to be assigned.

Kranz *et al.* [48, 52, 51] also use source-to-source transformations in the ORBIT compiler for SCHEME. Their set of optimizations is similar to Steele's set, with one notable difference: they propagate values, such as small function definitions, across compilation units.

Appel and Jim [10, 7] continue in this vein by using source-to-source transformations in compiling SML. Their optimizer does inlining, uncurrying, constant-folding, and dead-code elimination. In addition, they also do hoisting (moving variable bindings up or down in scope) and a form of CSE. They move function definitions up or down in scope (so that closures may be shared or not constructed until necessary), and they also move operators that are used only once into branches of switch statements if possible. However, they do not hoist expressions of out of function bodies (that is, they do not do a form of hoisting analogous to invariant removal).

Appel [7] finds that hoisting provides small improvements in execution time (5-10%) and that CSE provides no improvement in execution time. It is unclear why his form of CSE provides no improvement, because it is quite similar to my form of CSE. I speculate that CSE in the SML/NJ compiler interacted poorly with the closure conversion phase of the compiler. At that time, the SML/NJ compiler did not share closures and it also allocated continuation closures instead of stack frames. Thus, every common subexpression live across a function call was written to memory (saved in a continuation closure). Indeed, it could be written to memory at every function call, offsetting the benefit of avoiding the extra computation. In TIL's register allocator, a variable that is bound to a common subexpression and live across a function call may not be written to memory at all if it is saved in a callee-save register. If the variable is written to memory, it is written to the stack only once because TIL uses Chaitin's graph coloring register allocation, which allocates a variable to memory or a register for its entire live range.

Shivers [83] advocates the importance of adapting optimizations used in compilers for C and FORTRAN so that the optimizations can be applied to languages like SML. He shows how to approximate the control-flow graph for programs that use higher-order functions. He shows to use this information to do induction-variable elimination (IVE) and to potentially improve inlining. The optimizations that I describe in this chapter extend Shivers work. I have focused on the simpler problem of applying optimizations to programs that use first-order, recursive functions.

Peyton-Jones [69] and Santos [79] describe an optimizer for the Glasgow Haskell compiler that is similar in spirit to my optimizer. Like my optimizer, their optimizer uses many simple transformations that are iterated repeatedly. Their optimizer also uses a typed intermediate language, although the Glasgow compiler is unable to use type information to avoid tagging data (laziness, and the need to check for whether an expression is a thunk that needs to be evaluated or a value, seems to impede this).

Their transformations are similar at a high-level to my transformations: they do inlining, constant-folding, dead-code elimination, code motion optimizations such as hoisting, and optimizations on switch statements. The optimizers differ in their details because of the different source languages. Their optimizer must be careful to preserve laziness and to reason about laziness. My optimizer must be careful to preserve side-effects.

6.8 Conclusions

In this chapter, I have shown how to apply several optimizations that are known to improve loops in imperative languages such as C and FORTRAN to programs in a λ-calculus intermediate language. I have presented new algorithms to do common-subexpression elimination, elimination of redundant branches, hoisting of constant expressions, invariant removal, and eliminating unneeded comparisons. I have analyzed the asymptotic complexity of these algorithms and shown that core versions of these algorithms have an $O(N)$ or $O(N \log N)$

asymptotic complexity, where N is program size. This shows that the algorithms are practical. For hoisting of constant expressions and invariant removal, I have shown that extending these optimizations to move functions or branching expressions increases their asymptotic complexity form $O(N)$ to $O(N^2)$. For redundant comparision elimination, the asymptotic complexity may range between $O(N^2)$ and $O(N^4)$.

In addition, I have separated analysis from transformation when presenting the algorithms, and have argued that the correctness of all but one of the transformation phases can be justified by variants of the call-by-value λ-calculus.

Chapter 7

Compilation to machine code

In this chapter, I describe how TIL compiles B-form[1] programs to ALPHA machine code. First, I describe my approach to compiling B-form programs to machine code. Then, I give an overview of the phases that implement the translation. Next, I explain each phase in detail and show how TIL translates a polymorphic function to machine code.

My approach to compiling B-form programs to machine code is quite different from the approach suggested in the literature. Many researchers who have implemented higher-order languages such as SML have stated the importance of compiling functions well [52, 7, 81] — they have focused on strategies for representing environments for first-class functions. I believe that this is not the central problem for compiling languages such as SML, because optimization is so effective at eliminating higher-order functions. I will show the effectiveness of optimization at eliminating higher-order functions in Chapter 10.1. In contrast to the approaches suggested in the literature, I have taken conventional compiler technology and adapted it to compile B-form programs to machine code. Thus, I use a simple closure conversion strategy, but combine it with a conventional graph-coloring register allocator that uses callee and caller-save registers. To support the register allocator, I use a sophisticated tag-free garbage collector. I make certain that the translation to machine code recognizes constant expressions, such as constant records, and converts them to data at compile time. Also, I use the standard system assembler to do instruction scheduling. The end result is a translation to machine code that is similar to the approach used in more conventional compilers.

The translation to machine code consists of five phases. The first phase is *type reification*, which converts types that may be needed by the garbage collector to constructors. This phase examines the type of each bound variable; if it cannot determine from the type at compile time whether the garbage collector must trace the variable, it converts the type to a constructor, so that the garbage collector can examine the constructor at run time to determine whether it must trace the variable. For example, if the type of a variable x is a typecase on a constructor variable bound by a polymorphic function, type reification converts the type to a constructor. If a garbage collection occurs while x is live, the garbage collector can examine the constructor to determine whether it must trace the variable. In constrast, if the type of x is a record type, type reification does not convert the type to a

[1] The subset of LMLI produced by the optimizer.

constructor: TIL generates a table at compile time that tells the garbage collector it must trace the variable.

The second phase is *closure conversion*, which Morrisett implemented. Closure conversion closes each function (ensures that it has no free variables), so that the function will be suitable for translation to a machine-code function. It uses a simple strategy for choosing closure representations that depends on whether the function is known or is escaping. If the function is known, closure conversion adds the free constructor variables and value variables to the arguments for the function, following Kranz's approach [52]. If the function is escaping, closure conversion builds a closure for the function, which consists of a constructor environment, a value environment, and a closed version of the function. The environments respectively bind free constructor variables and value variables; closure conversion represents them as flat records [7]. The closed version of the function takes the environments as two additional arguments; closure conversion changes all references to free variables to be projections from the environments. Closure conversion closes only over variables bound within functions; it does not close over top-level variables that are not bound within functions.

TIL applies the global optimizer after both type reification and closure conversion; we can do this because type reification and closure conversion operate on B-form programs. For type reification, the optimizer eliminates common subexpressions involving reified types and moves loop-invariant constructor computations. For closure conversion, the optimizer sinks projections from environments into branches when possible and does constant folding, as described in Section 5.6.

The third phase is *translation to U-Bform*. This phase collapses similar constructor and term-level constructs to just term-level constructs. For example, it maps constructor tuples and record expressions to U-Bform record expressions. This simplifies the next phase of the compiler, which converts programs to a low-level assembly language, by reducing the size of the language on which that phase must operate. This phase also maps primitive constructors to their run-time representations. For example, it maps the `Int` constructor to 0 and the `Real` constructor to 1.

Because the translation collapses the constructor level and the term level together, U-Bform is an untyped language. Instead of being annotated with types, U-Bform variables are annotated with information that tells the garbage collector whether it must trace the variable. Note that this information may direct the garbage collector to consult a constructor variable at run time to determine traceability. For example, if a B-form variable `x` has type α, the garbage collector must consult α *at run time* to determine the traceability of `x`.

The fourth phase is *translation to register-transfer language (RTL)*. This phase selects sequences of assembly language instructions that implement higher-level U-Bform constructs. In addition, it chooses how each U-Bform variable is stored at the machine level (for example, as a compile-time constant, a global variable, or a variable local to a function). It also recognizes constant expressions and converts them to static data when possible. This phase also annotates RTL variables with representation information that is similar to the information that annotates U-Bform variables.

The fifth phase is *conversion to ALPHA assembly language and register allocation*. This phase uses conventional graph-coloring register allocation. It also generates the tables that tell the garbage collector how to determine which stack frame locations and registers must

```
fun map f =
    let fun m nil = nil
          | m (a::r) = (f a) :: (m r)
    in  m
    end
```

Figure 7.1: Original SML source code

be traced.

In the following sections, I describe each of the phases in more detail, and illustrate how each phase transforms the intermediate code for a polymorphic `map` function.

Figure 7.1 gives the SML source code for the `map` function. Figure 7.2 shows the B-form source code produced after optimization. I have simplified this source code: I have omitted some type information and turned off type-directed argument flattening during optimization.

7.1 Type reification

Type reification is divided into two conceptual passes: the first pass places all types on bound term variables in head-normal form (HNF) and the second pass examines the head constructor of each type to determine whether to reify the type. If the head constructor is

- a constructor variable, then type reification does nothing, because the type is already available at run time.
- a primitive constructor of arity-0 or arity-1, then type reification does nothing, because it can determine the traceability of this variable at compile time.
- a recursive type, then type reification unrolls the type once and examines the head constructor of the unrolled type. Type reification unrolls the recursive type by substituting the definition of the type for every occurrence of the type.
- a `typecase`, `fold`, or `listcase`, then type reification converts the type to a constructor computation, because we cannot determine traceability until run time.

For example, in the function `m` in Figure 7.2, consider the variables `a`, `r`, and `x6`. The type of `a`, `T(t1)`, is available at run time already. To place the type for `r` in HNF, we apply the `list` constructor function. The result is the recursive type list specialized to `t1`. When we unroll this type, we find that the head constructor is the primitive constructor `enumorrec`. Thus, we do not convert the type of `r` to a constructor. When we examine the HNF of the type of `x6`, we see that it is a record type, and we also do not convert the type to a constructor.

I require that all types for bound term variables be placed in HNF because closure conversion does not preserve the traceability of variables of arbitrary types. The HNF of

```
fixt map =
 Λt1,t2.
   let fix map' =
     λf: T(Let con t3=t1::nil
               con t4=<t3,t2>
               con t5=arrow[t4]
           in t5
           end).
      let fix m =
        λl : T(Let con t3=list[t1] in t3 end).
          let val x1: T(Let con t3=list[t2] in t3 end) =
               Switch_enumorrec l
               of 0 => λx2.enum 0 : T(Let con t3=list[t2] in t3 end)
                | 1 =>
                 λx3.let val a : T(t1) = (#0 x3)
                         val r : T(Let con t3=list[t1] in t3 end) = (#1 x3)
                         val x4 : T(t2) = f(a)
                         val x5 : T(Let con t3=list[t2] in t3 end) = m(r)
                         val x6 : T(Let con t3=list[t2]
                                        con t4=t3::nil
                                        con t5=t2::t4
                                        con t6=record[t5]
                                    in t6
                                    end) = {x4,x5}
                     in x6
                     end
          in x1
          end
      in m
      end
   in map'
   end
```

Figure 7.2: Simplified B-form source code after optimization

a type before and after closure conversion can differ, because closure conversion closes over constructor variables. Thus, traceability for a variable before closure conversion can differ from the traceability for the variable after closure conversion. However, closure conversion preserves the head constructor of the HNF of a type, so placing types in HNF before closure conversion prevents closure conversion from changing the traceability of a variable.

Type reification adds the following invariant to programs: if a bound term-variable has a type t whose head constructor when in HNF is not a primitive constructor, then t *must* be a constructor variable.

The actual implementation of type reification in TIL differs from the idealized description that I just gave. In practice, the first pass of type reification does not put all types in HNF to avoid unnecessarily increasing the size of intermediate programs. For example, if the only free variables of a type are either

- bound to known constructor functions
- or bound by polymorphic functions

and the type (and any known constructor functions that it calls) do not call any unknown constructor functions, the type does not need to placed in HNF. The normal form of the closure-converted type and the HNF of the original type will have the same head constructors.

This complicated criteria is actually met quite often: the front end of TIL translates SML datatypes to known constructor functions. These known constructor functions are then applied in many different places to simple types, such as a type variable or a primitive type.

If we did not do this optimization, placing types in HNF would inline the definition of an SML datatype at every place where the SML datatype is used to construct a type.

The example in Figure 7.2 already satisfies the preceding criteria, and in practice none of its types need to be placed in HNF. Thus, type reification leaves the example unchanged. Note that without optimizing what types are placed in HNF, the definition of the `list` constructor would be expanded at six different places in the program.

7.2 Closure conversion

To support closure conversion, Morrisett extended B-form expressions and declarations with four additional constructs:

- a `fixcode` declaration, which binds functions that take constructors and values as arguments,
- a `fixval` declaration, which binds a set of values mutually-recursively, much like `letrec` in Scheme. This allows us to create environments for recursive functions,
- a `closure` expression,
- and a `call` expression, which applies a closed function to its arguments.

(declarations)	d	::=	... | `let fixcode` $(x : \sigma$ `=` $codefn)^*$ `in` d
			`let fixval` $(x : \sigma$ `=` $e)^*$ `ind` d
(code)	$codefn$	::=	$\Lambda(t :: \kappa)^*.\lambda(x : \sigma)^*.decl$

Figure 7.3: Syntax of additional declarations for closure conversion

(expressions) e ::= ... | `closure`(v, cv, v) | `call`$(v, cv*, v*)$

Figure 7.4: Syntax of additional expressions for closure conversion

Figures 7.3 and 7.4 show the syntax for these additional constructs. The `fixcode` declaration binds functions that take both constructors and values as arguments. The types of the value variables are within the scope of the constructor variables. The `fixval` declaration binds only `record` expressions, `inject` expressions, and `closure` expressions.

A `closure` expression builds a closure from a closed function, a constructor variable environment, and a value variable environment. A `call` expression applies a closed function to its arguments. Note that the semantics of regular function application changes after closure conversion: a regular function application extracts the closed function and the environments from the closure, adds the environments to the appropriate argument lists, and invokes a `call` expression.

Closure conversion takes every B-form function bound by a `fix` or `fixtype` and maps it to a `fixcode` function. If a function escapes, closure conversion adds the constructor environment and the value environment to the respective argument lists of the function, making them the first arguments. If a function does not escape, then closure conversion does not pass environment arguments to the function. Instead, it changes all applications of the function to use the `call` construct, and passes the free variables as additional arguments.

Closure conversion uses a flat environment representation for the constructor and value environments [7]; it builds them using the Tuple constructors and `record` expressions respectively.

Closure conversion closes over only variables that are function arguments or are bound within functions. The locations of top-level variables are known at compile time, so their values are not stored in closures.

Figure 7.2 shows the `map` example after closure conversion. Because all the functions escape, they all take a constructor environment and value environment as arguments. For example, the closed version of map, `closed_map`, takes a constructor environment `cenv` as its first constructor argument. It takes a value environment `env` as its first value argument. In the function `closed_map'`, the type of the value variable f depends on the constructor environment `cenv`. In the closed function `m`, the types of the value environment `env` and the list argument `l` depend on the constructor environment.

All free variables are now projections from the appropriate environment. For example, in the type of the `f` argument to the function `closed_map'`, `t1` and t2, which were previously free, are now bound to projections from the constructor environment.

7.3 Translation to U-Bform

Figure 7.6 shows the syntax of U-Bform, which is a much smaller language than B-form. U-Bform is divided into four syntactic classes: traceabilities, which tell the garbage collector how to determine whether to trace a variable, values, expressions, and declarations.

There are six kinds of traceabilities:

- `TRACE`, which denotes values that the garbage collector must trace, such as records. At run time, these values will either be pointers or small integers that can be distinguished from pointers,
- `CODE`, which denotes pointers to closed functions,
- `INT`, which denotes integers,
- `REAL`, which denotes floating point numbers,
- `COMPUTE` π, which denotes values whose traceability is not known until run time. π is a path to the traceability information. The path is either the name of variable or a projection from a variable.
- `NOTNEEDED`, which denotes value whose traceability is not known until run time, but whose traceability is also not needed at run time. For example, traceability is not needed at run time for the return value of a function. The value of the function is immediately bound to a variable (whose traceability information is known), and a garbage collection cannot occur during the binding operation.

Values include variables, integers, floating point numbers, small integers that can be distinguished from pointers, and external identifiers. Expressions include values, strings, primitive operators of one argument, primitive operators of two arguments, array update, record creation, `write` expressions, and `switch` expressions for branching.

We use `write` expressions to create circular records for the mutually-recursive `fixval` declarations that closure conversion introduces and to create circular representations for recursive constructors. A write expression `write`(i, v_1, v_2) modifies the field of a record: it overwrites the ith field of v_1 with v_2. v_1 must be a variable that is bound to a record, inject, or closure expression.

We do not introduce a `fixval` construct at the U-Bform level for the following reason: we cannot use the trick of allocating an entire record for recursive constructors all at once at the machine level, as is typically done for environments for recursive functions. In general, we do not know how to tie the knot for recursive constructors: any amount of computation

```
fixcode closed_map = Λcenv,t1,t2.λenv:T(Let con t3=record[nil] in t3 end).
     let fixcode closed_map' = Λcenv.λenv:T(Let con t3=record[nil] in t3 end),
                 f:T(Let con t1=#0[cenv]
                         con t2=#1[cenv]
                         con t3=t1::nil
                         con t4=<t3,t2>
                         con t5=arrow[t4]
                     in t5
                     end).
           let con t1=#0[cenv]
               con t2=#1[cenv]
               fixcode m = Λcenv.λenv:T(Let con t1=#0[cenv]
                                   con t2=#1[cenv]
                                   con t3=t1::nil
                                   con t4= <t3,t2>
                                   con t5=arrow[t4]
                                   con t6=list[t1]
                                   con t7=t6::nil
                                   con t8=list[t2]
                                   con t9=<t7,t8>
                                   con t10=arrow[t9]
                                   con t11=t10::nil
                                   con t12=t5::t11
                                   con t13=record[t12]
                               in t13
                               end),
                       l:T(Let con t1=#0[cenv]
                               con t2=list[t1]
                           in t2
                           end).
               let con t1= #0[cenv]
                   con t2= #1[cenv]
                   val m' : ... = #1 env
                   val f : ... = #0 env
                   val x1: T(Let con t3=list[t2] in t3 end) =
                        Switch_enumorrec l
                         of 0 => λx2.enum 0
                          | 1 => λx3.
                            let val a:t1 = #0 x3
                                val r:T(Let con t3 = list[t1] in t3 end) = (#1 x3)
                                val x4:T(t2) = f(a)
                                val x5:T(Let con t3 = list[t2] in t3 end) = m(r)
                                val x6:T(Let con t3 = list[t2]
                                             con t4 = t3::nil
                                             con t5 = t2::t4
                                             con t6=record[t6]
                                         in t6
                                         end) = {x4,x5}
                            in x6
                            end
                in x1
                end
               con cenv'=<t1,t2>
               fixval env' : ... = {f,clo}
               and clo : ... = closure{m,cenv',env'}
           in  clo
           end
      con cenv' = <t1,t2>
      val map' : ... = closure{map',cenv,{}}
    in map'
    end
val map : ... = closure{closed_map,empty_cenv,{}}
```

Figure 7.5: B-form code after closure conversion

may occur during the evaluation of a recursive constructor. Thus, we are forced to allocate a dummy record for the constructor, and overwrite fields with their correct values later.

`Switch` expressions take declarations instead of functions as their arms. Declarations include value declarations, `let` declarations, which bind variables, `fix` declarations, which bind sets of mutually-recursive functions, and `raise` and `handle` declarations for exceptions.

TIL compiles B-form primitive constructors, recursive constructors, and lists of constructors to the following SML datatypes:

```
datatype con = Int | Real | String | Intarray | Realarray
             | Exn | Enum | Ptrarray of con | Arrow of con
             | Sum of conlist | Record of conlist
             | Enumorrec of conlist | Enumorsum of conlist
             | Excon of con | Deexcon of con | Mu of con

and conlist  = Nil | Cons of con * con
```

TIL implements the `con` datatype as a specialized Enumorsum, while it implements the `conlist` as an Enumorrec sum. It translates `Int` through `Enum` to small integers 0 through 5; it translates `Ptrarray` through `Mu` to a sum with 9 cases. For the `conlist` datatype, TIL translates `Nil` to the small integer 0 and `Cons` to a record. Tuples of constructors are translated to records.

TIL maps other B-form constructors and constructor declarations to corresonding U-Bform terms. It maps Listcase to a switch on the `conlist` datatype; it translates Fold, which folds a function across a list of constructors, to an U-bform `Fix` that applies the function to the list of constructors. It maps Typecase to a switch on the `con` datatype. TIL maps App to function application.

TIL's mapping of B-form values, expressions, and declarations to U-Bform terms is straightforward. A few of the cases need explanation. TIL maps `inject(`i`,`v_1 `...`v_n`)` to `record(`i`,`v_1`, ..., `v_n`)`. TIL translates each arm of a switch `switch (v) ...` from a B-form function to a U-Bform declaration that binds the function arguments. The bindings that TIL creates for the function arguments depend on the switch. If the switch is on:

- integers or enumerated sums, then the switch arms are functions without arguments, so TIL does not create any bindings.

- `enumorrec` or `enumorsum` sums, then the switch arms have the form $\lambda x.d$, so TIL binds each argument variable x to v (the argument of the switch).

- a general sum, then the switch arms have the form $\lambda x_i \ldots x_n.d$. For each bound variable x_i, TIL binds x_i to the $i+1$ field of v (the first field contains the case of the sum).

Recall from the preceding section on closure conversion that B-form applications are now applications of closures. TIL translates each B-form application to a U-Bform expression that extracts the closed function, the constructor environment, and the value environment from the closure and applies the closed function with the environments as additional arguments. It translates a B-form `call` to a U-Bform application.

(g.c. info)	ρ	$::=$	$\texttt{TRACE} \mid \texttt{CODE} \mid \texttt{INT} \mid \texttt{REAL} \mid \texttt{COMPUTE}\ \pi \mid \texttt{NOTNEEDED}$
(paths)	π	$::=$	$x \mid x.i$
(values)	v	$::=$	$x \mid i \mid r \mid \texttt{enum}(i) \mid \texttt{extern}(s, \rho)$
(expressions)	e	$::=$	$v \mid s \mid p_1(v) \mid p_2(v_1, v_2) \mid \texttt{update}(ar, v_1, v_2, v_3) \mid \texttt{record}(v^*) \mid$ $\texttt{write}(i, v_1, v_2) \mid \texttt{switch}(st)\ v\ \texttt{of}\ (i : d)^*\ \texttt{default} : od \mid$ $v(v^*)$
(declarations)	d	$::=$	$v : \rho \mid \texttt{let}\ x : \rho = e\ \texttt{in}\ d \mid \texttt{let fix}\ (x\texttt{=}f)^*\ \texttt{in}\ d \mid$ $\texttt{raise}\ v \mid \texttt{let}\ x : \rho = d_1\ \texttt{handle}\ f\ \texttt{in}\ d_2$
(functions)	f	$::=$	$\lambda(x{:}\rho)^* : \rho.d$
(opt. decl)	od	$::=$	$d \mid \epsilon$
(switch type)	st	$::=$	$\texttt{Int} \mid \texttt{Tagorboxed}$

Figure 7.6: Abstract syntax for expressions and declarations of U-Bform

TIL translates each expression bound by a `fixval` to separate `let` expressions. It uses the value 0 for variables that would be unbound, and later fills in the correct values for those variables using `writes`. For example, TIL translates the expression

```
let fixval x = {a,b,y,z}
       and y = { ... x  ... }
       and z = {  ... x ... }
...
```

to

```
let val x = {a,b,0,0}
    val y  = { ... x  ... }
    val z = {  ... x ... }
    val _ = x.2 <- y
    val _ = x.3 <- z
...
```

This expression uses the value `0` for `y` and `z` in the record bound to `x`; it then overwrites those fields with the correct values for `y` and `z` (note that record fields are numbered starting from 0).

TIL computes the traceability of a U-Bform variable as follows. If the corresponding variable is bound to a closed term-level function or constructor function, then the traceability is `CODE`. Otherwise,

- If the corresponding B-form variable is a constructor variable, the U-Bform variable is marked as `TRACE`. The datatype representing constructors has a traceable representation.
- If the corresponding B-form variable is a term-level variable, then TIL examines the normal form of the type of the variable. The normal form must be
  - a constructor variable,
  - a projection from a constructor variable (these are introduced by closure conversion),
  - or a primitive constructor.

  In the first two cases, TIL maps the type to a `COMPUTE`. In the last case, TIL maps the primitive constructors `Int` and `Real` to `INT` and `REAL` respectively, and it maps all other primitive constructors to `TRACE`.

The only optimization that TIL does on U-Bform programs is dead-code elimination. This eliminates unnecessary selects of record fields introduced by the translation of switches. It also drops computations for constructors that turn out to be unneeded after the conversion to U-Bform. This occurs, for example, if a constructor is only referenced in the type of a variable whose traceability can be determined at compile time.

Figure 7.7 shows the B-form function `m` from Figure 7.5 after TIL translates it to U-Bform. Each bound variable is annotated with its traceability. Even in this example of a polymorphic function, only the variables `a` and `x4` have traceabilities that are not known until run time. This illustrates an important point: even for polymorphic functions, traceabilities of many variables can be determined at compile time.

TIL translates the `Switch_enumorrec` in the function `m` to a `Switch_tagorboxed`. The nil case returns the value 0, which represents nil. The non-nil case rebinds the variable `l` to itself. It then proceeds to extract the head and tail of the list, and the components of the closure `f`, and call f. Finally, it applies itself to the remainder of the list, and creates the `cons` cell for the result list.

7.4 Conversion to RTL

RTL is a register-transfer language similar to ALPHA or other RISC-style assembly language. An RTL program consists of a list of assembly-language procedures, a list of static data, and a list of global variables. Each procedure has local variables (pseudo-registers). However, there is no notion of local data (that is, there is no stack allocation of data structures). Procedure arguments and results are passed "by value".

RTL provides "heavy-weight" function call and return mechanisms, and a form of inter-procedural goto for implementing exceptions. The call stack remains implicit.

To propagate information to the garbage collector, TIL annotates global and local variables with traceabilities. I extended the traceabilities from the U-Bform level to include locatives, which denote pointers into the middle of objects, and labels, which denote pointers to locations outside the heap.

The translation to RTL does three things: first, it decides the representation of U-Bform variables at the RTL level, second, it converts some constant expressions to RTL data, third, it translates U-Bform expressions to sequences of RTL instructions.

7.4.1 Representation of U-Bform variables

The translation to RTL must decide upon the representation of U-Bform variables. A U-Bform variable can be represented at the RTL level in one of three ways:

- as a local variable,
- as a compile-time constant that is loaded into a register when needed (for example, as a label, integer, or floating-point number),
- or as a global variable (a memory location).

The representations are ordered from most preferred to least preferred. A local variable is preferred above all, because local variables are usually implemented as machine registers. A compile-time constant is preferred above a global variable, because on ALPHA workstations programs can load compile-time constants in one instruction using the *gp* register, but they

```
val empty_cenv:TRACE = {}
fix closed_map =
 λtenv:TRACE,t1:TRACE,t2:TRACE,env:TRACE.
    let fix closed_map' =
          λtenv:TRACE,env:TRACE,f:TRACE.
            let val t1:TRACE = #0 tenv
                val t2:TRACE = #1 tenv
                fix m =
                 λtenv:TRACE,env:TRACE,l:TRACE.
                   let val x1:TRACE =
                      Switch_tagorboxed l
                      of 0 => 0
                       | 1 =>
                        let val l = l
                            val f:TRACE = #0 env
                            val m':TRACE = #1 env
                            val a:COMPUTE tenv.0 = #0 l
                            val r:TRACE = #1 l
                            val f_code:CODE = #0 f
                            val f_tenv:TRACE = #1 f
                            val f_env:TRACE = #2 f
                            val x4:COMPUTE tenv.1 = f_code(f_tenv,f_env,a)
                            val m'_code:CODE = #0 m'
                            val m'_tenv:TRACE = #1 m'
                            val m'_env:TRACE = #2 m'
                            val x5:TRACE = m'_code(m'_tenv,m'_env,r)
                            val x6:TRACE = {x4,x5}
                        in  x6
                        end
                   in x1
                   end
                val tenv':TRACE = {t1,t2}
                val clo:TRACE = {m,tenv',0}
                val env':TRACE = {f,clo}
                val _ = clo.2<-env'
            in  clo
            end
        val tenv':TRACE = {t1,t2}
        val map':TRACE = {closed_map',tenv',0}
    in  map'
    end
val map:TRACE = {closed_map,empty_cenv,0}
```

Figure 7.7: Map function after conversion to U-Bform

require two or more instructions to load global variables (one instruction to load the address of the variable using the *gp* register, another instruction to fetch the actual value).

The algorithm for deciding which representation to use is:

- If a U-Bform variable is a top-level variable, then
  - If it is bound to a constant expression (as defined in the following section), then represent the variable as a compile-time constant.
  - Otherwise represent the variable as global variable.
- Otherwise, if a U-Bform variable is not a top-level variable, then represent as a local variable.

7.4.2 Recognition of constant expressions

The translation to RTL also converts some U-Bform expressions to RTL data. For example, it tries to layout records and arrays at compile time. This has two advantages. First, it makes programs smaller. For records, the code to construct the record is always bigger than the data for the record. Second, it makes programs faster, by recognizing constant expressions in inner program loops. (Note, however, that hoisting constant expressions also moves constant expressions from inner loops).

`Write` expressions complicate the conversion of U-Bform expressions to RTL data, because a record that is created from constant values may be modified to contain a non-constant value later. For example, consider the following function:

```
fix f x =
   let val a = {100,200}
   in a.1 <- x;
      x
   end
```

the variable `a` is a bound to a record created from constants. However, the record is modified immediately.

In general, this means that we cannot just convert constant records to U-Bform data, because the different creations of a constant record may be distinguished from each other. For example, `f 5` $\neq$ `f 6`.

To simplify the presentation, I will first discuss the recognition of constant expressions assuming that the `write` construct is not present in the language. I will then discuss the extensions needed to handle the `write` construct.

Definition 8 *A U-Bform expression can be converted to RTL data if it is*

- *an integer, a floating point number, or a string,*
- *or it is a record expression all of whose free variables are bound to expressions that can be converted to RTL data,*

- *or it is an array creation expression bound at the top-level, all of whose free variables are bound to expressions that can be converted to RTL data.*

Array expressions at the top-level will be evaluated either once or not at all; thus we can allocate those arrays as static data.

It is straightforward to map the U-Bform expressions to RTL data. U-Bform integers and floating point numbers map to RTL integers and floating point numbers. Strings, record expressions, and array creation expressions map to RTL labels; the labels are addresses of appropriately initialized memory locations.

Handling `write`'s

The translation from B-form to U-Bform uses writes in limited ways to construct recursive data structures. It produces U-Bform programs that satisfy three conditions:

- The target of a `write` must be a variable that is bound to a record expression.
- A field of a record is the target of only one write expression.
- If a field of a record is the target of a write expression, then the field cannot be read between the time when the record is allocated and the write occurs.

Using the first restriction, we can handle `writes` by requiring that the free variables in record expressions and array creation expressions not be the targets of `writes`. This requirement, however, is too severe, because it prevents us from converting recursive data structures to RTL data. For example, consider code that creates a closure for a recursive function.

```
val tenv':TRACE = {t1,t2}
val clo:TRACE = {m,tenv',0}
val env':TRACE = {f,clo}
val _ = clo.2<-env'
```

The records for `clo` and `env'` can be allocated at compile time; the write simply modifies a constant record to point to another constant record.

Note that the second and third restrictions ensure that allocating `clo` and `env'` at compile time is correct, even if the code is evaluated repeatedly. On the second evaluation of the code, field 2 of `clo` will contain the address of `env`, not 0. However, the second restriction ensures that field 2 of `clo` is overwritten with exactly the same value, and the third restriction ensures that field 2 of `clo` can never be read before the `write` occurs. Thus, there is no way to tell that the second field has been overwritten already.

The requirement that we want is that constant data should point only to other constant data. In particular, it should never point to heap-allocated data. The following definition captures this requirement by ensuring that the free variables of a write expression must be bound to constant data.

Let S be the set of variables bound to expressions that will be converted to RTL data. Define S as the largest subset of the variables bound by a program such that:

- each variable is bound to an integer, a floating point number, or a string,
- or it is bound to a record expression all of whose free variables are in S
- or it is bound to an array creation expression bound at the top-level, all of whose free variables are in S.
- and, if $x \in S$ is the target of a write expression e, then all the free variables of e are in S.

The correctness of the conversion of expressions in S to RTL data is justified as follows. Any `write` that modifies a record in S will overwrite the field of the record with constant data. Because of the restrictions on the way `write`s are used, we cannot observe the effect of overwriting the field.

Note that S is a fixed point; we can compute it by assuming that all variables are in S, and then removing variables that do not satisfy the preceding requirements.

7.4.3 Translation to machine code

Finally, the translation to RTL chooses sequences of assembly language instructions that implement higher-level U-Bform constructs. In this section, I sketch the translation. I omit details where possible, because this translation is well understood for languages such as SML.

The translation is divided into three functions: $\mathrm{t}_{\mathrm{value}}$, which translates values, $\mathrm{t}_{\mathrm{exp}}$, which translates expressions, and $\mathrm{t}_{\mathrm{decl}}$, which translates declarations. The function $\mathrm{t}_{\mathrm{value}}(v)$ takes as an argument a value v to translate and produces a local variable as its result. The local variable will contain the machine representation of v. The translation of expressions, $\mathrm{t}_{\mathrm{exp}}(e, r)$, takes as arguments an expression e to translate and a local variable r that will contain the result of evaluating e. Likewise, the function $\mathrm{t}_{\mathrm{decl}}(d, r)$ takes as arguments a declaration d to translate and a local variable r that will contain the result of evaluating d.

Translating values

The translation $\mathrm{t}_{\mathrm{value}}(v)$ is defined as follows. If v is

- a constant, then create a new local variable r, generate code to load v into r, and return r.
- variable x, then the translation depends on the machine representation of x. If the representation is
  - a local variable r, then return r.
  - a global variable, then create two new local variables a and r. Load a with the address of the global variable, then use a to load r with the contents of the global variable. Finally, return r.
  - a compile-time constant, then create a new local variable r, generate code to load the constant into r, and return r

Translating expressions

The translation of expressions, $\mathrm{t}_{\mathrm{exp}}(e, r)$, is defined as follows. If e is

- a value v, then generate code to load v into the local variable r.
- a string s, then generate code to load the address of s into the local variable r.
- a function call $f(v_1, \ldots v_n)$, first use $\mathrm{t}_{\mathrm{value}}$ to generate code to load $v_1 \ldots v_n$ into local variables $r_1 \ldots r_n$.

  If f is bound to a function, then generate code to do the procedure call:

  ```
  r <- call f(r1, ... rn)
  ```

  Otherwise, use $\mathrm{t}_{\mathrm{value}}$ to generate code to load f into a local variable f' and then generate code to do an indirect procedure call:

  ```
  r <- call_indirect f'(r1, ... rn)
  ```

  Recall that procedure arguments and results are passed by value in RTL.

- a primitive operator $p_1(v)$ of one argument, then use $\mathrm{t}_{\mathrm{value}}$ to generate code to load v into a local variable r_1. Next, generate code to compute $p_1(r_1)$ and store the result in r.

  For example, translate `cos`(x), where x is stored in a local variable r_1, to the RTL instruction

  ```
  r <- cos r1
  ```

- $p_2(v_1, v_2)$ where p_2 is a primitive operator of two arguments, then use $\mathrm{t}_{\mathrm{value}}$ to generate code to load v_1 and v_2 into local variables. Then, generate code to compute p_2 applied to those local variables.

  For example, translate `plusi`(x, y), where x and y are stored in local variables r_1 and r_2 respectively, to

  ```
  r <- r1+r2
  trapb     ; check for overflow
  ```

- `update`(ar, v_1, v_2, v_3), then use $\mathrm{t}_{\mathrm{value}}$ to generate code to load v_1, v_2, and v_3 into local variables. Then generate code to compute the address of element v_2 of v_1 into a new local variable t_1. The computation depends on whether `ar` is an integer array, a floating point array, or a pointer array. Finally, generate code to update the memory location at t_1 with the value v_3 and to initialize r to 0.
- `record`$(v_1 \ldots v_n)$, then

  - Use t_{value} to generate code that loads $v_1 \ldots v_n$ into local variables $r_1 \ldots r_n$.
  - Next, generate code that checks whether enough space is left in the heap to construct the record.
  - Finally, generate code that constructs the record.

  Note that generating code for record construction is complicated by the fact that the header word for a record may need to be constructed *at run time* if one of the fields of the record has a `COMPUTE` representation. The header word for a record contains a bitmask that describes which fields must be traced by the garbage collector.

  If a field has a `COMPUTE` representation, then the bit in the header word for the field must be set at run time. Figure 7.8 illustrates the complications that result. This figure shows the code for the U-Bform function `m` after it has been translated to RTL. After the call to `m'`, the cons cell for the result record is constructed. The first field of this cons cell has a `COMPUTE` representation. The tag construction takes five instructions:

  - First, we load the static part of the tag into a local variable `static_tag`.
  - Next, we load the constructor from the `COMPUTE` into the register `t2`.
  - Then, we set the register `tmp` to 1 if the constructor is traceable, and 0 if it is an integer. We do not have to check for the `Real` constructor, because floating point numbers are always boxed when they are stored in records.
  - Finally, we shift `tmp` to the appropriate bit position and store the result in `dynamic_tag`. We then or the dynamic and start parts of tag together.

- a `switch`, then generate code that tests each arm of the switch sequentially until the code finds an arm whose condition matches. Translate each arm so that the result of evaluating the arm is stored in r, the result variable for the switch.

  For example, translate

  ```
  switch (a)
  0 : 5
  1 : 6
  2 : 7
  ```

  to

  ```
  if a==0 then
  r <- 5
  else if a=1 then
  r <- 6
  else if a=2 then
  r <- 7
  ```

  Note that it is easy to get the declarations for the arms to store their results into r. We simply pass r to t_{decl} when generating code for each arm.

Note also that the translation of large switches is poor. Larges switches should be translated to jump tables or code to do a binary search instead of code that does a sequential series of tests.

Translating declarations

Define the translation of declarations, $\mathrm{t}_{\mathrm{decl}}(d, r)$, as follows. If d is

- a value, then generate code to load the value into the result register r.
- a `let`-bound expression of the form `let` x = e in d', then allocate a new local variable r'. Apply $\mathrm{t}_{\mathrm{exp}}$ to (e, r') to translate e. Apply $\mathrm{t}_{\mathrm{decl}}$ to (d', r) to translate the body of the declaration.
- a function definition `let fix` $x_1 = f_1 \ldots x_n = f_n$ `in` d', then translate each function to an RTL procedure. (Recall that each function must be closed, except for occurrences of top-level variables.)

We also have to translate the declarations for exceptions. There are generally two ways to implement exceptions in a stack-like language such as RTL. The first method is "stack-unwinding". When an exception handler is installed, we place a marker on the stack. When an exception is raised, we unwind the stack until we find a marker. We then invoke the exception handler. The other way is to save enough context for each exception handler that we can "pop" a large number of stack frames at once and jump into the frame containing the exception handler.

With the former method, we can set an exception handler in constant time using only a few instructions. However, invoking an exception handler may take time proportional to the size of the stack. If N is the maximum depth of the stack, creating and invoking an exception handler may take $O(N)$ time. With the latter method, we can create and invoke an exception handler in constant time. Thus, the latter method is more asymptotically efficient. Because the latter method is more efficient, we use that method.

To save the context for an exception handler, we generate an *exception closure* when we enter an exception handler. These closures are allocated on the heap, although it would be more efficient to allocate them on the stack. As noted earlier, however, there is no support for stack-allocated data structures in RTL.

Exception closures have a standard format:

- the address of the handler,
- the stack pointer to restore,
- the previous exception handler closure,
- the values of local variables used by the handler.

When an exception is raised, we

- extract the location of the handler from the current exception closure,

- move the value assocated with the exception to a dedicated exception argument register,
- do an interprocedural goto to the handler.

When we enter an exception handler, we extract the values of the local variables used by the handler.

7.5 Register allocation and assembly

Before doing register allocation, TIL converts RTL programs to ALPHA assembly language with extensions similar to those for RTL. Then TIL does conventional graph-coloring register allocation to allocate physical registers for the pseudo-registers. It also generates tables describing layout and traceabilities of variables for each stack frame. Finally, TIL generates actual Alpha assembly language and invokes the system assembler, which does instruction scheduling and a creates a standard object file.

7.6 Conclusion

In this chapter, I have described the translation of B-form programs to ALPHA machine code. In this translation, I adapted existing compiler technology to translate B-form programs in order to produce good code for first-order monomorphic programs. The translation uses a simple closure conversion strategy. It tries to choose efficient representations of B-form variables: it converts some variables into "global" variables and it recognizes variables bound to constant expressions and converts them to labels. It uses conventional graph-coloring register allocation and supports a split caller/callee save register convention. Finally, it uses the system assembler to do instruction scheduling.

```
m{args = ([tenv : TRACE,env : TRACE,l : TRACE],[]),
  retaddr = (retreg : LABEL),
  result = ([resultreg : TRACE],[])}
{
start:
  beq l, nilcase
  ldl f : TRACE , 0(env)
  ldl m' : TRACE , 4(env)
  ldl a : COMPUTE tenv.0 , 0(l)
  ldl r : TRACE , 4(l)
  ldl f_code : CODE , 0(f)
  ldl f_tenv : TRACE , 4(f)
  ldl f_env : TRACE , 8(f)
call1:
  x4 : COMPUTE tenv.1 <- call f_code(f_tenv,f_env,a)
  ldl m'_code : CODE , 0(m')
  ldl m'_tenv : TRACE, 4(m')
  ldl m'_env : TRACE , 8(m')
call2:
  x5 <- call m'_code(m'_tenv,m'_env,r)
  needgc  3
  li 268435472,static_tag : INT  ; load static portion of tag
  ldl t2 : TRACE , 4(tenv)       ; load type variable
  li 0,tmp : INT                 ; set tag bit to 0
  cmvune t2, 1 , tmp             ; if t2 ≠ 0 (the int constructor),
                                 ; set tag bit to 1
  sll tmp, 3 , dynamic_tag : INT ; shift tag bit into position
  or dynamic_tag, static_tag,  tag : INT  ; combine tag parts
  stl dynamic_tag , 0(heapptr)   ; store tag
  stl x4, 4(heapptr)             ; store first word in record
  stl x5, 8(heapptr)             ; store second word in record
  addl heapptr , 4 , resultreg : TRACE
  addl heapptr , 12 , heapptr
  br exit
nilcase:
  li 0, resultreg
exit:
  return retaddr
}
```

Figure 7.8: Code for the function m after conversion to RTL

Part III

Evaluation

In the second part of the thesis, I described an approach to compiling SML programs based on two ideas:

- a pay-as-you-go compilation strategy, where programmers pay for advanced language features only when they use them.
- applying optimizations that are known to improve loops in imperative languages such as C and Fortran to recursive SML functions.

In this part of the thesis, I test whether these ideas improve performance.

In Chapter 8, I provide evidence that a pay-as-you-go compilation strategy is an effective strategy for compiling SML programs. I establish that the TIL compiler produces competitive code for an SML compiler by comparing it to the SML/NJ compiler, a widely used reference compiler for SML. In fact, TIL produces code that is not only competitive — it is often much better than code produced by the SML/NJ compiler. I also discuss the many differences between the two compilers that make it difficult to explain conclusively why TIL's code is so much better.

In Chapter 9, I measure the effect of the loop optimizations described in Chapter 6 on program performance. First, I show that the optimizations reduce execution time by 8% to 87%. The effect of the optimizations varies widely and depends on the benchmark program, so I do not present averages. Second, I measure the effect of each optimization to determine which ones are most important. Finally, I measure how specific the effect of each optimization is to TIL framework.

In Chapter 10, I present some other interesting measurements of the optimizer. First, I measure the effect of optimization on the numbers of polymorphic and higher-order functions. Second, I measure the effect of types on intermediate program size.

Chapter 8

Comparison against the SML/NJ compiler

In this chapter, I demonstrate that the TIL compiler produces good code for an SML compiler by comparing it with the SML/NJ compiler. I compare the performance of code produced by TIL against code produced by SML/NJ in several dimensions: execution time, total heap allocation, physical memory footprint, and the size of the executable.

In Section 8.1, I introduce the benchmarks that I use throughout this part of the thesis. In Section 8.2, I compare performance of these benchmarks when compiled by TIL and when compiled by SML/NJ. The code produced by TIL often performs much better than that produced by SML/NJ. In Section 8.3, I discuss how you might study why the performance of TIL's code is better and the differences between the two compilers that make it difficult to conduct such a study.

8.1 Benchmarks

Table 8.1 describes the benchmark programs, which range in size from 62 lines to about 2000 lines of code. The benchmarks cover a range of application areas including scientific computing, list-processing, systems programming, and compilers. Appel used some of these programs to measure ML performance [7].

The set of benchmark programs that I use in this part of the thesis includes only some of the programs that I used in the first part of the thesis. This is because the TIL compiler does not yet compile the full SML module system (it does not support nested structures or functors). Thus, I could not use CW (the Concurrency Workbench), VLIW (the very-long-instruction-word scheduler), and YACC (the parser generator). It is possible to eliminate all uses of nested structures or functors by rewriting the source code by hand, but these programs were simply too large (from 3200 to 5700 lines) and complex for me to do so. I was able to eliminate these language constructs from Lexgen and Simple, however.

The fact that the TIL compiler does not yet compile the entire SML module system is not a fundamental limitation of the TIL approach. Harper and Stone [38] show how to extend the TIL framework to support modules.

Program	lines	Description
Checksum	241	Checksum fragment from the Foxnet [14], doing 5000 checksums on a 4096-byte array.
FFT	246	Fast fourier transform, multiplying polynomials up to degree 65,536
Knuth-Bendix	618	An implementation of the Knuth-Bendix completion algorithm.
Lexgen	1123	A lexical-analyzer generator [11], processing the lexical description of Standard ML.
Life	146	The game of Life implemented using lists [72].
Matmult	62	Integer matrix multiply, on 200x200 integer arrays.
PIA	2065	The Perspective Inversion Algorithm [94] deciding the location of an object in a perspective video image.
Simple	870	A spherical fluid-dynamics program [31], run for 4 iterations with grid size of 100.

Table 8.1: Benchmark Programs

I compiled whole programs during benchmarking (the programs were single closed modules). I extended the built-in ML types with safe 2-dimensional arrays. The 2-d array operations do bounds checking on each dimension and then use unsafe 1-d array operations. Arrays are stored in column-major order.

8.2 Comparison against SML/NJ

For TIL, I compiled programs with all optimizations enabled, including the loop optimizations. For SML/NJ, I compiled programs using the default optimization settings. I used version 1.08 of the SML/NJ compiler.

TIL always prefixes a set of operations on to each module that it compiles to facilitate optimization. This "inline" prelude contains 2-d array operations, commonly used list functions, and so forth. To provide a fair comparison withSML/NJ, I created separate copies of the benchmark programs for SML/NJ, and placed equivalent "prelude" code at the beginning of each program by hand.

Because TIL creates stand-alone executables, I used the `exportFn` facility of SML/NJ to create stand-alone programs. The `exportFn` function of SML/NJ dumps part of the heap to disk and throws away the interactive system.

I measured execution time on a DEC ALPHA AXP 3000/250 workstation, running OSF/1, version 3.2C, using the UNIX `getrusage` function. For SML/NJ, I started timing after the heap had been reloaded. For TIL, I measured the entire execution time of the process, including load time. I made 5 runs of each program on an unloaded workstation and chose the lowest execution time. Each workstation had 96MBytes of physical memory, so paging was not a factor in the measurements.

I measured total heap allocation by instrumenting the TIL run-time system to count

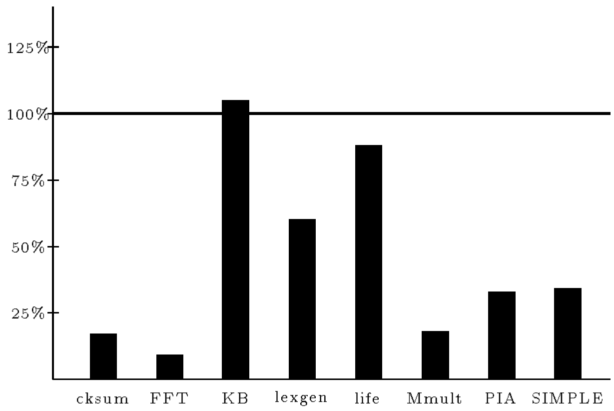

Figure 8.1: TIL Execution Time Relative to SML/NJ

the bytes allocated. I used existing instrumentation in the SML/NJ run-time system. I measured the maximum amount of physical memory during execution using `getrusage`. I used the `size` program to measure the size of executables for TIL. For SML/NJ, I used the `size` program to measure the size of the run-time system and then added the size of the heap created by `exportFn`.

Figures 8.1 through 8.3 present the measurements. For each benchmark, measurements for TIL were normalized to those for SML/NJ and then graphed. SML/NJ performance is the 100% mark on all the graphs.

Figure 8.1 presents relative running times. On average, programs compiled by TIL run 3.3 times faster than programs compiled by SML/NJ. In fact, all programs except `Knuth-Bendix` and `Life` are substantially faster when compiled by TIL. I speculate that we see less of a speed-up for `Knuth-Bendix` and `Life` because they make heavy use of list-processing, which SML/NJ does a good job of compiling.

Figure 8.2 compares the relative amounts of heap allocation. On average, the amount of data heap-allocated by the TIL program is about 17% of the amount allocated by the SML/NJ program. This is not surprising, because TIL uses a stack while SML/NJ allocates frames on the heap.

Figure 8.3 presents the relative maximum amounts of physical memory used. On average, TIL programs use half the memory used by SML/NJ programs. I see that floating-point programs use the least amount of memory relative to comparable SML/NJ programs. I speculate that this is due to the fact that TIL always keeps floating-point values unboxed when stored in arrays.b

TIL stand-alone programs are about half the size of stand-alone heaps and the runtime system of SML/NJ. The difference in size is mostly due to the different sizes of the runtime systems and standard libraries for the two compilers. (TIL's runtime system is about 100K, while SML/NJ's runtime is about 425K.) The program sizes for TIL confirm that generating tables for nearly tag-free garbage collection consumes a modest amount of space, and that the inlining strategy used by TIL produces code of reasonable size.

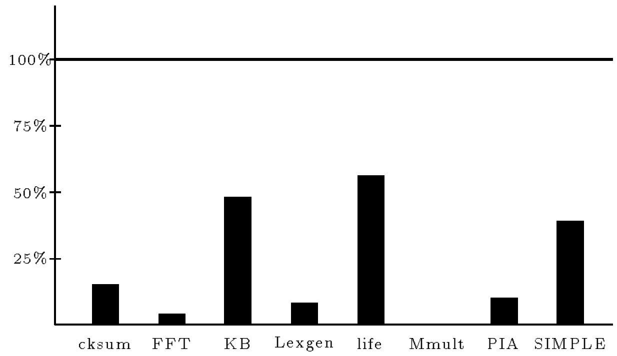

Figure 8.2: TIL Heap Allocation Relative to SML/NJ

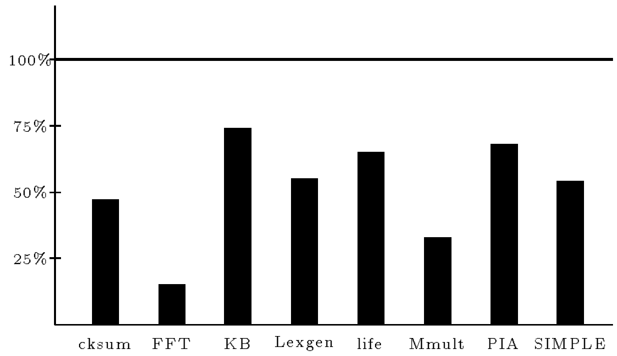

Figure 8.3: TIL Physical Memory Used Relative to SML/NJ

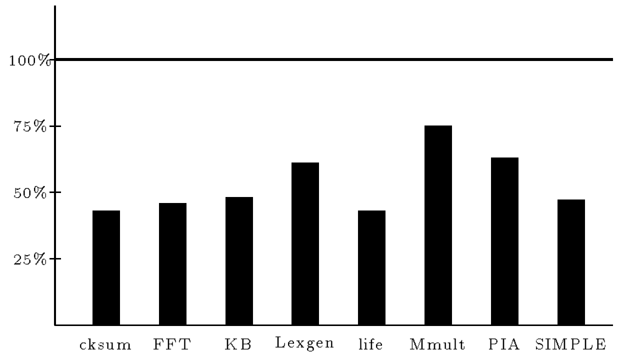

Figure 8.4: TIL Executable Size Relative to SML/NJ

8.3 Further comparison

The previous section leaves an interesting problem: the performance of code compiled by TIL is often much better than the performance of code compiled by SML/NJ. To determine why this is the case, you would have to start with one system and vary each design choice until you have tried all the design choices made by the other system.

This would be time-consuming to do because TIL differs from SML/NJ at nearly every level. The major design differences between TIL and SML/NJ are:

- intensional polymorphism:

  TIL constructs and passes types around at run time, whereas SML/NJ does not.

- optimization:

  TIL does loop optimizations, whereas SML/NJ does not. TIL also uses different algorithms for inlining and uncurrying.

- closure conversion:

  TIL uses a simple closure conversion strategy, whereas SML/NJ uses a sophisticated closure conversion strategy that also implements a split callee/caller-save register convention. Also, TIL's closure converter does not close over variables defined at the top-level (outside functions).

- recognizing constant expressions:

  TIL converts constant expressions, such as records, to their machine data represntation at compile time.

- register allocation:

  TIL uses a conventional graph-coloring register allocator that supports a split callee/caller save register convention for procedure calls, whereas SML/NJ uses a simple register-tracking scheme.

- allocation of procedure activation records:

  TIL uses stack allocation of procedure activation records, whereas SML/NJ uses heap allocation of procedure activation records.

- instruction scheduling

  TIL uses the system assembler, whereas SML/NJ uses their own instruction scheduler.

- garbage collection

  TIL uses a tag-free garbage collector, whereas SML/NJ uses a garbage collector that requires tagging.

To try SML/NJ's design choices in TIL, you would have to rewrite the following phases of TIL:

- the type-directed optimizations that improve data representations
- the inlining and uncurrying optimizations
- the closure converter
- the register allocator
- the instruction scheduler
- the garbage collector

This is an enormous amount of work — it is nearly equivalent to rewriting TIL.

8.4 Conclusion

In this chapter, I have demonstrated that TIL, with all optimizations including the loop optimizations enabled, produces good code for an SML compiler. On average, its code is 3.3 times faster than code produced by the SML/NJ compiler. Thus, TIL is a reasonable system in which to measure the effects of the loop optimizations. I have explained why it would be difficult to fully understand why TIL's code is so much faster that SML/NJ's code. Finally, I also have described the benchmarks used in this part of the thesis.

Chapter 9

Effect of loop optimizations

In this chapter, I demonstrate my claim that applying optimizations that improve loops to SML programs improves performance. In Section 9.1, I measure the combined effect of the optimizations that I describe in Chapter 6 on performance. It is important to enable all of these optimizations at once because they can interact in ways that improve each other. In Section 9.2, I study the effect of individual optimizations from Chapter 6 on performance. Finally, in Section 9.3, I examine how specific these results are to the typed intermediate language framework used by TIL.

9.1 Combined effect of loop optimizations

In this section, I measure the combined effect of the "loop" optimizations that I describe in Chapter 6 on performance. These optimizations are CSE, invariant removal, hoisting, comparison elimination, and redundant switch elimination. For the sake of conciseness, I will refer to these optimization as "the loop optimizations" throughout this Chapter, although of course there are many other optimizations to improve loops.

Figures 9.1 through 9.5 show the effect of adding the loop optimizations to the TIL optimizer. The 100% mark on each graph is the performance of each benchmark when I compiled it using just the TIL optimizer (as described in Chapter 5). The columns show the performance when I compiled the benchmarks using the TIL optimizer *and* the loop optimizations. Table 9.1 presents the data in the graphs in tabular form; each number is the ratio of performance with the loop optimiz ations to performance without the loop optimizations. Appendix B contains the original data that I used to calculate the ratios.

I use the same metrics that I used in Chapter 8 to compare performance: execution time, amount of data heap allocated, amount of data copied by the generational garbage collector, maximum physical memory used by a program while it ran, and the size of the executable. I ran each program 100 times and used the arithmetic mean of the measurements for each metric. Appendix B presents the means and standard deviations. Only execution time and maximum physical memory used varied across runs: the standard error for those measurements was less than one percent (at a 99% confidence level).

The variance in performance across benchmarks is high, so I will not summarize performance using medians or means. Table 9.1 contains medians and geometric means if you are interested in them.

Figure 9.1 shows the effect of the loop optimizations on execution time. The loop optimizations reduce execution time by 8% to 87%. They never increase execution time. Their effect on execution time, not surprisingly, is greatest on scientific programs, such as FFT, Matmult, and SIMPLE.

Figure 9.2 shows the effect of the loop optimizations on the amount of heap-allocated data. For Lexgen, Matmult, and Life they increase heap allocation by 4 to 18%. For FFT, however, they reduce it by 98%. The loop optimizations may increase heap allocation if they move heap-allocating expressions to points in programs where the expressions are executed more frequently than their original location.

Figure 9.3 shows the effect of the loop optimizations on the amount of data copied by the generational garbage collector. For Lexgen and Simple, they increase the amount of data copied by 1 and 4%, respectively. This is presumably because the loop optimizations are not safe for space [7]. For other programs, such as Checksum, Life, and FFT, they nearly eliminate all copying of data. They have no effect on Matmult, because the generational collector does not copy any data when Matmult runs.

An optimization is safe for space if it does not cause a program to keep data live when the data would be dead in the original program. For example, invariant removal is not safe for space. It may cause a large data structure to live across iterations, thus causing more data to be copied by the generational collector.

Figure 9.4 shows the effect of the loop optimizations on the total amount of physical memory used by programs. The loop optimizations reduce the amount of physical memory used by programs, by reducing the amount of data copied by the generational collector. FFT, Life, and Checksum use 9% to 94% less memory. Even though the loop optimizations are not safe for space, memory usage increases only for SIMPLE, and only by 2%.

Finally, Figure 9.5 shows the effect of the loop optimizations on program executable sizes. The optimizations always make program executables smaller and reduce program sizes by 3% to 26%.

To summarize, my experimental measurements show that for these benchmarks the loop optimizations:

- reduce execution time always,
- are important for scientific programs,
- may increase or decrease heap allocation,
- may increase physical memory usage negligibly but sometimes reduce it dramatically,
- and reduce sizes of program executables always.

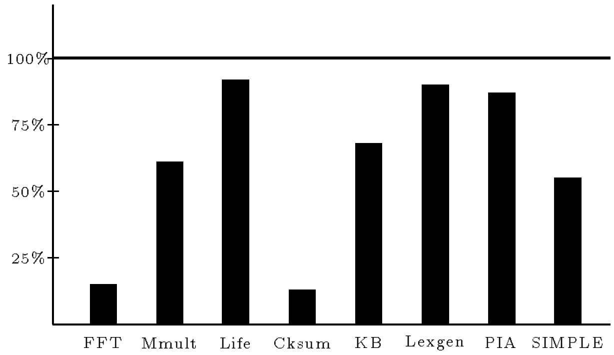

Figure 9.1: Effect of loop optimizations on total time

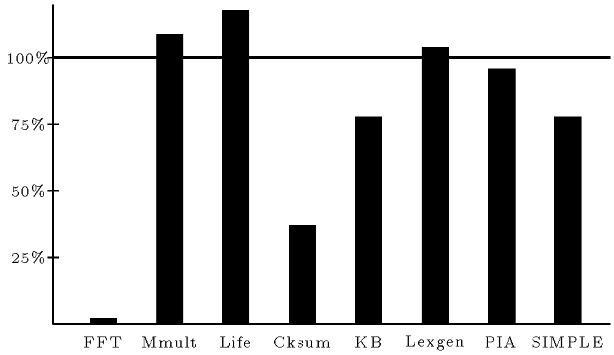

Figure 9.2: Effect of loop optimizations on heap allocation

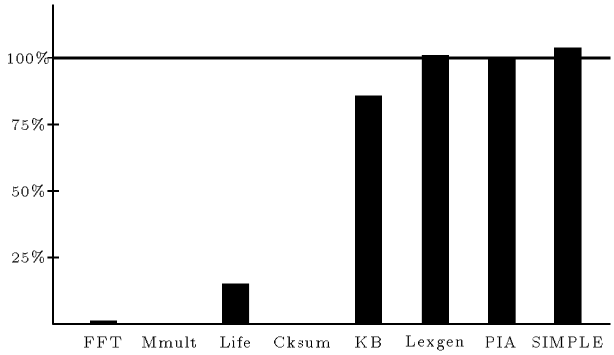

Figure 9.3: Effect of loop optimizations on data copied by the garbage collector

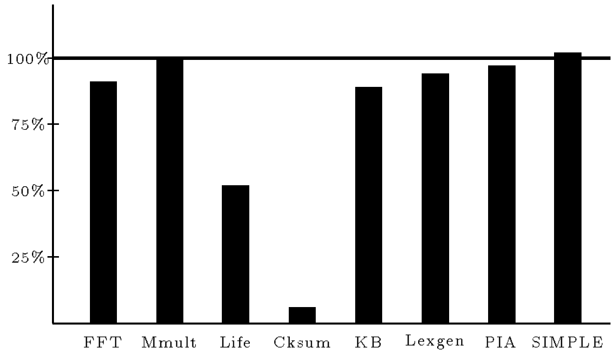

Figure 9.4: Effect of loop optimizations on physical memory

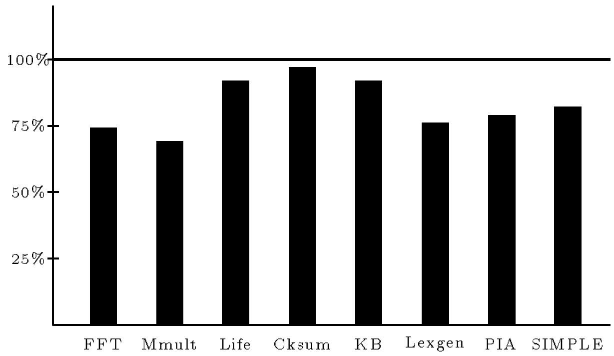

Figure 9.5: Effect of loop optimizations on code size

Program	Time	Heap alloc.	GC copying	Phys mem.	Code size
checksum	0.13	0.37	0.00	0.06	0.97
fft	0.15	0.02	0.01	0.91	0.74
kb	0.68	0.78	0.86	0.89	0.92
life	0.92	1.18	0.15	0.52	0.92
lexgen	0.90	1.04	1.01	0.94	0.76
matmult	0.61	1.09	-	1.00	0.95
pia	0.87	0.96	1.00	0.97	0.79
simple	0.55	0.78	1.04	1.02	0.82
Median	0.65	0.87	0.86	0.93	0.87
Geo. mean	0.49	0.53	0.33	0.63	0.85

Table 9.1: Tabular comparison of performance with and without loop optimizations

9.2 Effects of individual optimizations

In the previous section, I demonstrated that the loop optimizations can improve the performance of SML programs significantly. In this section, I investigate which optimizations are most important to improving performance. I do this by measuring the effect on performance of each loop optimization when it is added to the TIL optimizer.

I was unable to measure the effect of invariant removal on the performance of PIA and SIMPLE because TIL did not generate garbage collection tables for large stack frames properly. TIL uses an escape mechanism to handle the rare case of stack frames with more than 64 elements, but this mechanism was not working when I made these measurements.

9.2.1 Effect on execution time

Figure 9.6 shows the average effect on execution time of adding each loop optimization to the TIL optimizer. The 100% mark represents execution time when I compiled the benchmark programs using just the TIL optimizer. Each column shows execution time when I compiled the benchmark programs using the TIL optimizer and a particular optimization. I used the geometric mean to average the percentages across benchmarks. We see that adding CSE, adding hoisting or adding invariant removal reduces execution time by nearly a third. Adding comparison elimination reduces execution time by 5%, and adding switch elimination has no effect on execution time at all.

Figures 9.7 through 9.11 shows the effect of adding each loop optimization to the TIL optimizer on each of the benchmark programs. We see that adding comparison elimination has a significant effect only on Matmult and Lexgen. In contrast, adding CSE, hoisting, or invariant removal improves the execution times of most benchmark programs. For scientific programs adding invariant removal improves performance much more than just hoisting. For FFT, adding invariant removal reduces execution time by 28%, but adding hoisting reduces execution time by only 5%. Similarly, for Matmult, adding invariant removal reduces execution time by 17%, but adding hoisting increases execution time by 1%. For non-scientific programs, adding invariant removal or hoisting has nearly the same effect on performance.

9.2.2 Effect on heap allocation

Figure 9.12 shows the average effect on heap allocation of adding each loop optimization to the TIL optimizer. Figures 9.13 through 9.17 show the effect of each optimization on each benchmark. The 100% mark on each graph is the base case of just the TIL optimizer.

We see that comparison elimination and switch elimination have no effect on heap allocation. This is what I expected, because they do not affect language constructs that cause heap allocation. The effect of CSE on heap allocation ranges from a 64% reduction to a 2% increase in heap allocated data. For CSE, most of the reduction in heap allocation is due to the elimination of constructor computations: elimination of constructor computations accounts completely for the reductions seen for Checksum, Knuth-Bendix, and Life, but it does not account for the reductions seen in FFT.

9.2.3 Effect on GC copying

Figure 9.18 shows the average effect on GC copying of adding each loop optimization to the TIL optimizer. Figures 9.19 through 9.23 show the effect of each optimization on each benchmark. Again, the 100% mark on each graph is the base case of just the TIL optimizer. In computing the geometric means for Figure 9.18, I excluded Matmult, because the garbage collector does no copying of data for Matmult.

We see that adding comparison elimination or switch elimination has no effect on GC copying, as we would expect. These optimizations do not affect any language constructs that do heap allocation. Adding CSE, adding hoisting, or adding invariant removal each have nearly the same effect on GC copying. They differ in their effect on FFT primarily, but the GC copying for FFT is insignificant (only 85 Kbytes) even with the base case TIL optimizer.

9.2.4 Effect on physical memory usage

Figure 9.24 shows the average effect on physical memory usage of adding each individual loop optimization to the TIL optimizer. Figures 9.25 through Figures 9.29 show the effect of adding each optimization on each benchmark. Again, the 100% mark on each graph is the base case of just the TIL optimizer.

There is a strong correlation between the effects of optimizations on GC copying and physical memory usage: we see that adding comparison elimination or adding switch elimination has no effect on memory usage and that adding CSE, adding hoisting, or adding invariant removal has almost the same effect on memory usage as adding each does on GC copying. This correlation is not surprising with a generational collector. GC copying determines the amount of data in older generations, and the data in older generations determines the physical memory usage of a program.

9.2.5 Effect on program executable size

Figure 9.30 shows the average effect on program executable size of adding each loop optimization to the TIL optimizer. Figures 9.31 through Figures 9.35 show the effect of adding each optimization on each benchmark. The 100% mark on each graph is the base case of the TIL optimizer.

We see that each optimization has a small effect on program executable size and that no particular optimization stands out. The overall reduction in program executable size that I observed in Section 9.1 appears to be the cumulative result of the small effects of the individual optimizations.

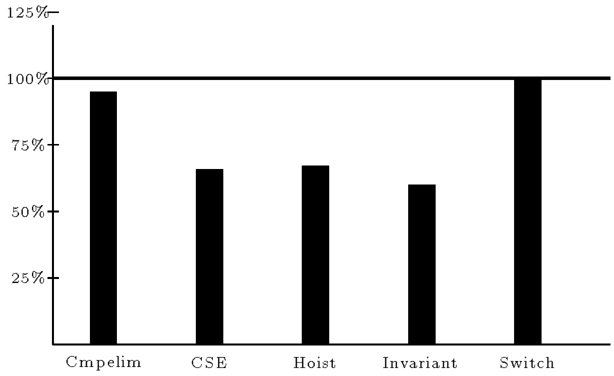

Figure 9.6: Relative execution time (geo. mean)

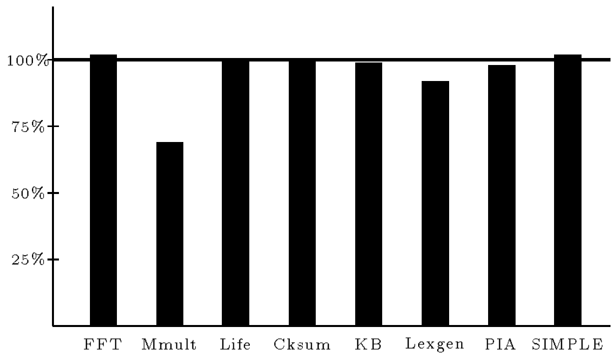

Figure 9.7: Effect of comparison elimination on total time

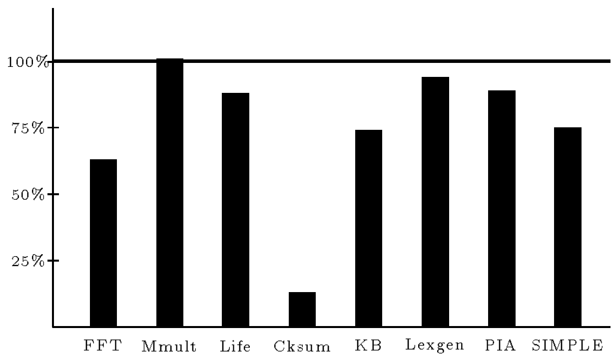

Figure 9.8: Effect of CSE on total time

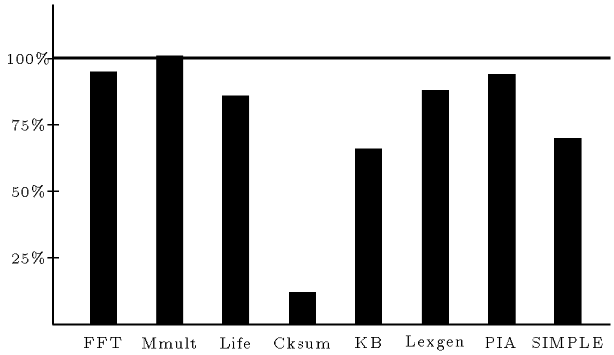

Figure 9.9: Effect of hoisting on total time

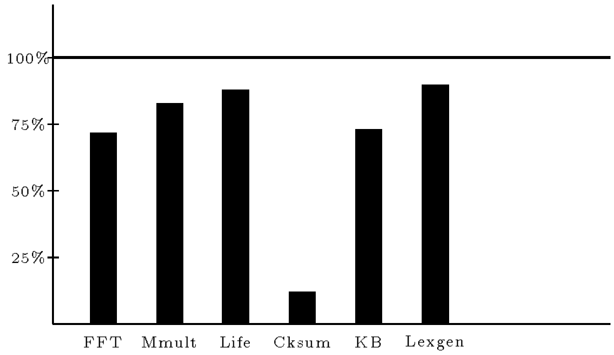

Figure 9.10: Effect of invariant removal on total time

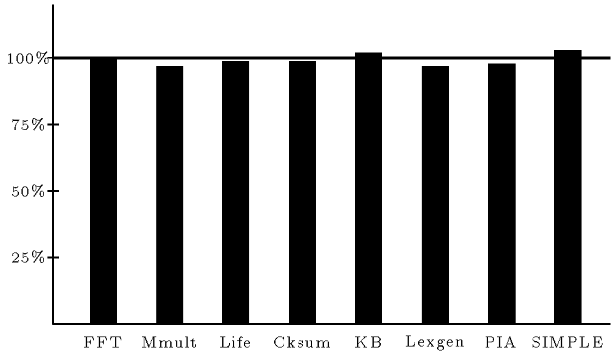

Figure 9.11: Effect of switch on total time

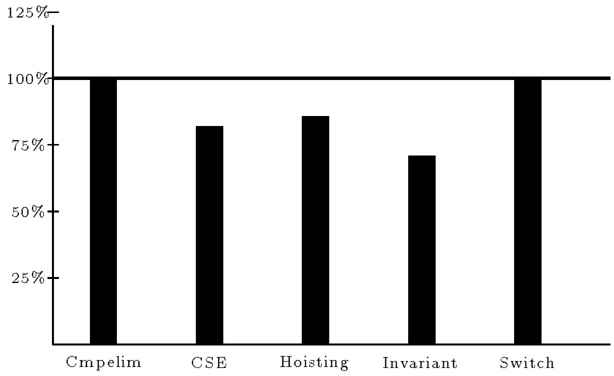

Figure 9.12: Relative heap allocation (geo. mean)

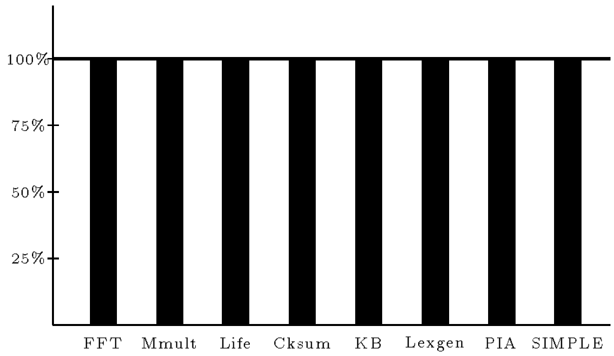

Figure 9.13: Effect of comparison elimination on heap allocation

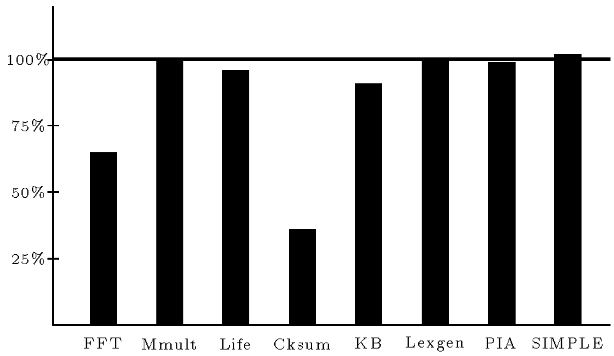

Figure 9.14: Effect of CSE on heap allocation

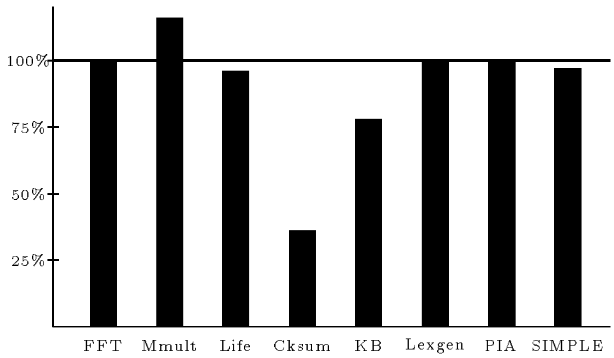

Figure 9.15: Effect of hoisting on heap allocation

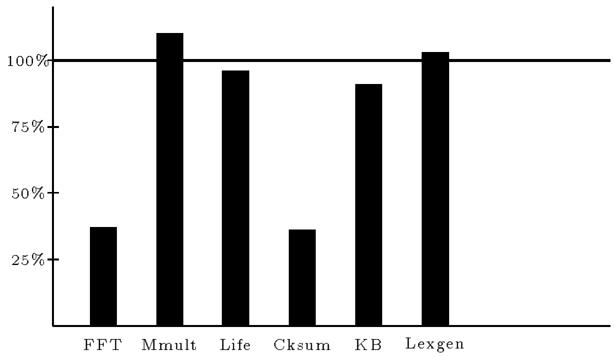

Figure 9.16: Effect of invariant removal on heap allocation

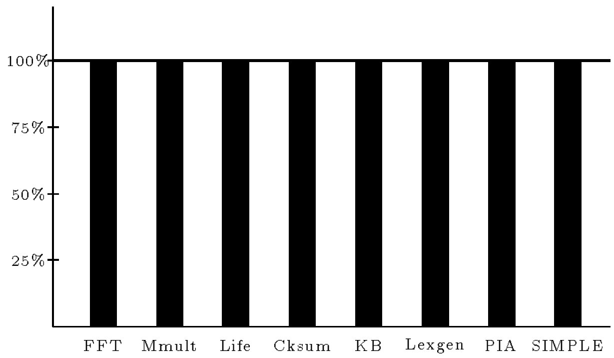

Figure 9.17: Effect of switch on heap allocation

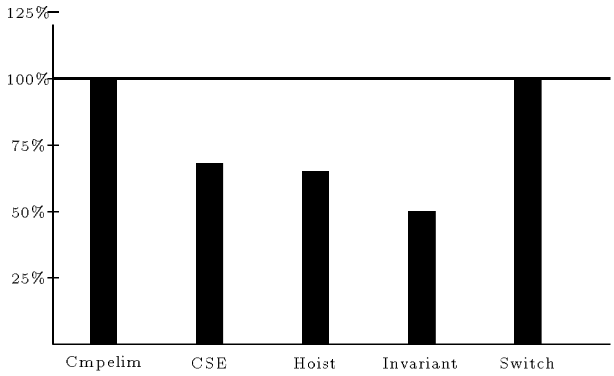

Figure 9.18: Relative GC copying (geo. mean)

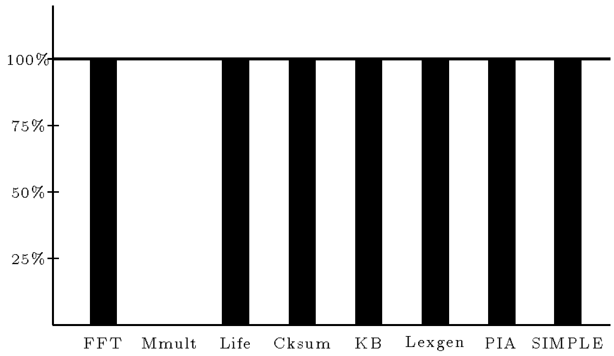

Figure 9.19: Effect of comparison elimination on GC copying

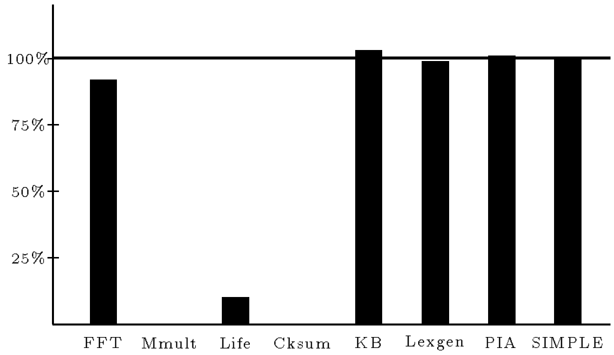

Figure 9.20: Effect of CSE on GC copying

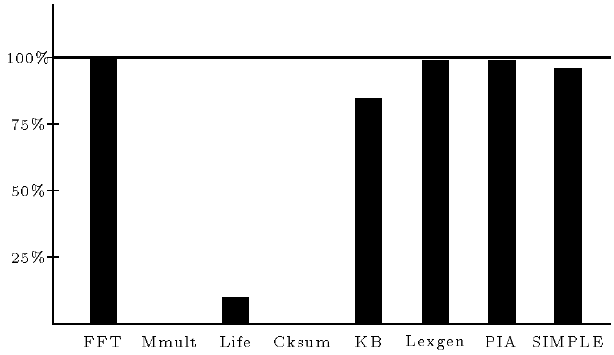

Figure 9.21: Effect of hoisting on GC copying

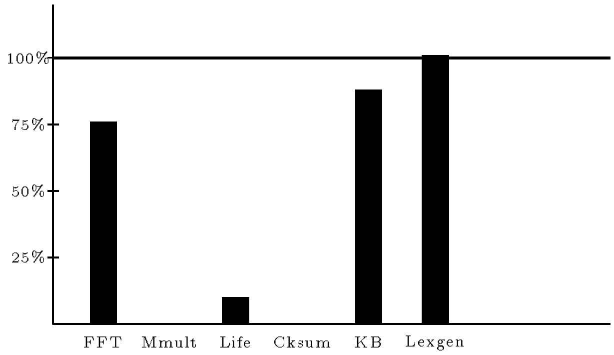

Figure 9.22: Effect of invariant removal on GC copying

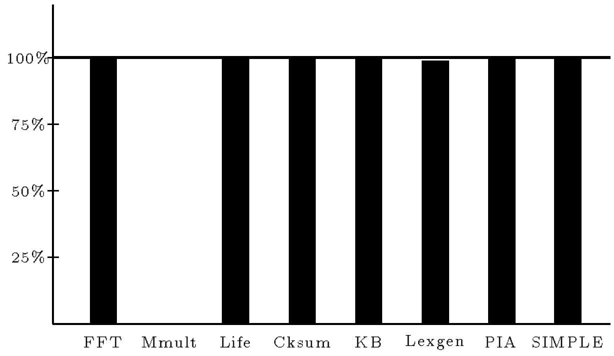

Figure 9.23: Effect of switch on GC copying

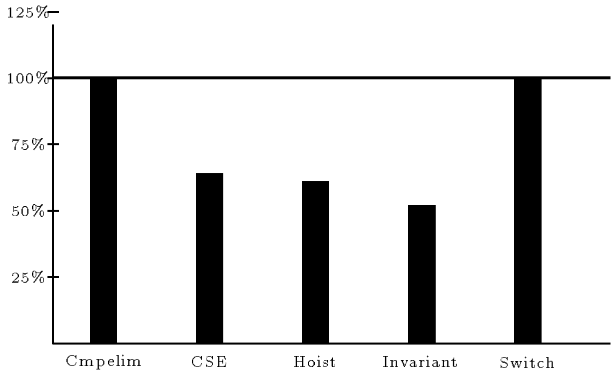

Figure 9.24: Relative phsyical memory usage (geo. mean)

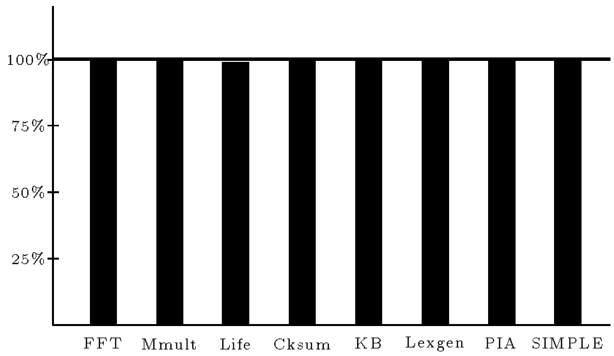

Figure 9.25: Effect of comparison elimination on physical memory usage

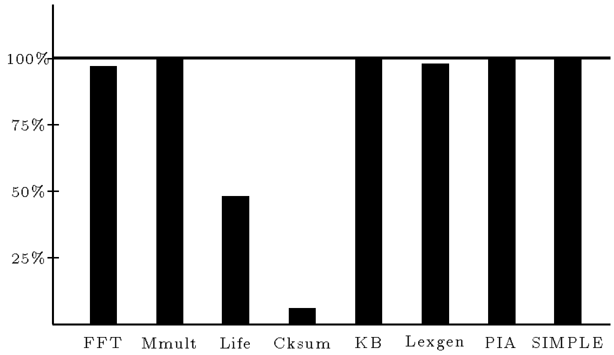

Figure 9.26: Effect of CSE on physical memory usage

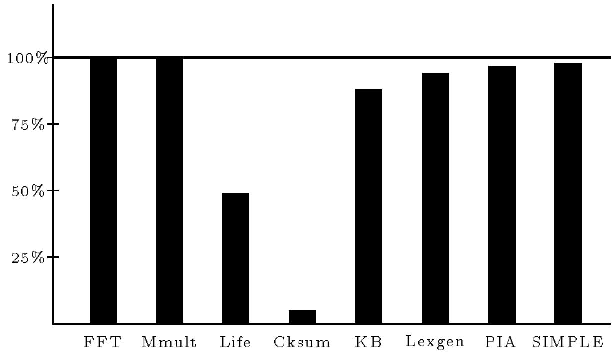

Figure 9.27: Effect of hoisting on physical memory usage

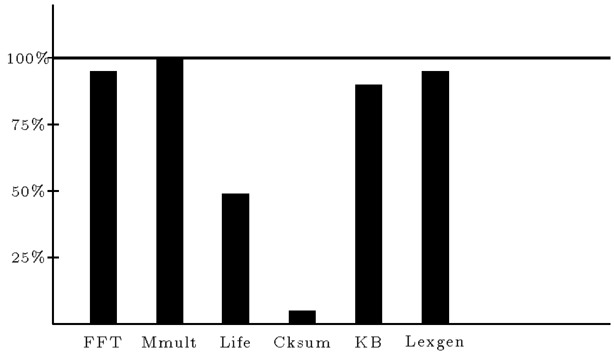

Figure 9.28: Effect of invariant removal on physical memory usage

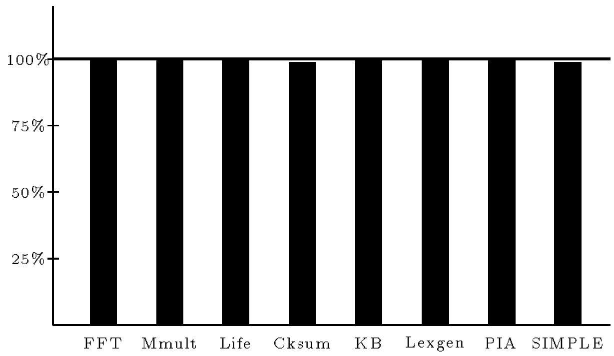

Figure 9.29: Effect of switch on physical memory usage

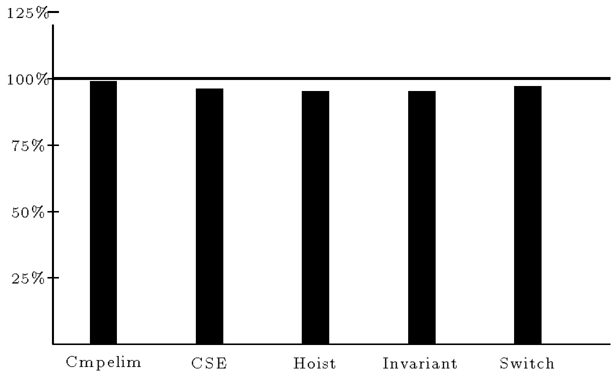

Figure 9.30: Relative executable program size (geo. mean)

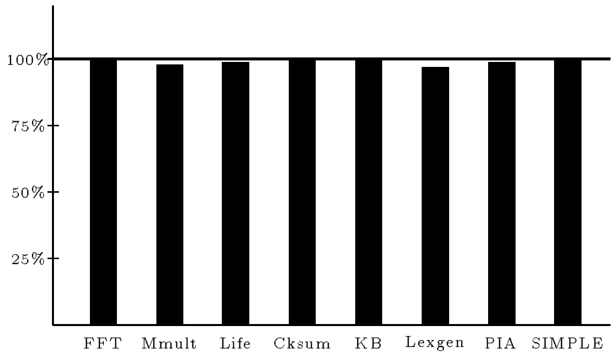

Figure 9.31: Effect of comparison elimination on executable program size

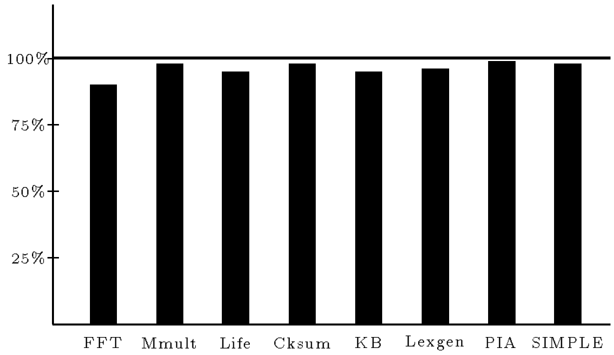

Figure 9.32: Effect of CSE on executable program size

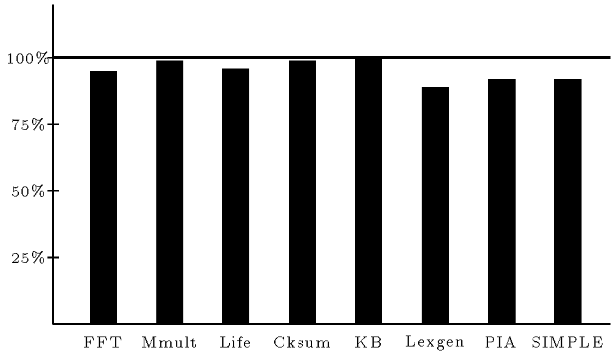

Figure 9.33: Effect of hoisting on executable program size

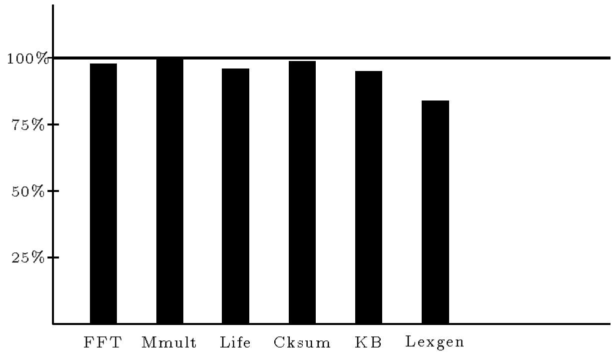

Figure 9.34: Effect of invariant removal on executable program size

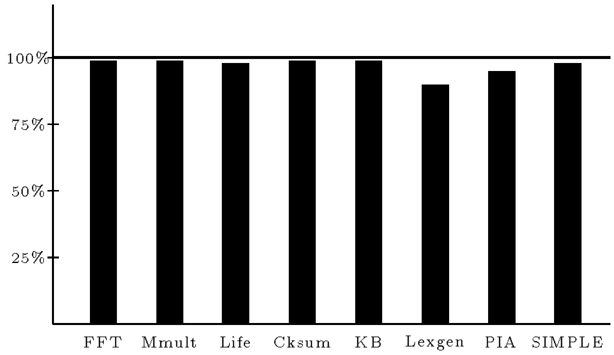

Figure 9.35: Effect of switch on executable program size

9.3 Intensional Polymorphism

In Section 9.1, I demonstrated that applying optimizations that improve loops to SML programs can improve the performance of SML programs significantly. In this section, I address how specific this result is to the setting of intensional polymorphism.

Intensional polymorphism adds two costs to programs at run time: the cost of passing types and the cost of type construction. In Chapter 10, I shall show that there are no polymorphic functions left after B-form optimization. Thus, the optimizer eliminates the cost of passing types for the benchmark programs. However, the cost of type construction remains, even though the benchmark programs are monomorphic after B-form optimization. Types cannot just be erased; they are needed for the garbage collector. Thus, some of the benefit of the loop optimizations may be due to reducing the cost of type construction and may be specific to the setting of intensional polymorphism.

To assess the effect of the setting of intensional polymorphism on the effect of the loop optimizations, I performed the following experiment on loop optimizations that affect both constructors and expressions. These are the code motion optimizations: common subexpression elimination, hoisting and invariant removal. First, I measured the effect of the code motion optimizations on performance when they were restricted to optimizing only constructors. Then, I measured the effect of the code motion optimizaitons when they were allowed to optimize constructors and expressions. This allows us to gauge the additional speed-up due to optimizing expressions; this additional speed-up is what might be possible in a more conventional compiler setting.

I did not directly measure the effect of moving only expressions because of the typed setting in which I was working. In particular, in the setting of intensional polymorphism, an expression can only be moved in the scope of its free constructor variables. Thus, if you do not move constructors, you are constraining the movement of expressions in a way that would not occur in an untyped setting.

Figure 9.36 compares the relative improvements in execution time with code motion of constructors against the relative improvements in execution time with code motion of constructors and expressions. The 100% mark represents the performance with no loop optimizations, the hollow bars show performance with only code motion of constructors, and the solid bars show performance with code motion of constructors and expressions.

We see the following interesting results. For Life, Checksum, and Knuth-Bendix, code motion of constructors accounts for nearly all the improvements in execution time that are due to the loop optimizations. For FFT, Matmult, Lexgen, and PIA, the optimization of expressions accounts for the improvements in execution time that are due to the loop optimizations. For SIMPLE, the improvements in execution time are roughly split between optimizing expressions and optimizing constructors.

We can draw the following conclusions from these results. For list-processing programs, the improvements in execution time that are due to the loop optimizations are specific to the setting of intensional polymorphism. For other SML programs, especially scientific programs, however, the improvements in execution time that are due to the loop optimizations are general.

Figure 9.37 compares the relative improvements in heap allocation with code motion

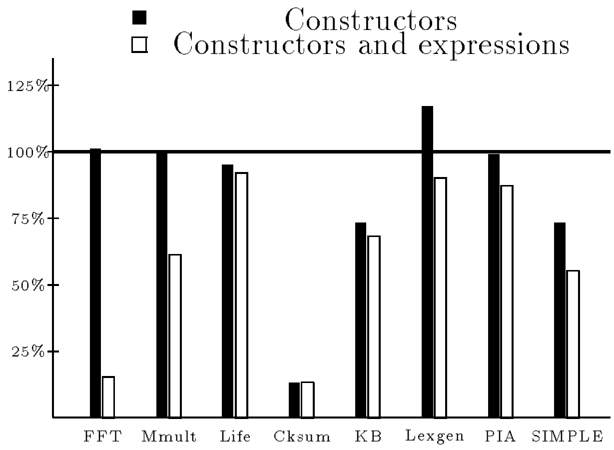

Figure 9.36: Execution time with code motion of constructors compared to execution time with code motion of constructors and expressions.

of constructors against the relative improvements in heap allocation with code motion of constructors and expressions. As before, the 100% mark represents performance with no loop optimizations, the hollow bars are for code motion of constructors, and the solid bars are for code motion of constructors and expressions.

We see the following results. For Checksum and Knuth-Bendix, the large reductions in total heap allocation due to the loop optimizations were due to code motion of constructors almost entirely. For FFT and SIMPLE, the large reductions in heap allocation were due to code motion of expressions almost entirely. For Matmult, Life, Lexgen, and PIA code motion of constructors had little or no effect on heap allocation and code motion of expressions tended to increase the amount of heap allocation.

We can draw the following conclusions about the loop optimizations in a general setting. For most programs, code motion of expressions has no effect on heap allocation or tends to increase heap allocation. For scientific programs, however, code motion of expressions can reduce heap allocation significantly.

9.4 Conclusions

In this chapter, I have demonstrated that applying optimizations that improve loops to SML programs improves performance significantly for my benchmarks. Performance improves across a range of metrics, including execution time and total physical memory used by programs. I studied the effect of individual optimizations, and demonstrated that the code motion optimizations were most important to improving performance. Finally, I studied how specific these results were to the TIL framework. I found that improvements in heap allocation due to the loop optimizations were specific to the TIL framework. Outside the TIL framework, the loop optimizations were likely to have no effect on heap allocation or to increase heap allocation, except for scientific programs. I found that the improvements

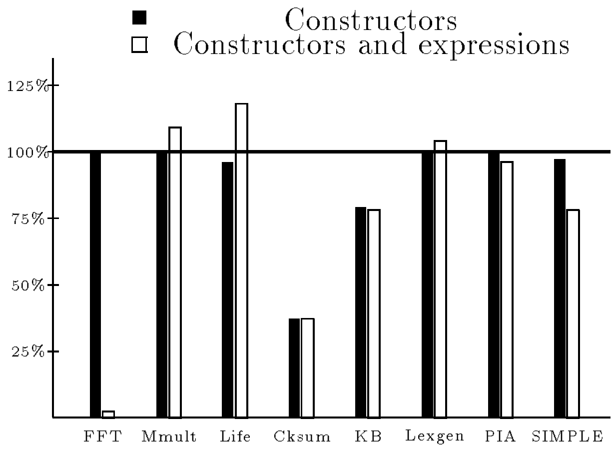

Figure 9.37: Heap allocation with code motion of constructors compared to heap allocation with code motion of constructors and expressions.

in execution time due to the loop optimizations for list-processing programs were specific to the TIL framework, but that improvements in execution time for other kinds of programs were likely to be found in other approaches to compiling SML.

Chapter 10

Other measurements

In this chapter, I present some other interesting measurements of programs compiled by TIL. First, I investigate the effect of optimization on the numbers of polymorphic and higher-order functions when whole programs are compiled. Second, I study the effect of types on intermediate program size. Types may make intermediate programs bigger; this could make compilers that use typed intermediate languages require more memory and be slower than compilers that use untyped intermediate languages.

10.1 Optimization and the numbers of polymorphic or higher-order functions.

In this section, I measure the effect of optimization on the numbers of polymorphic or higher-order functions when whole programs are compiled. To do this, I instrumented TIL to count the number of functions in a program before and after B-form optimization and classify them as (1) polymorphic or (2) higher-order but not polymorphic. All functions bound by `Fixtype` are counted as polymorphic. For functions bound by `Fix`, their types are placed in a normal form and then examined to see if they contain a function type in either the domain or range type. TIL did one round of dead-code elimination before it gathered the numbers for before B-form optimization, so that it only counted functions that are actually used in programs.

Table 10.1 and Table 10.2 present the percentage of functions that are polymorphic or higher-order before and after B-form optimization. Table 10.1 displays the percentage of polymorphic functions, while Table 10.2 shows the percentage of higher-order but not polymorphic functions.

From these numbers we can draw two conclusions. First, polymorphic or higher-order functions are widely used in SML source programs. On average, 41% of functions are polymorphic or higher-order. Second, optimization is successful at reducing the numbers of polymorphic or higher-order functions. There are no polymorphic functions left after B-form optimization. For six of the eight benchmarks, there are no higher-order functions left. For the other two benchmarks, fewer than 1% of functions are higher-order.

One concern is that inline expansion may be reducing the numbers of polymorphic or

Program	Poly(before) (%)	Poly(after) (%)
checksum	19	0
fft	37	0
leroy	31	0
lexgen	23	0
life	29	0
matmult	46	0
pia	17	0
simple	31	0
arith. mean	29	0

Figure 10.1: Percentage of functions which are polymorphic before and after optimization

Program	HO(before) (%)	HO(after) (%)
checksum	2	0
fft	4	0
leroy	27	2
lexgen	8	0
life	20	0
matmult	0	0
pia	25	0
simple	11	1
arith. mean	12	0

Figure 10.2: Percentage of functions that are higher-order but not polymorphic before and after optimization

program	0	100	225	300	400	500	600	700	800	900	1000
checksum	50	50	58	86	68	68	68	68	68	68	68
fft	60	54	57	57	57	57	57	57	57	57	57
leroy	–	73	89	98	98	108	108	125	125	125	125
lexgen	169	196	174	327	251	222	236	266	266	362	362
life	69	51	59	60	60	60	60	60	60	60	60
matmult	42	40	49	48	48	48	48	48	48	48	48
pia	133	111	152	172	153	182	187	187	177	177	177
simple	125	135	148	161	169	164	190	191	190	191	179

Figure 10.3: Code size (in thousands of bytes) versus size parameter controlling inline expansion

higher-order functions at the expense of greatly increasing program size. To show that this is not the case, I varied the size parameter s controlling inline expansion and measured code size as s varied. Recall that inline expansion will inline a function only if it is non-recursive and suitably small (less then s). I varied s between 0 and 1000 in increments of 100 and measured machine code size as s changed. I include only machine code for the SML program and the standard library; I exclude machine code for the run-time system.

Figure 10.3 shows code size in bytes as the inline expansion parameter s is varied. The point corresponding to $s = 225$ is the default setting for the inline parameter size; it is the point at which I made the measurements described earlier in the thesis. The point corresponding to $s = 0$ is with no inline expansion; TIL still inlines non-recursive functions that are applied only once.

From Figure 10.3 we can draw several conclusions. First, by comparing $s = 0$ and $s = 225$, we see that excessive code expansion is not occurring. Code size increases by at most 25%. For some programs, it actually decreases. This is due to polymorphic functions being inlined, with smaller code being generated for the monomorphic versions of those functions. Second, we see that code size does not increase excessively when we vary s over a wide range — at most there is a factor of two difference between the minimum and maximum code sizes.

These measurements have several implications for the design of TIL. First, they show that choosing to make monomorphic functions fast at the expense of making polymorphic functions slow was a good design choice. Second, they show that a simple closure conversion strategy was a reasonable design choice. We do not need more sophisticated strategies for these benchmarks. Finally, they show that relying the known call graph in the optimizer, instead of doing an analysis to approximate unknown functions, is a reasonable choice.

10.2 Type information and intermediate program size

One concern in using a typed intermediate language instead of an untyped intermediate language is that the types may make intermediate programs much bigger. This would mean that compilers that use typed intermediate languages would use more memory than compilers which use untyped intermediate languages. In addition, they could also be slower, since some optimizations for typed intermediate languages need to traverse the types.

Program	Before	After
checksum	3.98	3.15
fft	3.97	1.81
leroy	7.34	4.35
lexgen	5.00	3.55
life	5.98	4.18
matmult	4.78	2.34
pia	4.89	3.44
simple	6.10	2.09
geo. mean	5.15	2.97

Figure 10.4: Ratios of total program size to term-level size, without careful treatment of types.

Figure 10.4 shows the increase in intermediate program size due to types for TIL, before and after B-form optimization, without any careful management of types. For each benchmark, it presents the ratio of total program size to the size of term-level expressions and declarations. The term-level expressions and declarations correspond to an untyped program, so this ratio tells how much bigger types make programs. I measured size by instrumenting the compiler to count syntax nodes in intermediate programs. The count excludes value nodes and base type nodes. We see that types make programs on average roughly five times bigger before optimization, and three times bigger after optimization.

You can reduce the effect of types on program size by managing types carefully so that types are *shared*. For example, the type of a record expression of the form `record(a,b)` has the types of `a` and `b` as sub-components. If we name those types, say as τ_1 and τ_2, then the type of the record expression only has a size of three, instead of the size(τ_1) + size$(\tau_2) + 3$.

To name types so that they can be shared, I added a `let` form which binds a type variable to a type expression to the B-form intermediate language. I then modified dead-code elimination to eliminate useless type variables. I also modified invariant and hoist to hoist these `let`-bound types when possible. Next, I modified CSE to eliminate type expressions, in addition to redundant constructor expressions and term-level expressions. CSE only replaces type expressions with type variables, even though it could replace type expressions with constructor variables. Finally, I modified other optimizations to preserve the `let`-bound types in programs.

Figure 10.5 shows the increase in intermediate program size due to types with this management scheme. The column labelled "before" presents the ratio of sizes after one round of CSE but before any other optimization. The column labelled "after" presents the ratio of sizes after B-form optimization. We see that with careful management of types, programs are on average two times bigger before optimization and only 15% bigger after optimization.

One question is whether doing a round of other optimizations, such as hoisting, in addition to CSE, might dramatically decrease the effect of types on program size at the beginning of optimization. The answer is no; the effect of types on program size gradually drops as

Program	Before	After
checksum	2.07	1.30
fft	1.83	1.03
leroy	2.38	1.27
lexgen	1.62	1.13
life	1.96	1.14
matmult	2.51	1.19
pia	1.64	1.11
simple	1.64	1.03
geo. mean	1.93	1.15

Figure 10.5: Ratios of total program size to term-level size, with careful treatment of types

the optimizer runs. I speculate that this is due to the optimizer eliminating polymorphic functions and the type information needed for those functions.

Chapter 11

Future work

In this chapter, I suggest some future directions that researchers may wish to explore. For convenience, I have divided this chapter into two sections. In the first section, I suggest some comparative studies of language implementation techniques that may be worthwhile for languages like SML. In the second section, I discuss some future directions for optimizing SML programs.

11.1 Comparative studies of language implementation techniques

In this thesis, I have advocated an empirical approach to implementing modern programming languages. First, I studied the memory-system performance of SML programs and the cost of automatic storage management. Based on these studies, I concluded that reducing the instruction counts of SML programs was important. I then studied the effect of loop optimizations on SML program performance.

It is possible to carry this empirical approach much further, by systematically studying and comparing different programming language implementation techniques. The following studies may be worthwhile for languages like SML:

- comparing different data representation techniques for polymorphic garbage-collected languages. There are at least three different techniques that should be compared: using a conventional universal representation, where data is represented as a tagged machine word, Leroy's approach [56], where polymorphic values are placed in a universal representation but monomorphic values are kept in a native machine representation, and intensional polymorphism [37].

- comparing different inlining techniques. It would be interesting to compare SML/NJ's inlining strategy to TIL's inlining strategy. For example, TIL does not inline recursive functions, while SML/NJ does.

- comparing different techniques for automatic storage management. It would be interesting to compare fully tag-free automatic storage management [90] to the mostly tag-free approach used in TIL.

11.2 Optimization

Several directions worth pursuing in the area of optimization are proving the correctness of the optimizations, studying how to improve the current optimizations, and implementing a broader range of loop optimizations.

11.2.1 Correctness of programs produced by the optimizer

In Chapter 6, I conclude that the call-by-value λ-calculus with simple extensions justifies almost all the transformations that I use in the optimization algorithms. Thus, you should be able to prove the correctness of the optimizer using a conceptually straightforward (but tedious) approach. You would first have to re-establish all this theory for LMLI. After you developed suitable machinery, you would need to formalize the algorithms and prove that they transform programs in accordance with the calculus. You could do this by borrowing techniques from systems for transforming programs, such as NuPRL. You could encapsulate programs as an abstract data type that has operations that transform programs according to a calculus. You could modify each optimization to produce a list of transformations to apply (i.e. a proof of correctness). The only part of the optimizer that you would need to prove correct would be the abstract operations on programs.

11.2.2 Improving current optimizations

Better effects analyses would allow the current code motion algorithms to move more expressions. The interesting question is how much of an improvement can be achieved at what cost in compile time. One direction to pursue is analyzing the effects of function applications, so that they could be eliminated by common-subexpression elimination. CSE can be done for applications of functions even when the functions may not terminate or may raise exceptions; the only requirement is that functions not modify or use the store.

Barth's work [13] and related work on computing interprocedural summary information could be extended to compute effects that occur when a function is applied. Interprocedural summary information tells what variables are used or modified by functions, or what variables are aliased to other variables. The analysis could use the known fragment of the call graph and assume that any effect can occur when an unknown function is called. To determine whether a function is effect-free, it would simply scan its body for "impure" expressions. Of course, the analysis would have to be transitive, and scan any functions it calls and so on. The analysis could be done in $O(N)$ time, where N is the size of the program if the lattice of Section 5.3.1 is used, functions are traversed in depth-first search using the call graph, and mutually-recursive functions are coalesced as single nodes in the call graph,

A cautionary note is that Richardson and Ganapathi [75] found that for Pascal programs interprocedural summary information did not improve optimization. However, their optimizer did not move function applications.

11.2.3 Additional optimizations

It would be interesting to implement a much broader range of optimizations that improve loops in the context of SML. Optimizations particularly worth examining are strength reduction and induction variable elimination, loop-based techniques for reducing the cost of array bounds checking, and unrolling recursive functions.

It would also be interesting to implement optimizations that improve imperative language features. Consider, for example, eliminating redundant fetches of array elements or eliminating fetches of array elements that are invariant in recursive functions.

I regard optimizations that improve imperative language features as distinct from the loop optimizations that I described in Chapter 6. The loop optimizations do not do any path analysis of side-effects, which optimizations that improve imperative language features would need to do. For example, if an optimization was going to move a side-effecting expression e from point A to point B in a program, it would need to examine side effects along all execution paths between A and B to prove that the part of the state that e uses is identical at points A and B.

11.2.4 Effect of separate compilation on optimization

It would be interesting to study the effect of separate compilation on optimization (and, in general, on the performance of programs compiled by TIL). There are several direction worth pursuing. First, it would be worthwhile to study optimization across compilation units. I conjecture that inlining across compilation units will be important. I also conjecture that propagating effects information across compilation units will be important. Second, it would be worthwhile to study optimizations that reduce the cost of intensional polymorphism when types are unknown. For example, the tag construction that occurs in Example 7.8 in Chapter 7 is invariant in the recursive function: the tag could be constructed outside the recursive function and passed as an additional argument.

Chapter 12

Conclusion

This chapter is divided into three sections. In the first section, I summarize the specific contributions of the thesis, most of which are concerned with compilation of programming languages like SML. In the second section, I take a step back from the details of compiling SML to give some advice to compiler writers concerning implementing new programming languages. Finally, I take a step back from compilation to discuss the broader implications of this thesis — and work like it — for the field of computing.

12.1 Summary of contributions

The primary contributions that I have made in this thesis are:

- demonstrating that a "pay-as-you-go" compilation strategy, where programmers pay for advanced language features only when they use them, is a practical strategy for compiling SML programs.
- showing that compilers for languages such as SML should focus on generating good code for recursive functions, just as compilers for conventional languages such as C focus on generating good code for loops.

Additional contributions that I have made while completing this thesis are:

- demonstrating an alternative approach to compiling languages with higher-order functions. The conventional approach to compiling languages with higher-order functions is to focus on "compiling functions well" by choosing efficient environment representations for first-class functions. In contrast, my approach is to use optimization to eliminate higher-order functions and to adapt conventional compiler technology to compile optimized programs to efficient machine code.
- presenting algorithms that show how to apply several optimizations that are known to improve loops to SML programs. I have given algorithms for common-subexpression elimination, invariant removal, eliminating redundant switches, and eliminating redundant comparisons.

- demonstrating that higher-order functions do not have to interfere with optimization. In practice, inlining and uncurrying are effective at reducing the number of higher-order functions.

- showing how to use the call graph of a program to estimate how frequently various program points are executed.

- showing that heap allocation and copying garbage collection can have good memory-system performance, given the right memory system.

12.2 Lessons for compiler writers

The previous section listed a number of contributions, most of which are specific to the problem of compiling languages like SML. In the process of completing this thesis, I also learned a number of lessons about implementing new programming languages. To a compiler writer who wishes to implement a new programming language, I offer the following suggestions, distilled from my experience completing the work described in this thesis:

- Focus on making the core of your new language fast. It is a mistake to focus on the advanced features of the language at the expense of the more mundane aspects of the language. The advanced features may be more interesting to work on, but programs are likely to spend most of their time doing mundane operations such as arithmetic.

- Measure your compiled code carefully. It is difficult to understand where programs spend their time on modern architectures. and intuition, as this thesis has demonstrated, is an unreliable guide. It is also difficult to predict *a priori* what particular features of the language to spend time optimizing.

  Of course, this is easier to say than to do for new programming languages. It can be difficult to find "representative" programs.

- Try simple *local* optimizations first. I define a local optimization as an optimization that can work on a fragment of a program. In particular, a local optimization does not require an analysis of an entire program for correctness. Almost all the optimizations described in this thesis fit this description.

  There are two reasons to prefer local optimizations. First, as a general rule-of-thumb, simple optimizations will give you 80% of the benefit of optimization. Obtaining the last 20% is much harder. Second, local optimizations are less fragile than global optimizations when programs are changed. No one wants to use a programming language where a seemingly minor program change disables optimization and causes a program to become a factor of two slower.

- Build analyses to suit transformations and not *vice versa*. It is a mistake to build an analysis and then try to find a program transformation that can use of the results of the analysis. It is far better to start with a transformation and then develop an analysis suitable for the transformation.

12.3 The big picture

So, what does this work mean for people involved with computers who are not compiler writers? It means a great deal I believe. Programming languages are the fundamental tools of programmers: they have a pervasive effect on how programs are written, the reliability of programs, and the scale of programs that can be written. A poorly-designed programming language, such as C++, forces a programmer into a morass of details and obscure language rules. It leads to large programs that are nearly incomprehensible and that never quite work right. A well-designed programming language, such as SML, on the other hand makes it a joy to program.

For too long, most of the world has favored poorly designed languages in the name of efficiency. This work presents evidence that we can have well-designed high-level languages such as SML *and* efficiency. More work does remain to be done, particularly in the areas of memory usage of garbage collection, interoperability, and extending languages like SML to be able to handle low-level systems tasks. I believe that with sufficient implementation work, programs written in languages such as SML can match programs written in languages such as C and C++ in performance. Indeed, perhaps one day programs written in high-level languages such as SML will regularly exceed the performance of programs written in more low-level languages.

Appendix A

Memory-System Performance Summary Tables

Table A.1 and Table A.2 give the CPI for each of the benchmark programs measured in Chapter 2 using version 0.91 of the SML/NJ compiler. The following abbreviations are used in the CPI table:

wa/wn: write allocate/write no allocate
s/ns: subblock placement/no subblock placement
4/8: Block size = 4 words/Block size = 8 words

	Associativity = 1							Associativity = 2						
Config	8K	16K	32K	64K	128K	256K	512K	8K	16K	32K	64K	128K	256K	512K
CW														
wn,ns,4	2.41	2.07	1.88	1.73	1.43	1.30	1.18	2.22	1.96	1.78	1.62	1.42	1.30	1.17
wa,ns,4	2.44	2.09	1.90	1.74	1.44	1.31	1.18	2.24	1.98	1.79	1.63	1.43	1.31	1.17
wa,s,4	1.96	1.62	1.44	1.39	1.24	1.20	1.17	1.77	1.50	1.33	1.25	1.21	1.18	1.16
wn,ns,8	2.18	1.89	1.72	1.60	1.37	1.27	1.18	1.98	1.76	1.62	1.50	1.35	1.26	1.17
wa,ns,8	2.19	1.89	1.72	1.60	1.37	1.27	1.18	1.98	1.76	1.61	1.49	1.35	1.26	1.17
wa,s,8	1.88	1.59	1.43	1.38	1.23	1.20	1.17	1.68	1.46	1.32	1.25	1.21	1.18	1.16
Leroy														
wn,ns,4	2.32	2.18	1.96	1.87	1.79	1.65	1.37	2.17	2.03	1.89	1.82	1.76	1.65	1.40
wa,ns,4	2.53	2.40	2.18	2.08	1.98	1.83	1.50	2.38	2.25	2.09	2.00	1.95	1.83	1.57
wa,s,4	1.66	1.52	1.30	1.20	1.15	1.12	1.09	1.51	1.37	1.21	1.12	1.10	1.09	1.08
wn,ns,8	2.05	1.93	1.75	1.67	1.60	1.50	1.30	1.92	1.81	1.68	1.61	1.57	1.50	1.33
wa,ns,8	2.12	2.00	1.82	1.73	1.66	1.56	1.35	1.99	1.87	1.74	1.67	1.63	1.55	1.38
wa,s,8	1.56	1.44	1.26	1.18	1.13	1.11	1.09	1.43	1.32	1.19	1.11	1.09	1.08	1.07
Lexgen														
wn,ns,4	3.59	1.94	1.84	1.72	1.65	1.50	1.31	2.04	1.85	1.70	1.64	1.60	1.50	1.34
wa,ns,4	3.61	1.96	1.85	1.74	1.67	1.51	1.31	2.06	1.87	1.71	1.66	1.61	1.51	1.35
wa,s,4	3.17	1.53	1.42	1.31	1.27	1.21	1.19	1.62	1.43	1.28	1.22	1.20	1.19	1.18
wn,ns,8	2.96	1.78	1.67	1.57	1.51	1.39	1.27	1.86	1.71	1.56	1.49	1.45	1.38	1.28
wa,ns,8	2.97	1.79	1.68	1.57	1.51	1.40	1.27	1.87	1.71	1.57	1.50	1.46	1.39	1.28
wa,s,8	2.70	1.51	1.41	1.30	1.26	1.21	1.19	1.60	1.44	1.30	1.22	1.20	1.18	1.18
Life														
wn,ns,4	1.79	1.70	1.65	1.62	1.61	1.42	1.20	1.77	1.64	1.61	1.60	1.60	1.48	1.19
wa,ns,4	1.80	1.70	1.65	1.62	1.60	1.42	1.20	1.74	1.64	1.60	1.60	1.59	1.48	1.19
wa,s,4	1.39	1.29	1.24	1.21	1.20	1.19	1.19	1.33	1.23	1.20	1.19	1.19	1.19	1.18
wn,ns,8	1.65	1.55	1.49	1.47	1.46	1.34	1.19	1.57	1.54	1.46	1.45	1.44	1.37	1.19
wa,ns,8	1.65	1.55	1.50	1.47	1.45	1.34	1.19	1.57	1.51	1.45	1.45	1.44	1.37	1.19
wa,s,8	1.39	1.29	1.24	1.21	1.20	1.19	1.19	1.31	1.25	1.20	1.19	1.19	1.18	1.18
Pia														
wn,ns,4	2.23	1.93	1.80	1.75	1.62	1.34	1.12	2.23	1.83	1.73	1.72	1.63	1.36	1.12
wa,ns,4	2.27	1.96	1.82	1.77	1.65	1.36	1.12	2.26	1.85	1.76	1.74	1.66	1.39	1.12
wa,s,4	1.66	1.36	1.22	1.18	1.15	1.13	1.10	1.66	1.25	1.15	1.14	1.12	1.11	1.10
wn,ns,8	1.92	1.70	1.59	1.55	1.46	1.26	1.11	1.87	1.60	1.53	1.51	1.45	1.28	1.11
wa,ns,8	1.95	1.72	1.60	1.56	1.47	1.27	1.11	1.89	1.61	1.54	1.52	1.47	1.30	1.11
wa,s,8	1.55	1.32	1.21	1.17	1.14	1.12	1.10	1.49	1.22	1.14	1.13	1.11	1.10	1.10

Table A.1: Cycles per useful instructions, part 1.

Config	Associativity = 1 8K	16K	32K	64K	128K	256K	512K	Associativity = 2 8K	16K	32K	64K	128K	256K	512K
Simple														
wn,ns,4	2.32	2.02	1.79	1.73	1.70	1.68	1.62	1.98	1.74	1.70	1.70	1.68	1.66	1.63
wa,ns,4	2.35	2.05	1.81	1.75	1.72	1.70	1.64	2.01	1.76	1.72	1.72	1.69	1.68	1.65
wa,s,4	1.80	1.50	1.26	1.21	1.19	1.18	1.15	1.46	1.21	1.18	1.17	1.16	1.15	1.14
wn,ns,8	2.03	1.79	1.57	1.51	1.49	1.47	1.43	1.72	1.52	1.49	1.48	1.47	1.46	1.44
wa,ns,8	2.05	1.80	1.58	1.53	1.50	1.48	1.44	1.74	1.54	1.50	1.49	1.48	1.47	1.44
wa,s,8	1.70	1.45	1.23	1.18	1.16	1.15	1.13	1.39	1.19	1.15	1.15	1.14	1.13	1.13
VLIW														
wn,ns,4	3.26	2.73	2.30	1.98	1.79	1.35	1.15	3.06	2.64	2.19	1.88	1.69	1.37	1.14
wa,ns,4	3.29	2.75	2.32	2.00	1.82	1.36	1.15	3.08	2.66	2.21	1.91	1.72	1.40	1.14
wa,s,4	2.67	2.13	1.70	1.39	1.31	1.17	1.13	2.47	2.04	1.59	1.29	1.16	1.14	1.12
wn,ns,8	2.62	2.25	1.93	1.70	1.57	1.28	1.14	2.48	2.16	1.85	1.63	1.50	1.30	1.13
wa,ns,8	2.63	2.26	1.94	1.70	1.58	1.29	1.14	2.48	2.16	1.85	1.64	1.51	1.31	1.13
wa,s,8	2.23	1.86	1.55	1.31	1.25	1.16	1.12	2.09	1.76	1.46	1.25	1.16	1.13	1.12
Yacc														
wn,ns,4	2.38	2.16	1.99	1.90	1.83	1.64	1.33	2.13	1.99	1.89	1.84	1.79	1.65	1.33
wa,ns,4	2.42	2.20	2.02	1.92	1.86	1.67	1.34	2.16	2.02	1.92	1.86	1.82	1.68	1.35
wa,s,4	1.85	1.63	1.45	1.35	1.32	1.27	1.21	1.59	1.45	1.35	1.29	1.26	1.24	1.20
wn,ns,8	2.11	1.90	1.75	1.67	1.62	1.49	1.28	1.87	1.75	1.67	1.62	1.58	1.49	1.28
wa,ns,8	2.13	1.92	1.77	1.68	1.63	1.50	1.29	1.89	1.76	1.68	1.63	1.59	1.50	1.29
wa,s,8	1.76	1.55	1.40	1.32	1.29	1.25	1.20	1.52	1.40	1.32	1.27	1.24	1.22	1.19

Table A.2: Cycles per useful instructions, continued.

Appendix B

Performance numbers

This appendix contains the actual performances numbers used in Chapter 8 and Chapters 9.

B.1 Comparison of TIL against SML/NJ

Program	Exec. time (s)		TIL/NJ
	TIL	NJ	
Checksum	1.94	11.64	0.17
FFT	1.48	16.13	0.09
Knuth-Bendix	1.83	1.74	1.05
Lexgen	1.53	2.53	0.60
Life	1.26	1.43	0.88
Matmult	1.40	7.95	0.18
PIA	0.36	1.10	0.33
SIMPLE	8.05	23.89	0.34
Geo. mean			0.34

Table B.1: Comparison of running times

Program	Heap alloc. (Kbytes)		TIL/NJ
	TIL	NJ	
Checksum	140,517	961,586	0.15
FFT	8,895	209,818	0.042
Knuth-Bendix	45,729	94,493	0.48
Lexgen	8,791	110,749	0.079
Life	24,850	44,198	0.56
Matmult	0	249,942	—-
PIA	5,322	53,850	0.10
SIMPLE	315,042	807,133	0.39
Geo. mean (excluding Matmult)			0.18

Table B.2: Comparison of heap allocation

Program	Phys. mem. (Kbytes)		TIL/NJ
	TIL	NJ	
Checksum	848	1456	0.47
FFT	2696	17592	0.15
Knuth-Bendix	2648	3496	0.74
Lexgen	1624	2952	0.55
Life	848	1264	0.65
Matmult	512	1560	0.33
PIA	1144	1648	0.68
SIMPLE	10008	17200	0.54
Geo. mean			0.45

Table B.3: Comparison of maximum physical memory used

Program	Exec. size (Kbytes)		TIL/NJ
	TIL	NJ	
Checksum	288	638	0.43
FFT	304	649	0.46
Knuth-Bendix	328	671	0.48
Lexgen	440	731	0.61
Life	280	651	0.43
Matmult	576	767	0.75
PIA	456	724	0.63
SIMPLE	928	876	0.47
Geo. mean			0.52

Table B.4: Comparison of stand-alone executable sizes

Program	Comp. time (s)		TIL/NJ
	TIL	NJ	
Checksum	9.74	1.49	6.5
FFT	12.2	1.97	6.2
Knuth-Bendix	77.1	7.79	9.9
Lexgen	248.67	13.12	18.9
Life	23.4	2.73	8.6
Matmult	4.12	1.02	4.04
PIA	219.61	15.83	13.9
SIMPLE	227.61	18.31	12.4
Geo. mean			9.1

Table B.5: Comparison of compilation times

B.2 Effects of loop optimizations

This section contains the measurements of the effects of the loop optimizations (comparison elimination, CSE, hoisting, invariant removal, and redundant switch elimination). For each benchmark, I measured performance with all optimizations enabled (allopts), all optimizations but the loop optimizations enabled (noloopopts), and then with each loop optimization enabled one at a time (cmpelim, CSE, invariant, hoist, and switch). For the measurements of each loop optimization enabled by itself, all optimizations but the other loop optimizations were still enabled.

The measurements are the averages of 100 runs of the program; I report standard deviations for all the measurements.

I also calculated ratios of performance. The ratios are relative to noloopopts.

```
Checksum

allopts:

TIME(s): total time      =   2.10   std. dev. 0.34
          user time      =   2.03   std. dev. 0.34
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.05   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.04   std. dev. 0.01
          gc-stack time  =   0.01   std. dev. 0.00
GC(k):   bytes alloced   = 143889360   std. dev.   0
         bytes copied    =       544   std. dev.   0
MEM(K):  max phys mem    =       689   std. dev.   2
PAGING:  page reclaims   =       103   std. dev.   2
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     10404   std. dev.   0
         CodeSize        =     47988   std. dev.   0

noloopopts:

TIME(s): total time      =  15.89   std. dev. 0.07
          user time      =   5.54   std. dev. 0.02
          sys time       =   0.05   std. dev. 0.01
          gc-all time    =  10.30   std. dev. 0.05
          majorgc-copy   =   0.27   std. dev. 0.01
          gc-copy time   =  10.25   std. dev. 0.04
          gc-stack time  =   0.05   std. dev. 0.01
GC(k):   bytes alloced   = 390598672   std. dev.   0
         bytes copied    = 123364696   std. dev.   0
MEM(K):  max phys mem    =     12280   std. dev.   0
PAGING:  page reclaims   =      1551   std. dev.   2
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     11068   std. dev.   0
         CodeSize        =     49584   std. dev.   0

cmpelim:

TIME(s): total time      =  15.92   std. dev. 0.04
          user time      =   5.58   std. dev. 0.02
          sys time       =   0.03   std. dev. 0.01
          gc-all time    =  10.31   std. dev. 0.03
          majorgc-copy   =   0.27   std. dev. 0.00
          gc-copy time   =  10.27   std. dev. 0.03
          gc-stack time  =   0.03   std. dev. 0.00
GC(k):   bytes alloced   = 390598672   std. dev.   0
         bytes copied    = 123364696   std. dev.   0
MEM(K):  max phys mem    =     12280   std. dev.   0
PAGING:  page reclaims   =      1551   std. dev.   2
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     11068   std. dev.   0
         CodeSize        =     49776   std. dev.   0

cse:

TIME(s): total time      =   2.19   std. dev. 0.01
          user time      =   2.12   std. dev. 0.01
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.05   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.04   std. dev. 0.01
          gc-stack time  =   0.01   std. dev. 0.00
GC(k):   bytes alloced   = 144009328   std. dev.   0
         bytes copied    =     60464   std. dev.   0
MEM(K):  max phys mem    =       745   std. dev.   2
PAGING:  page reclaims   =       110   std. dev.   2
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     10628   std. dev.   0
         CodeSize        =     48988   std. dev.   0

hoist:

TIME(s): total time      =   1.96   std. dev. 0.01
          user time      =   1.89   std. dev. 0.01
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.05   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.04   std. dev. 0.01
          gc-stack time  =   0.01   std. dev. 0.00
GC(k):   bytes alloced   = 143889804   std. dev.   0
         bytes copied    =       988   std. dev.   0
MEM(K):  max phys mem    =       697   std. dev.   2
PAGING:  page reclaims   =       104   std. dev.   2
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     10796   std. dev.   0
         CodeSize        =     49344   std. dev.   0

invariant:

TIME(s): total time      =   1.99   std. dev. 0.01
          user time      =   1.92   std. dev. 0.01
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.05   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.04   std. dev. 0.00
          gc-stack time  =   0.01   std. dev. 0.00
GC(k):   bytes alloced   = 143889784   std. dev.   0
         bytes copied    =       968   std. dev.   0
MEM(K):  max phys mem    =       696   std. dev.   2
PAGING:  page reclaims   =       104   std. dev.   2
          page faults    =         0   std. dev.   0
```

```
MISC:    GCTableSize      =    10800   std. dev.   0
         CodeSize         =    49292   std. dev.   0

switch:

TIME(s): total time       =  15.76   std. dev. 0.04
          user time       =   5.52   std. dev. 0.02
          sys time        =   0.03   std. dev. 0.01
          gc-all time     =  10.21   std. dev. 0.02
          majorgc-copy    =   0.26   std. dev. 0.00
          gc-copy time    =  10.18   std. dev. 0.02
          gc-stack time   =   0.03   std. dev. 0.01
GC(k):   bytes alloced    = 390598672   std. dev.   0
         bytes copied     = 123364696   std. dev.   0
MEM(K):  max phys mem     =    12272   std. dev.   0
PAGING:  page reclaims    =     1550   std. dev.   2
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    11068   std. dev.   0
         CodeSize         =    49312   std. dev.   0

Ratios:

std opts + cmpelim / std opts :

TIME(s): total time       =  1.002
          user time       =  1.006
          sys time        =  0.731
          gc-all time     =  1.001
          majorgc-copy    =  1.024
          gc-copy time    =  1.002
          gc-stack time   =  0.709
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    1.004

std opts + cse / std opts :

TIME(s): total time       =  0.137
          user time       =  0.382
          sys time        =  0.415
          gc-all time     =  0.005
          majorgc-copy    =  0.000
          gc-copy time    =  0.004
          gc-stack time   =  0.217
GC(k):   bytes alloced    =    0.369
         bytes copied     =    0.000
MEM(K):  max phys mem     =    0.061
PAGING:  page reclaims    =    0.071
          page faults     =   n.d.
MISC:    GCTableSize      =    0.960
         CodeSize         =    0.988

std opts + hoist / std opts :

TIME(s): total time       =  0.123
          user time       =  0.342
          sys time        =  0.419
          gc-all time     =  0.005
          majorgc-copy    =  0.000
          gc-copy time    =  0.004
          gc-stack time   =  0.216
GC(k):   bytes alloced    =    0.368
         bytes copied     =    0.000
MEM(K):  max phys mem     =    0.057
PAGING:  page reclaims    =    0.067
          page faults     =   n.d.
MISC:    GCTableSize      =    0.975
         CodeSize         =    0.995

std opts + invariant / std opts :

TIME(s): total time       =  0.125
          user time       =  0.346
          sys time        =  0.400
          gc-all time     =  0.005
          majorgc-copy    =  0.000
          gc-copy time    =  0.004
          gc-stack time   =  0.227
GC(k):   bytes alloced    =    0.368
         bytes copied     =    0.000
MEM(K):  max phys mem     =    0.057
PAGING:  page reclaims    =    0.067
          page faults     =   n.d.
MISC:    GCTableSize      =    0.976
         CodeSize         =    0.994

std opts + switch / std opts :

TIME(s): total time       =  0.992
          user time       =  0.995
          sys time        =  0.718
          gc-all time     =  0.991
          majorgc-copy    =  0.971
          gc-copy time    =  0.993
          gc-stack time   =  0.656
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    0.999
PAGING:  page reclaims    =    0.999
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    0.995

std opts + allopts / std opts :

TIME(s): total time       =  0.132
          user time       =  0.365
          sys time        =  0.407
          gc-all time     =  0.005
          majorgc-copy    =  0.000
          gc-copy time    =  0.004
          gc-stack time   =  0.288
GC(k):   bytes alloced    =    0.368
         bytes copied     =    0.000
MEM(K):  max phys mem     =    0.056
PAGING:  page reclaims    =    0.067
         page faults     =   n.d.
MISC:    GCTableSize      =    0.940
         CodeSize         =    0.968

FFT:

allopts:

TIME(s): total time       =   1.46   std. dev. 0.05
          user time       =   1.42   std. dev. 0.05
          sys time        =   0.05   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
```

```
          gc-stack time    =   0.00   std. dev. 0.00
GC(k):   bytes alloced    = 9108496   std. dev.   0
         bytes copied     =     680   std. dev.   0
MEM(K):  max phys mem     =    2696   std. dev.   0
PAGING:  page reclaims    =     352   std. dev.   3
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   10620   std. dev.   0
         CodeSize         =   50468   std. dev.   0

noloopopts:

TIME(s): total time       =   9.75   std. dev. 0.06
          user time       =   9.50   std. dev. 0.05
          sys time        =   0.06   std. dev. 0.01
          gc-all time     =   0.18   std. dev. 0.03
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.13   std. dev. 0.02
          gc-stack time   =   0.05   std. dev. 0.01
GC(k):   bytes alloced    = 403882364   std. dev.   0
         bytes copied     =   85432   std. dev.   0
MEM(K):  max phys mem     =    2952   std. dev.   0
PAGING:  page reclaims    =     384   std. dev.   3
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   20116   std. dev.   0
         CodeSize         =   67972   std. dev.   0

cmpelim:

TIME(s): total time       =   9.97   std. dev. 0.04
          user time       =   9.69   std. dev. 0.05
          sys time        =   0.06   std. dev. 0.01
          gc-all time     =   0.22   std. dev. 0.02
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.16   std. dev. 0.01
          gc-stack time   =   0.07   std. dev. 0.01
GC(k):   bytes alloced    = 403882364   std. dev.   0
         bytes copied     =   85432   std. dev.   0
MEM(K):  max phys mem     =    2952   std. dev.   0
PAGING:  page reclaims    =     384   std. dev.   3
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   20116   std. dev.   0
         CodeSize         =   67996   std. dev.   0

cse:

TIME(s): total time       =   6.15   std. dev. 0.10
          user time       =   5.99   std. dev. 0.10
          sys time        =   0.06   std. dev. 0.01
          gc-all time     =   0.10   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.07   std. dev. 0.01
          gc-stack time   =   0.03   std. dev. 0.01
GC(k):   bytes alloced    = 263001232   std. dev.   0
         bytes copied     =   78632   std. dev.   0
MEM(K):  max phys mem     =    2872   std. dev.   0
PAGING:  page reclaims    =     375   std. dev.   2
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   16396   std. dev.   0
         CodeSize         =   61708   std. dev.   0

hoist:

TIME(s): total time       =   9.35   std. dev. 0.32
          user time       =   9.13   std. dev. 0.32
          sys time        =   0.06   std. dev. 0.01
          gc-all time     =   0.16   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.12   std. dev. 0.01
          gc-stack time   =   0.04   std. dev. 0.01
GC(k):   bytes alloced    = 403882428   std. dev.   0
         bytes copied     =   85472   std. dev.   0
MEM(K):  max phys mem     =    2952   std. dev.   0
PAGING:  page reclaims    =     384   std. dev.   3
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   18236   std. dev.   0
         CodeSize         =   65096   std. dev.   0

invariant:

TIME(s): total time       =   7.06   std. dev. 0.06
          user time       =   6.94   std. dev. 0.06
          sys time        =   0.05   std. dev. 0.00
          gc-all time     =   0.06   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.05   std. dev. 0.01
          gc-stack time   =   0.01   std. dev. 0.00
GC(k):   bytes alloced    = 153045404   std. dev.   0
         bytes copied     =   65480   std. dev.   0
MEM(K):  max phys mem     =    2808   std. dev.   0
PAGING:  page reclaims    =     366   std. dev.   3
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   17464   std. dev.   0
         CodeSize         =   66804   std. dev.   0

switch:

TIME(s): total time       =   9.78   std. dev. 0.03
          user time       =   9.56   std. dev. 0.02
          sys time        =   0.06   std. dev. 0.01
          gc-all time     =   0.17   std. dev. 0.02
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.12   std. dev. 0.01
          gc-stack time   =   0.05   std. dev. 0.01
GC(k):   bytes alloced    = 403882364   std. dev.   0
         bytes copied     =   85432   std. dev.   0
MEM(K):  max phys mem     =    2952   std. dev.   0
PAGING:  page reclaims    =     384   std. dev.   3
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =   20116   std. dev.   0
         CodeSize         =   67644   std. dev.   0

Ratios:

std opts + cmpelim / std opts :

TIME(s): total time       =  1.023
          user time       =  1.020
          sys time        =  0.913
          gc-all time     =  1.194
          majorgc-copy    = n.d.
          gc-copy time    =  1.178
          gc-stack time   =  1.229
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    1.000

std opts + cse / std opts :

TIME(s): total time       =  0.630
          user time       =  0.630
```

```
          sys time        =  0.953
          gc-all time     =  0.530
          majorgc-copy    = n.d.
          gc-copy time    =  0.547
          gc-stack time   =  0.478
GC(k):   bytes alloced    =    0.651
         bytes copied     =    0.920
MEM(K):  max phys mem     =    0.973
PAGING:  page reclaims    =    0.977
          page faults     =   n.d.
MISC:    GCTableSize      =    0.815
         CodeSize         =    0.908
```

std opts + hoist / std opts :

```
TIME(s): total time       =  0.959
          user time       =  0.961
          sys time        =  1.037
          gc-all time     =  0.846
          majorgc-copy    = n.d.
          gc-copy time    =  0.885
          gc-stack time   =  0.749
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    0.907
         CodeSize         =    0.958
```

std opts + invariant / std opts :

```
TIME(s): total time       =  0.724
          user time       =  0.730
          sys time        =  0.896
          gc-all time     =  0.350
          majorgc-copy    = n.d.
          gc-copy time    =  0.386
          gc-stack time   =  0.271
GC(k):   bytes alloced    =    0.379
         bytes copied     =    0.766
MEM(K):  max phys mem     =    0.951
PAGING:  page reclaims    =    0.953
          page faults     =   n.d.
MISC:    GCTableSize      =    0.868
         CodeSize         =    0.983
```

std opts + switch / std opts :

```
TIME(s): total time       =  1.004
          user time       =  1.006
          sys time        =  0.928
          gc-all time     =  0.903
          majorgc-copy    = n.d.
          gc-copy time    =  0.913
          gc-stack time   =  0.880
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    0.995
```

std opts + allopts / std opts :

```
TIME(s): total time       =  0.150
          user time       =  0.149
          sys time        =  0.744
          gc-all time     =  0.020
          majorgc-copy    = n.d.
          gc-copy time    =  0.022
          gc-stack time   =  0.015
GC(k):   bytes alloced    =    0.023
         bytes copied     =    0.008
MEM(K):  max phys mem     =    0.913
PAGING:  page reclaims    =    0.917
          page faults     =   n.d.
MISC:    GCTableSize      =    0.528
         CodeSize         =    0.742
```

Knuth-Bendix

allopts:

```
TIME(s): total time       =   1.94   std. dev. 0.04
          user time       =   1.50   std. dev. 0.04
          sys time        =   0.02   std. dev. 0.00
          gc-all time     =   0.42   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.18   std. dev. 0.01
          gc-stack time   =   0.24   std. dev. 0.00
GC(k):   bytes alloced    = 46827344   std. dev.   0
         bytes copied     =  1776380   std. dev.   0
MEM(K):  max phys mem     =     2696   std. dev.   0
PAGING:  page reclaims    =      353   std. dev.   2
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    22000   std. dev.   0
         CodeSize         =    81992   std. dev.   0
```

noloopopts:

```
TIME(s): total time       =   2.85   std. dev. 0.09
          user time       =   2.25   std. dev. 0.09
          sys time        =   0.03   std. dev. 0.00
          gc-all time     =   0.57   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.27   std. dev. 0.01
          gc-stack time   =   0.30   std. dev. 0.01
GC(k):   bytes alloced    = 59744436   std. dev.   0
         bytes copied     =  2061016   std. dev.   0
MEM(K):  max phys mem     =     3048   std. dev.   0
PAGING:  page reclaims    =      398   std. dev.   2
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    22748   std. dev.   0
         CodeSize         =    88812   std. dev.   0
```

cmpelim:

```
TIME(s): total time       =   2.83   std. dev. 0.01
          user time       =   2.25   std. dev. 0.01
          sys time        =   0.02   std. dev. 0.00
          gc-all time     =   0.56   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.26   std. dev. 0.00
          gc-stack time   =   0.30   std. dev. 0.01
GC(k):   bytes alloced    = 59744436   std. dev.   0
         bytes copied     =  2061016   std. dev.   0
MEM(K):  max phys mem     =     3048   std. dev.   0
PAGING:  page reclaims    =      398   std. dev.   2
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    22748   std. dev.   0
         CodeSize         =    88840   std. dev.   0
```

```
cse:

TIME(s): total time      =   2.13   std. dev. 0.01
          user time      =   1.62   std. dev. 0.01
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.49   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.22   std. dev. 0.01
          gc-stack time  =   0.27   std. dev. 0.00
GC(k):   bytes alloced   = 54817816   std. dev.   0
         bytes copied    =  2134024   std. dev.   0
MEM(K):  max phys mem    =     3056   std. dev.   0
PAGING:  page reclaims   =      398   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    21400   std. dev.   0
         CodeSize        =    85064   std. dev.   0

hoist:

TIME(s): total time      =   1.89   std. dev. 0.02
          user time      =   1.48   std. dev. 0.01
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.40   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.18   std. dev. 0.01
          gc-stack time  =   0.22   std. dev. 0.00
GC(k):   bytes alloced   = 46895404   std. dev.   0
         bytes copied    =  1768656   std. dev.   0
MEM(K):  max phys mem    =     2704   std. dev.   0
PAGING:  page reclaims   =      353   std. dev.   3
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    23176   std. dev.   0
         CodeSize        =    89328   std. dev.   0

invariant:

TIME(s): total time      =   2.10   std. dev. 0.02
          user time      =   1.62   std. dev. 0.02
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.46   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.19   std. dev. 0.01
          gc-stack time  =   0.27   std. dev. 0.00
GC(k):   bytes alloced   = 54531944   std. dev.   0
         bytes copied    =  1818524   std. dev.   0
MEM(K):  max phys mem    =     2744   std. dev.   0
PAGING:  page reclaims   =      359   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    21368   std. dev.   0
         CodeSize        =    84852   std. dev.   0

switch:

TIME(s): total time      =   2.93   std. dev. 0.04
          user time      =   2.35   std. dev. 0.02
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.56   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.26   std. dev. 0.01
          gc-stack time  =   0.30   std. dev. 0.01
GC(k):   bytes alloced   = 59744436   std. dev.   0
         bytes copied    =  2061016   std. dev.   0
MEM(K):  max phys mem    =     3048   std. dev.   0
PAGING:  page reclaims   =      398   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    22748   std. dev.   0
         CodeSize        =    87964   std. dev.   0
```

```
Ratios:

std opts + cmpelim / std opts :

TIME(s): total time      =  0.994
          user time      =  1.002
          sys time       =  0.818
          gc-all time    =  0.971
          majorgc-copy   = n.d.
          gc-copy time   =  0.961
          gc-stack time  =  0.981
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
          page faults    =   n.d.
MISC:    GCTableSize     =    1.000
         CodeSize        =    1.000

std opts + cse / std opts :

TIME(s): total time      =  0.747
          user time      =  0.718
          sys time       =  0.877
          gc-all time    =  0.853
          majorgc-copy   = n.d.
          gc-copy time   =  0.802
          gc-stack time  =  0.899
GC(k):   bytes alloced   =    0.918
         bytes copied    =    1.035
MEM(K):  max phys mem    =    1.003
PAGING:  page reclaims   =    1.000
          page faults    =   n.d.
MISC:    GCTableSize     =    0.941
         CodeSize        =    0.958

std opts + hoist / std opts :

TIME(s): total time      =  0.664
          user time      =  0.656
          sys time       =  0.689
          gc-all time    =  0.697
          majorgc-copy   = n.d.
          gc-copy time   =  0.642
          gc-stack time  =  0.747
GC(k):   bytes alloced   =    0.785
         bytes copied    =    0.858
MEM(K):  max phys mem    =    0.887
PAGING:  page reclaims   =    0.887
          page faults    =   n.d.
MISC:    GCTableSize     =    1.019
         CodeSize        =    1.006

std opts + invariant / std opts :

TIME(s): total time      =  0.737
          user time      =  0.720
          sys time       =  0.777
          gc-all time    =  0.799
          majorgc-copy   = n.d.
          gc-copy time   =  0.687
          gc-stack time  =  0.899
GC(k):   bytes alloced   =    0.913
         bytes copied    =    0.882
MEM(K):  max phys mem    =    0.900
PAGING:  page reclaims   =    0.902
```

```
         page faults     =   n.d.
MISC:    GCTableSize     =    0.939
         CodeSize        =    0.955
```

std opts + switch / std opts :

```
TIME(s): total time      =  1.029
         user time       =  1.043
         sys time        =  0.892
         gc-all time     =  0.979
         majorgc-copy    = n.d.
         gc-copy time    =  0.969
         gc-stack time   =  0.989
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
         page faults     =   n.d.
MISC:    GCTableSize     =    1.000
         CodeSize        =    0.990
```

std opts + allopts / std opts :

```
TIME(s): total time      =  0.680
         user time       =  0.666
         sys time        =  0.744
         gc-all time     =  0.731
         majorgc-copy    = n.d.
         gc-copy time    =  0.672
         gc-stack time   =  0.783
GC(k):   bytes alloced   =    0.784
         bytes copied    =    0.862
MEM(K):  max phys mem    =    0.885
PAGING:  page reclaims   =    0.887
         page faults     =   n.d.
MISC:    GCTableSize     =    0.967
         CodeSize        =    0.923
```

Lexgen

allopts:

```
TIME(s): total time      =   1.56   std. dev. 0.02
         user time       =   1.46   std. dev. 0.02
         sys time        =   0.03   std. dev. 0.00
         gc-all time     =   0.07   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.06   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 9002360   std. dev.   0
         bytes copied    =  787656   std. dev.   0
MEM(K):  max phys mem    =    1616   std. dev.   0
PAGING:  page reclaims   =     219   std. dev.   3
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   44896   std. dev.   0
         CodeSize        =  160756   std. dev.   0
```

noloopopts:

```
TIME(s): total time      =   1.73   std. dev. 0.00
         user time       =   1.63   std. dev. 0.00
         sys time        =   0.03   std. dev. 0.00
         gc-all time     =   0.08   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.07   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8688980   std. dev.   0
         bytes copied    =  779448   std. dev.   0
MEM(K):  max phys mem    =    1720   std. dev.   0
PAGING:  page reclaims   =     232   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   57828   std. dev.   0
         CodeSize        =  212300   std. dev.   0
```

cmpelim:

```
TIME(s): total time      =   1.61   std. dev. 0.01
         user time       =   1.50   std. dev. 0.01
         sys time        =   0.03   std. dev. 0.01
         gc-all time     =   0.08   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.07   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8688980   std. dev.   0
         bytes copied    =  779448   std. dev.   0
MEM(K):  max phys mem    =    1728   std. dev.   0
PAGING:  page reclaims   =     233   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   57020   std. dev.   0
         CodeSize        =  207492   std. dev.   0
```

cse:

```
TIME(s): total time      =   1.64   std. dev. 0.01
         user time       =   1.55   std. dev. 0.01
         sys time        =   0.03   std. dev. 0.00
         gc-all time     =   0.07   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.07   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8689028   std. dev.   0
         bytes copied    =  779164   std. dev.   0
MEM(K):  max phys mem    =    1696   std. dev.   0
PAGING:  page reclaims   =     229   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   55064   std. dev.   0
         CodeSize        =  205276   std. dev.   0
```

hoist:

```
TIME(s): total time      =   1.53   std. dev. 0.01
         user time       =   1.44   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.07   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.06   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8689132   std. dev.   0
         bytes copied    =  779100   std. dev.   0
MEM(K):  max phys mem    =    1632   std. dev.   0
PAGING:  page reclaims   =     221   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   49812   std. dev.   0
         CodeSize        =  190636   std. dev.   0
```

invariant:

```
TIME(s): total time      =   1.57   std. dev. 0.01
         user time       =   1.48   std. dev. 0.01
         sys time        =   0.03   std. dev. 0.00
         gc-all time     =   0.07   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.07   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
```

```
GC(k):   bytes alloced    =  9002364   std. dev.   0
         bytes copied     =   787732   std. dev.   0
MEM(K):  max phys mem     =     1640   std. dev.   0
PAGING:  page reclaims    =      222   std. dev.   4
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    49128   std. dev.   0
         CodeSize         =   179148   std. dev.   0

switch:

TIME(s): total time       =   1.69   std. dev. 0.05
          user time       =   1.58   std. dev. 0.05
          sys time        =   0.03   std. dev. 0.00
          gc-all time     =   0.08   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.07   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced    =  8688980   std. dev.   0
         bytes copied     =   779436   std. dev.   0
MEM(K):  max phys mem     =     1720   std. dev.   0
PAGING:  page reclaims    =      232   std. dev.   4
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    53816   std. dev.   0
         CodeSize         =   192824   std. dev.   0

Ratios:

std opts + cmpelim / std opts :

TIME(s): total time       =  0.929
          user time       =  0.926
          sys time        =  0.910
          gc-all time     =  1.007
          majorgc-copy    = n.d.
          gc-copy time    =  1.004
          gc-stack time   =  1.082
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.005
PAGING:  page reclaims    =    1.004
          page faults     =   n.d.
MISC:    GCTableSize      =    0.986
         CodeSize         =    0.977

std opts + cse / std opts :

TIME(s): total time       =  0.949
          user time       =  0.952
          sys time        =  0.863
          gc-all time     =  0.909
          majorgc-copy    = n.d.
          gc-copy time    =  0.901
          gc-stack time   =  1.100
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    0.986
PAGING:  page reclaims    =    0.987
          page faults     =   n.d.
MISC:    GCTableSize      =    0.952
         CodeSize         =    0.967

std opts + hoist / std opts :

TIME(s): total time       =  0.882
          user time       =  0.884
          sys time        =  0.827
          gc-all time     =  0.866
          majorgc-copy    = n.d.
          gc-copy time    =  0.859
          gc-stack time   =  1.082
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    0.949
PAGING:  page reclaims    =    0.953
          page faults     =   n.d.
MISC:    GCTableSize      =    0.861
         CodeSize         =    0.898

std opts + invariant / std opts :

TIME(s): total time       =  0.908
          user time       =  0.909
          sys time        =  0.839
          gc-all time     =  0.900
          majorgc-copy    = n.d.
          gc-copy time    =  0.897
          gc-stack time   =  0.975
GC(k):   bytes alloced    =    1.036
         bytes copied     =    1.011
MEM(K):  max phys mem     =    0.953
PAGING:  page reclaims    =    0.957
          page faults     =   n.d.
MISC:    GCTableSize      =    0.850
         CodeSize         =    0.844

std opts + switch / std opts :

TIME(s): total time       =  0.973
          user time       =  0.972
          sys time        =  0.952
          gc-all time     =  1.003
          majorgc-copy    = n.d.
          gc-copy time    =  1.004
          gc-stack time   =  0.978
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    0.931
         CodeSize         =    0.908

std opts + allopts / std opts :

TIME(s): total time       =  0.898
          user time       =  0.898
          sys time        =  0.926
          gc-all time     =  0.885
          majorgc-copy    = n.d.
          gc-copy time    =  0.882
          gc-stack time   =  0.953
GC(k):   bytes alloced    =    1.036
         bytes copied     =    1.011
MEM(K):  max phys mem     =    0.940
PAGING:  page reclaims    =    0.944
          page faults     =   n.d.
MISC:    GCTableSize      =    0.776
         CodeSize         =    0.757

Life

allopts:

TIME(s): total time       =   1.31   std. dev. 0.06
```

```
         user time       =   1.26   std. dev. 0.05
         sys time        =   0.03   std. dev. 0.01
         gc-all time     =   0.02   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.02   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 25446708   std. dev.   0
         bytes copied    =   145012   std. dev.   0
MEM(K):  max phys mem    =      833   std. dev.   3
PAGING:  page reclaims   =      120   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    11368   std. dev.   0
         CodeSize        =    53044   std. dev.   0
```

noloopopts:

```
TIME(s): total time      =   1.43   std. dev. 0.01
         user time       =   1.32   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.10   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.09   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 21562840   std. dev.   0
         bytes copied    =   944480   std. dev.   0
MEM(K):  max phys mem    =     1617   std. dev.   3
PAGING:  page reclaims   =      219   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    12316   std. dev.   0
         CodeSize        =    57464   std. dev.   0
```

cmpelim:

```
TIME(s): total time      =   1.44   std. dev. 0.01
         user time       =   1.32   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.10   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.09   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 21562840   std. dev.   0
         bytes copied    =   944480   std. dev.   0
MEM(K):  max phys mem    =     1617   std. dev.   3
PAGING:  page reclaims   =      219   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    12320   std. dev.   0
         CodeSize        =    57356   std. dev.   0
```

cse:

```
TIME(s): total time      =   1.27   std. dev. 0.00
         user time       =   1.24   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.02   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.02   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20723992   std. dev.   0
         bytes copied    =    98780   std. dev.   0
MEM(K):  max phys mem    =      792   std. dev.   1
PAGING:  page reclaims   =      115   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    11904   std. dev.   0
         CodeSize        =    54976   std. dev.   0
```

hoist:

```
TIME(s): total time      =   1.24   std. dev. 0.01
         user time       =   1.21   std. dev. 0.01
         sys time        =   0.01   std. dev. 0.00
         gc-all time     =   0.02   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.01   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20724024   std. dev.   0
         bytes copied    =    98848   std. dev.   0
MEM(K):  max phys mem    =      793   std. dev.   3
PAGING:  page reclaims   =      116   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    11748   std. dev.   0
         CodeSize        =    55368   std. dev.   0
```

invariant:

```
TIME(s): total time      =   1.27   std. dev. 0.04
         user time       =   1.24   std. dev. 0.04
         sys time        =   0.01   std. dev. 0.00
         gc-all time     =   0.02   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.01   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20724004   std. dev.   0
         bytes copied    =    98792   std. dev.   0
MEM(K):  max phys mem    =      793   std. dev.   2
PAGING:  page reclaims   =      116   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    11924   std. dev.   0
         CodeSize        =    55500   std. dev.   0
```

switch:

```
TIME(s): total time      =   1.42   std. dev. 0.00
         user time       =   1.31   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.09   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.09   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 21562840   std. dev.   0
         bytes copied    =   944480   std. dev.   0
MEM(K):  max phys mem    =     1617   std. dev.   3
PAGING:  page reclaims   =      219   std. dev.   2
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    12216   std. dev.   0
         CodeSize        =    56644   std. dev.   0
```

Ratios:

std opts + cmpelim / std opts :

```
TIME(s): total time      =  1.003
         user time       =  1.003
         sys time        =  0.975
         gc-all time     =  1.019
         majorgc-copy    = n.d.
         gc-copy time    =  1.011
         gc-stack time   =  1.353
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
         page faults     =   n.d.
MISC:    GCTableSize     =    1.000
         CodeSize        =    0.998
```

```
std opts + cse / std opts :

TIME(s): total time      =  0.890
          user time      =  0.939
          sys time       =  0.874
          gc-all time    =  0.203
          majorgc-copy   = n.d.
          gc-copy time   =  0.188
          gc-stack time  =  0.936
GC(k):   bytes alloced   =    0.961
         bytes copied    =    0.105
MEM(K):  max phys mem    =    0.490
PAGING:  page reclaims   =    0.526
          page faults    =   n.d.
MISC:    GCTableSize     =    0.967
         CodeSize        =    0.957

std opts + hoist / std opts :

TIME(s): total time      =  0.868
          user time      =  0.919
          sys time       =  0.835
          gc-all time    =  0.164
          majorgc-copy   = n.d.
          gc-copy time   =  0.142
          gc-stack time  =  1.085
GC(k):   bytes alloced   =    0.961
         bytes copied    =    0.105
MEM(K):  max phys mem    =    0.490
PAGING:  page reclaims   =    0.530
          page faults    =   n.d.
MISC:    GCTableSize     =    0.954
         CodeSize        =    0.964

std opts + invariant / std opts :

TIME(s): total time      =  0.885
          user time      =  0.938
          sys time       =  0.806
          gc-all time    =  0.176
          majorgc-copy   = n.d.
          gc-copy time   =  0.158
          gc-stack time  =  0.902
GC(k):   bytes alloced   =    0.961
         bytes copied    =    0.105
MEM(K):  max phys mem    =    0.490
PAGING:  page reclaims   =    0.530
          page faults    =   n.d.
MISC:    GCTableSize     =    0.968
         CodeSize        =    0.966

std opts + switch / std opts :

TIME(s): total time      =  0.995
          user time      =  0.997
          sys time       =  0.903
          gc-all time    =  0.989
          majorgc-copy   = n.d.
          gc-copy time   =  0.985
          gc-stack time  =  1.128
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
          page faults    =   n.d.
MISC:    GCTableSize     =    0.992
         CodeSize        =    0.986

std opts + allopts / std opts :

TIME(s): total time      =  0.918
          user time      =  0.959
          sys time       =  1.460
          gc-all time    =  0.236
          majorgc-copy   = n.d.
          gc-copy time   =  0.206
          gc-stack time  =  1.583
GC(k):   bytes alloced   =    1.180
         bytes copied    =    0.154
MEM(K):  max phys mem    =    0.515
PAGING:  page reclaims   =    0.548
          page faults    =   n.d.
MISC:    GCTableSize     =    0.923
         CodeSize        =    0.923

Matmult

allopts:

TIME(s): total time      =   1.41   std. dev. 0.00
          user time      =   1.40   std. dev. 0.00
          sys time       =   0.01   std. dev. 0.00
          gc-all time    =   0.00   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.00   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   =     376   std. dev.   0
         bytes copied    =       0   std. dev.   0
MEM(K):  max phys mem    =     504   std. dev.   0
PAGING:  page reclaims   =      80   std. dev.   2
          page faults    =       0   std. dev.   0
MISC:    GCTableSize     =    8760   std. dev.   0
         CodeSize        =   41556   std. dev.   0

noloopopts:

TIME(s): total time      =   2.31   std. dev. 0.00
          user time      =   2.30   std. dev. 0.00
          sys time       =   0.01   std. dev. 0.00
          gc-all time    =   0.00   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.00   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   =     344   std. dev.   0
         bytes copied    =       0   std. dev.   0
MEM(K):  max phys mem    =     504   std. dev.   0
PAGING:  page reclaims   =      81   std. dev.   1
          page faults    =       0   std. dev.   0
MISC:    GCTableSize     =    9052   std. dev.   0
         CodeSize        =   43752   std. dev.   0

cmpelim:

TIME(s): total time      =   1.60   std. dev. 0.00
          user time      =   1.58   std. dev. 0.00
          sys time       =   0.01   std. dev. 0.00
          gc-all time    =   0.00   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.00   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   =     344   std. dev.   0
         bytes copied    =       0   std. dev.   0
MEM(K):  max phys mem    =     504   std. dev.   0
```

```
PAGING:  page reclaims    =      81   std. dev.   2
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =    8964   std. dev.   0
         CodeSize         =   43096   std. dev.   0

cse:

TIME(s): total time       =   2.34   std. dev. 0.00
          user time       =   2.33   std. dev. 0.00
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced    =     344   std. dev.   0
         bytes copied     =       0   std. dev.   0
MEM(K):  max phys mem     =     504   std. dev.   0
PAGING:  page reclaims    =      81   std. dev.   2
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =    8924   std. dev.   0
         CodeSize         =   43096   std. dev.   0

hoist:

TIME(s): total time       =   2.34   std. dev. 0.01
          user time       =   2.33   std. dev. 0.01
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced    =     400   std. dev.   0
         bytes copied     =       0   std. dev.   0
MEM(K):  max phys mem     =     504   std. dev.   0
PAGING:  page reclaims    =      81   std. dev.   2
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =    8912   std. dev.   0
         CodeSize         =   43520   std. dev.   0

invariant:

TIME(s): total time       =   1.93   std. dev. 0.01
          user time       =   1.92   std. dev. 0.01
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced    =     380   std. dev.   0
         bytes copied     =       0   std. dev.   0
MEM(K):  max phys mem     =     504   std. dev.   0
PAGING:  page reclaims    =      80   std. dev.   1
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =    8964   std. dev.   0
         CodeSize         =   43824   std. dev.   0

switch:

TIME(s): total time       =   2.25   std. dev. 0.00
          user time       =   2.24   std. dev. 0.00
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced    =     344   std. dev.   0
         bytes copied     =       0   std. dev.   0
```

```
MEM(K):  max phys mem     =     504   std. dev.   0
PAGING:  page reclaims    =      81   std. dev.   2
          page faults     =       0   std. dev.   0
MISC:    GCTableSize      =    9052   std. dev.   0
         CodeSize         =   43428   std. dev.   0

Ratios:

std opts + cmpelim / std opts :

TIME(s): total time       =  0.690
          user time       =  0.689
          sys time        =  0.948
          gc-all time     = n.d.
          majorgc-copy    = n.d.
          gc-copy time    = n.d.
          gc-stack time   = n.d.
GC(k):   bytes alloced    =    1.000
         bytes copied     =   n.d.
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    0.990
         CodeSize         =    0.985

std opts + cse / std opts :

TIME(s): total time       =  1.012
          user time       =  1.013
          sys time        =  0.883
          gc-all time     = n.d.
          majorgc-copy    = n.d.
          gc-copy time    = n.d.
          gc-stack time   = n.d.
GC(k):   bytes alloced    =    1.000
         bytes copied     =   n.d.
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    0.986
         CodeSize         =    0.985

std opts + hoist / std opts :

TIME(s): total time       =  1.014
          user time       =  1.014
          sys time        =  0.982
          gc-all time     = n.d.
          majorgc-copy    = n.d.
          gc-copy time    = n.d.
          gc-stack time   = n.d.
GC(k):   bytes alloced    =    1.163
         bytes copied     =   n.d.
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    0.985
         CodeSize         =    0.995

std opts + invariant / std opts :

TIME(s): total time       =  0.837
          user time       =  0.836
          sys time        =  0.956
          gc-all time     = n.d.
          majorgc-copy    = n.d.
          gc-copy time    = n.d.
```

```
         gc-stack time   = n.d.
GC(k):   bytes alloced   =    1.105
         bytes copied    =   n.d.
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    0.988
         page faults     =   n.d.
MISC:    GCTableSize     =    0.990
         CodeSize        =    1.002

std opts + switch / std opts :

TIME(s): total time      =  0.974
         user time       =  0.975
         sys time        =  0.923
         gc-all time     = n.d.
         majorgc-copy    = n.d.
         gc-copy time    = n.d.
         gc-stack time   = n.d.
GC(k):   bytes alloced   =    1.000
         bytes copied    =   n.d.
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
         page faults     =   n.d.
MISC:    GCTableSize     =    1.000
         CodeSize        =    0.993

std opts + allopts / std opts :

TIME(s): total time      =  0.612
         user time       =  0.610
         sys time        =  0.928
         gc-all time     = n.d.
         majorgc-copy    = n.d.
         gc-copy time    = n.d.
         gc-stack time   = n.d.
GC(k):   bytes alloced   =    1.093
         bytes copied    =   n.d.
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    0.988
         page faults     =   n.d.
MISC:    GCTableSize     =    0.968
         CodeSize        =    0.950

PIA

allopts:

TIME(s): total time      =   0.37   std. dev. 0.01
         user time       =   0.31   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.04   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.04   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 5450576   std. dev.   0
         bytes copied    =  345200   std. dev.   0
MEM(K):  max phys mem    =    1144   std. dev.   0
PAGING:  page reclaims   =     159   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   39212   std. dev.   0
         CodeSize        =  140788   std. dev.   0

noloopopts:

TIME(s): total time      =   0.42   std. dev. 0.01
         user time       =   0.36   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.04   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.04   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 5664272   std. dev.   0
         bytes copied    =  344552   std. dev.   0
MEM(K):  max phys mem    =    1184   std. dev.   0
PAGING:  page reclaims   =     165   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   49584   std. dev.   0
         CodeSize        =  179132   std. dev.   0

cmpelim:

TIME(s): total time      =   0.41   std. dev. 0.00
         user time       =   0.36   std. dev. 0.00
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.04   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.04   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 5664272   std. dev.   0
         bytes copied    =  344648   std. dev.   0
MEM(K):  max phys mem    =    1184   std. dev.   0
PAGING:  page reclaims   =     165   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   49556   std. dev.   0
         CodeSize        =  179032   std. dev.   0

cse:

TIME(s): total time      =   0.38   std. dev. 0.01
         user time       =   0.32   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.04   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.04   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 5654760   std. dev.   0
         bytes copied    =  348080   std. dev.   0
MEM(K):  max phys mem    =    1184   std. dev.   0
PAGING:  page reclaims   =     164   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   51596   std. dev.   0
         CodeSize        =  178356   std. dev.   0

hoist:

TIME(s): total time      =   0.40   std. dev. 0.01
         user time       =   0.34   std. dev. 0.01
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.04   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.04   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 5664372   std. dev.   0
         bytes copied    =  343816   std. dev.   0
MEM(K):  max phys mem    =    1152   std. dev.   0
PAGING:  page reclaims   =     160   std. dev.   4
         page faults     =       0   std. dev.   0
MISC:    GCTableSize     =   43176   std. dev.   0
         CodeSize        =  164860   std. dev.   0

switch:

TIME(s): total time      =   0.41   std. dev. 0.00
         user time       =   0.35   std. dev. 0.00
```

```
         sys time        =   0.02   std. dev. 0.00
         gc-all time     =   0.04   std. dev. 0.00
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.04   std. dev. 0.00
         gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced   =  5664272   std. dev.   0
         bytes copied    =   344640   std. dev.   0
MEM(K):  max phys mem    =     1184   std. dev.   0
PAGING:  page reclaims   =      165   std. dev.   4
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    47344   std. dev.   0
         CodeSize        =   170584   std. dev.   0

Ratios:

std opts + cmpelim / std opts :

TIME(s): total time      =  0.982
         user time       =  1.003
         sys time        =  0.725
         gc-all time     =  0.954
         majorgc-copy    = n.d.
         gc-copy time    =  0.965
         gc-stack time   =  0.612
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
         page faults     =   n.d.
MISC:    GCTableSize     =    0.999
         CodeSize        =    0.999

std opts + cse / std opts :

TIME(s): total time      =  0.893
         user time       =  0.887
         sys time        =  0.859
         gc-all time     =  0.974
         majorgc-copy    = n.d.
         gc-copy time    =  0.982
         gc-stack time   =  0.736
GC(k):   bytes alloced   =    0.998
         bytes copied    =    1.010
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    0.994
         page faults     =   n.d.
MISC:    GCTableSize     =    1.041
         CodeSize        =    0.996

std opts + hoist / std opts :

TIME(s): total time      =  0.948
         user time       =  0.954
         sys time        =  0.929
         gc-all time     =  0.929
         majorgc-copy    = n.d.
         gc-copy time    =  0.932
         gc-stack time   =  0.845
GC(k):   bytes alloced   =    1.000
         bytes copied    =    0.998
MEM(K):  max phys mem    =    0.973
PAGING:  page reclaims   =    0.970
         page faults     =   n.d.
MISC:    GCTableSize     =    0.871
         CodeSize        =    0.920

std opts + switch / std opts :

TIME(s): total time      =  0.981
         user time       =  0.979
         sys time        =  0.983
         gc-all time     =  0.996
         majorgc-copy    = n.d.
         gc-copy time    =  0.996
         gc-stack time   =  1.008
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
         page faults     =   n.d.
MISC:    GCTableSize     =    0.955
         CodeSize        =    0.952

std opts + allopts / std opts :

TIME(s): total time      =  0.873
         user time       =  0.866
         sys time        =  0.931
         gc-all time     =  0.910
         majorgc-copy    = n.d.
         gc-copy time    =  0.917
         gc-stack time   =  0.713
GC(k):   bytes alloced   =    0.962
         bytes copied    =    1.002
MEM(K):  max phys mem    =    0.966
PAGING:  page reclaims   =    0.963
         page faults     =   n.d.
MISC:    GCTableSize     =    0.791
         CodeSize        =    0.786

SIMPLE:

allopts:

TIME(s): total time      =   8.01   std. dev. 0.04
         user time       =   7.28   std. dev. 0.03
         sys time        =   0.11   std. dev. 0.01
         gc-all time     =   0.62   std. dev. 0.01
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.58   std. dev. 0.01
         gc-stack time   =   0.04   std. dev. 0.01
GC(k):   bytes alloced   = 322603348   std. dev.   0
         bytes copied    =  4551040   std. dev.   0
MEM(K):  max phys mem    =     9904   std. dev.   0
PAGING:  page reclaims   =     1257   std. dev.   3
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    30264   std. dev.   0
         CodeSize        =   138252   std. dev.   0

noloopopts:

TIME(s): total time      =  14.48   std. dev. 0.04
         user time       =  13.64   std. dev. 0.03
         sys time        =   0.12   std. dev. 0.01
         gc-all time     =   0.72   std. dev. 0.02
         majorgc-copy    =   0.00   std. dev. 0.00
         gc-copy time    =   0.65   std. dev. 0.02
         gc-stack time   =   0.07   std. dev. 0.01
GC(k):   bytes alloced   = 414037196   std. dev.   0
         bytes copied    =  4383064   std. dev.   0
MEM(K):  max phys mem    =     9752   std. dev.   0
PAGING:  page reclaims   =     1239   std. dev.   4
         page faults     =        0   std. dev.   0
MISC:    GCTableSize     =    40128   std. dev.   0
```

```
        CodeSize         =   167852   std. dev.   0

cmpelim:

TIME(s): total time      =  14.89   std. dev. 0.07
          user time      =  14.11   std. dev. 0.06
          sys time       =   0.13   std. dev. 0.01
          gc-all time    =   0.65   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.59   std. dev. 0.02
          gc-stack time  =   0.05   std. dev. 0.01
GC(k):   bytes alloced   = 414037196   std. dev.   0
         bytes copied    =  4383288   std. dev.   0
MEM(K):  max phys mem    =     9752   std. dev.   0
PAGING:  page reclaims   =     1239   std. dev.   4
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    40128   std. dev.   0
         CodeSize        =   167856   std. dev.   0

cse:

TIME(s): total time      =  10.92   std. dev. 0.06
          user time      =  10.18   std. dev. 0.05
          sys time       =   0.12   std. dev. 0.02
          gc-all time    =   0.62   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.56   std. dev. 0.02
          gc-stack time  =   0.06   std. dev. 0.01
GC(k):   bytes alloced   = 423571572   std. dev.   0
         bytes copied    =  4409796   std. dev.   0
MEM(K):  max phys mem    =     9760   std. dev.   0
PAGING:  page reclaims   =     1238   std. dev.   4
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    43160   std. dev.   0
         CodeSize        =   165472   std. dev.   0

hoist:

TIME(s): total time      =  10.19   std. dev. 0.38
          user time      =   9.39   std. dev. 0.36
          sys time       =   0.12   std. dev. 0.01
          gc-all time    =   0.68   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.63   std. dev. 0.02
          gc-stack time  =   0.05   std. dev. 0.01
GC(k):   bytes alloced   = 402813428   std. dev.   0
         bytes copied    =  4219072   std. dev.   0
MEM(K):  max phys mem    =     9576   std. dev.   0
PAGING:  page reclaims   =     1215   std. dev.   3
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    31492   std. dev.   0
         CodeSize        =   155088   std. dev.   0
switch:

TIME(s): total time      =  14.95   std. dev. 0.06
          user time      =  14.12   std. dev. 0.06
          sys time       =   0.12   std. dev. 0.01
          gc-all time    =   0.71   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.64   std. dev. 0.01
          gc-stack time  =   0.06   std. dev. 0.01
GC(k):   bytes alloced   = 414037196   std. dev.   0
         bytes copied    =  4384128   std. dev.   0
MEM(K):  max phys mem    =     9744   std. dev.   0
PAGING:  page reclaims   =     1237   std. dev.   4
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    39768   std. dev.   0
```

```
         CodeSize        =   166152   std. dev.   0

Ratios:

std opts + cmpelim / std opts :

TIME(s): total time      =  1.028
          user time      =  1.035
          sys time       =  1.117
          gc-all time    =  0.892
          majorgc-copy   = n.d.
          gc-copy time   =  0.914
          gc-stack time  =  0.708
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    1.000
PAGING:  page reclaims   =    1.000
          page faults    =   n.d.
MISC:    GCTableSize     =    1.000
         CodeSize        =    1.000

std opts + cse / std opts :

TIME(s): total time      =  0.754
          user time      =  0.746
          sys time       =  1.062
          gc-all time    =  0.853
          majorgc-copy   = n.d.
          gc-copy time   =  0.859
          gc-stack time  =  0.800
GC(k):   bytes alloced   =    1.023
         bytes copied    =    1.006
MEM(K):  max phys mem    =    1.001
PAGING:  page reclaims   =    0.999
          page faults    =   n.d.
MISC:    GCTableSize     =    1.076
         CodeSize        =    0.986

std opts + hoist / std opts :

TIME(s): total time      =  0.704
          user time      =  0.689
          sys time       =  1.029
          gc-all time    =  0.940
          majorgc-copy   = n.d.
          gc-copy time   =  0.976
          gc-stack time  =  0.635
GC(k):   bytes alloced   =    0.973
         bytes copied    =    0.963
MEM(K):  max phys mem    =    0.982
PAGING:  page reclaims   =    0.981
          page faults    =   n.d.
MISC:    GCTableSize     =    0.785
         CodeSize        =    0.924

std opts + switch / std opts :

TIME(s): total time      =  1.033
          user time      =  1.036
          sys time       =  1.065
          gc-all time    =  0.975
          majorgc-copy   = n.d.
          gc-copy time   =  0.993
          gc-stack time  =  0.816
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.999
```

```
PAGING:  page reclaims     =    0.998
          page faults      =   n.d.
MISC:    GCTableSize       =    0.991
         CodeSize          =    0.990

std opts + allopts / std opts :

TIME(s): total time        =  0.553
          user time        =  0.534
          sys time         =  0.950
          gc-all time      =  0.851
          majorgc-copy     = n.d.
          gc-copy time     =  0.892
          gc-stack time    =  0.490
GC(k):   bytes alloced     =    0.779
         bytes copied      =    1.038
MEM(K):  max phys mem      =    1.016
PAGING:  page reclaims     =    1.014
          page faults      =   n.d.
MISC:    GCTableSize       =    0.754
         CodeSize          =    0.824
```

B.3 Effects of loop optimizations on constructor computations

This section contains the measurements of the effects of individual loop optimizations when the optimizations are limited to constructor computations. The measurements are similar to the measurements in the previous section. I exclude switch elimination and comparison elimination, because these do not apply to constructor computations.

```
Checksum

allopts:

TIME(s): total time        =   2.20   std. dev. 0.17
          user time        =   2.11   std. dev. 0.16
          sys time         =   0.03   std. dev. 0.02
          gc-all time      =   0.06   std. dev. 0.01
          majorgc-copy     =   0.00   std. dev. 0.00
          gc-copy time     =   0.05   std. dev. 0.01
          gc-stack time    =   0.02   std. dev. 0.00
GC(k):   bytes alloced     = 143889328   std. dev.   0
         bytes copied      =        512   std. dev.   0
MEM(K):  max phys mem      =        800   std. dev.   2
PAGING:  page reclaims     =        117   std. dev.   2
          page faults      =          0   std. dev.   0
MISC:    GCTableSize       =      10652   std. dev.   0
         CodeSize          =      49048   std. dev.   0

noloopopts:

TIME(s): total time        =  15.96   std. dev. 0.05
          user time        =   5.58   std. dev. 0.02
          sys time         =   0.04   std. dev. 0.01
          gc-all time      =  10.35   std. dev. 0.03
          majorgc-copy     =   0.27   std. dev. 0.00
          gc-copy time     =  10.31   std. dev. 0.03
          gc-stack time    =   0.03   std. dev. 0.01
GC(k):   bytes alloced     = 390598672   std. dev.   0
         bytes copied      = 123364696   std. dev.   0
MEM(K):  max phys mem      =      12392   std. dev.   0
PAGING:  page reclaims     =       1566   std. dev.   3
          page faults      =          0   std. dev.   0
MISC:    GCTableSize       =      11068   std. dev.   0
         CodeSize          =      49724   std. dev.   0

cse:

TIME(s): total time        =   2.08   std. dev. 0.03
          user time        =   1.99   std. dev. 0.02
          sys time         =   0.02   std. dev. 0.01
          gc-all time      =   0.07   std. dev. 0.01
          majorgc-copy     =   0.00   std. dev. 0.00
          gc-copy time     =   0.05   std. dev. 0.01
          gc-stack time    =   0.02   std. dev. 0.00
GC(k):   bytes alloced     = 144009328   std. dev.   0
         bytes copied      =      60464   std. dev.   0
MEM(K):  max phys mem      =        856   std. dev.   2
PAGING:  page reclaims     =        124   std. dev.   2
          page faults      =          0   std. dev.   0
MISC:    GCTableSize       =      10700   std. dev.   0
         CodeSize          =      49024   std. dev.   0

hoist:

TIME(s): total time        =   2.01   std. dev. 0.03
          user time        =   1.93   std. dev. 0.02
          sys time         =   0.02   std. dev. 0.00
          gc-all time      =   0.07   std. dev. 0.01
          majorgc-copy     =   0.00   std. dev. 0.00
          gc-copy time     =   0.05   std. dev. 0.01
          gc-stack time    =   0.02   std. dev. 0.00
GC(k):   bytes alloced     = 143889760   std. dev.   0
         bytes copied      =        944   std. dev.   0
MEM(K):  max phys mem      =        817   std. dev.   2
PAGING:  page reclaims     =        120   std. dev.   2
          page faults      =          0   std. dev.   0
MISC:    GCTableSize       =      10972   std. dev.   0
         CodeSize          =      49604   std. dev.   0

invariant:

TIME(s): total time        =   1.99   std. dev. 0.01
          user time        =   1.90   std. dev. 0.01
          sys time         =   0.02   std. dev. 0.00
          gc-all time      =   0.06   std. dev. 0.01
          majorgc-copy     =   0.00   std. dev. 0.00
          gc-copy time     =   0.05   std. dev. 0.01
          gc-stack time    =   0.02   std. dev. 0.00
GC(k):   bytes alloced     = 143889760   std. dev.   0
         bytes copied      =        944   std. dev.   0
MEM(K):  max phys mem      =        816   std. dev.   2
PAGING:  page reclaims     =        120   std. dev.   2
          page faults      =          0   std. dev.   0
MISC:    GCTableSize       =      10972   std. dev.   0
         CodeSize          =      49480   std. dev.   0

Ratios:

std opts + cse / std opts :
```

```
TIME(s): total time      =  0.131
          user time      =  0.357
          sys time       =  0.616
          gc-all time    =  0.007
          majorgc-copy   =  0.000
          gc-copy time   =  0.005
          gc-stack time  =  0.601
GC(k):   bytes alloced   =    0.369
         bytes copied    =    0.000
MEM(K):  max phys mem    =    0.069
PAGING:  page reclaims   =    0.079
          page faults    =   n.d.
MISC:    GCTableSize     =    0.967
         CodeSize        =    0.986

std opts + hoist / std opts :

TIME(s): total time      =  0.126
          user time      =  0.345
          sys time       =  0.568
          gc-all time    =  0.006
          majorgc-copy   =  0.000
          gc-copy time   =  0.005
          gc-stack time  =  0.570
GC(k):   bytes alloced   =    0.368
         bytes copied    =    0.000
MEM(K):  max phys mem    =    0.066
PAGING:  page reclaims   =    0.077
          page faults    =   n.d.
MISC:    GCTableSize     =    0.991
         CodeSize        =    0.998

std opts + invariant / std opts :

TIME(s): total time      =  0.125
          user time      =  0.341
          sys time       =  0.513
          gc-all time    =  0.006
          majorgc-copy   =  0.000
          gc-copy time   =  0.005
          gc-stack time  =  0.519
GC(k):   bytes alloced   =    0.368
         bytes copied    =    0.000
MEM(K):  max phys mem    =    0.066
PAGING:  page reclaims   =    0.077
          page faults    =   n.d.
MISC:    GCTableSize     =    0.991
         CodeSize        =    0.995

std opts + allopts / std opts :

TIME(s): total time      =  0.138
          user time      =  0.378
          sys time       =  0.843
          gc-all time    =  0.006
          majorgc-copy   =  0.000
          gc-copy time   =  0.004
          gc-stack time  =  0.474
GC(k):   bytes alloced   =    0.368
         bytes copied    =    0.000
MEM(K):  max phys mem    =    0.065
PAGING:  page reclaims   =    0.075
          page faults    =   n.d.
MISC:    GCTableSize     =    0.962
         CodeSize        =    0.986
```

```
FFT

allopts:

TIME(s): total time      =  10.37   std. dev. 0.08
          user time      =  10.13   std. dev. 0.08
          sys time       =   0.08   std. dev. 0.01
          gc-all time    =   0.16   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.12   std. dev. 0.01
          gc-stack time  =   0.04   std. dev. 0.01
GC(k):   bytes alloced   = 403882364   std. dev.   0
         bytes copied    =     85432   std. dev.   0
MEM(K):  max phys mem    =      3072   std. dev.   0
PAGING:  page reclaims   =       400   std. dev.   3
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     20116   std. dev.   0
         CodeSize        =     68264   std. dev.   0

noloopopts:

TIME(s): total time      =  10.27   std. dev. 0.22
          user time      =  10.04   std. dev. 0.18
          sys time       =   0.07   std. dev. 0.01
          gc-all time    =   0.16   std. dev. 0.04
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.11   std. dev. 0.02
          gc-stack time  =   0.05   std. dev. 0.02
GC(k):   bytes alloced   = 403882364   std. dev.   0
         bytes copied    =     85432   std. dev.   0
MEM(K):  max phys mem    =      3072   std. dev.   0
PAGING:  page reclaims   =       400   std. dev.   3
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     20116   std. dev.   0
         CodeSize        =     68056   std. dev.   0

cse:

TIME(s): total time      =  10.85   std. dev. 0.80
          user time      =  10.64   std. dev. 0.80
          sys time       =   0.07   std. dev. 0.01
          gc-all time    =   0.15   std. dev. 0.01
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.11   std. dev. 0.01
          gc-stack time  =   0.04   std. dev. 0.01
GC(k):   bytes alloced   = 403882364   std. dev.   0
         bytes copied    =     85432   std. dev.   0
MEM(K):  max phys mem    =      3072   std. dev.   0
PAGING:  page reclaims   =       400   std. dev.   3
          page faults    =         0   std. dev.   0
MISC:    GCTableSize     =     20116   std. dev.   0
         CodeSize        =     67892   std. dev.   0

hoist:

TIME(s): total time      =  10.32   std. dev. 0.07
          user time      =  10.08   std. dev. 0.06
          sys time       =   0.07   std. dev. 0.01
          gc-all time    =   0.18   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.13   std. dev. 0.01
          gc-stack time  =   0.05   std. dev. 0.01
GC(k):   bytes alloced   = 403882364   std. dev.   0
         bytes copied    =     85432   std. dev.   0
MEM(K):  max phys mem    =      3072   std. dev.   0
PAGING:  page reclaims   =       400   std. dev.   3
          page faults    =         0   std. dev.   0
```

```
MISC:    GCTableSize      =    20116   std. dev.   0
         CodeSize         =    67944   std. dev.   0

invariant:

TIME(s): total time       =  10.27   std. dev. 0.08
          user time       =  10.05   std. dev. 0.07
          sys time        =   0.07   std. dev. 0.01
          gc-all time     =   0.15   std. dev. 0.02
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.11   std. dev. 0.01
          gc-stack time   =   0.04   std. dev. 0.01
GC(k):   bytes alloced    = 403882364   std. dev.   0
         bytes copied     =    85432   std. dev.   0
MEM(K):  max phys mem     =     3072   std. dev.   0
PAGING:  page reclaims    =      400   std. dev.   3
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    20116   std. dev.   0
         CodeSize         =    67964   std. dev.   0

Ratios:

std opts + cse / std opts :

TIME(s): total time       =  1.057
          user time       =  1.059
          sys time        =  1.075
          gc-all time     =  0.908
          majorgc-copy    = n.d.
          gc-copy time    =  0.930
          gc-stack time   =  0.860
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    0.998

std opts + hoist / std opts :

TIME(s): total time       =  1.005
          user time       =  1.003
          sys time        =  1.051
          gc-all time     =  1.105
          majorgc-copy    = n.d.
          gc-copy time    =  1.135
          gc-stack time   =  1.037
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    0.998

std opts + invariant / std opts :

TIME(s): total time       =  1.000
          user time       =  1.001
          sys time        =  1.029
          gc-all time     =  0.949
          majorgc-copy    = n.d.
          gc-copy time    =  0.966
          gc-stack time   =  0.913
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    0.999

std opts + allopts / std opts :

TIME(s): total time       =  1.010
          user time       =  1.008
          sys time        =  1.174
          gc-all time     =  1.016
          majorgc-copy    = n.d.
          gc-copy time    =  1.079
          gc-stack time   =  0.872
GC(k):   bytes alloced    =    1.000
         bytes copied     =    1.000
MEM(K):  max phys mem     =    1.000
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    1.000
         CodeSize         =    1.003

Knuth-Bendix

allopts:

TIME(s): total time       =   2.09   std. dev. 0.11
          user time       =   1.64   std. dev. 0.09
          sys time        =   0.03   std. dev. 0.01
          gc-all time     =   0.42   std. dev. 0.02
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.18   std. dev. 0.01
          gc-stack time   =   0.24   std. dev. 0.01
GC(k):   bytes alloced    = 46895360   std. dev.   0
         bytes copied     =  1768796   std. dev.   0
MEM(K):  max phys mem     =     2824   std. dev.   0
PAGING:  page reclaims    =      370   std. dev.   3
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    21864   std. dev.   0
         CodeSize         =    85636   std. dev.   0

noloopopts:

TIME(s): total time       =   2.85   std. dev. 0.04
          user time       =   2.28   std. dev. 0.04
          sys time        =   0.02   std. dev. 0.01
          gc-all time     =   0.55   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.26   std. dev. 0.01
          gc-stack time   =   0.29   std. dev. 0.01
GC(k):   bytes alloced    = 59744436   std. dev.   0
         bytes copied     =  2061016   std. dev.   0
MEM(K):  max phys mem     =     3168   std. dev.   0
PAGING:  page reclaims    =      415   std. dev.   3
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    22748   std. dev.   0
         CodeSize         =    88820   std. dev.   0

cse:

TIME(s): total time       =   2.14   std. dev. 0.02
          user time       =   1.64   std. dev. 0.02
          sys time        =   0.02   std. dev. 0.00
          gc-all time     =   0.48   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.21   std. dev. 0.00
```

```
          gc-stack time   =   0.26   std. dev. 0.01
GC(k):   bytes alloced    = 54826920   std. dev.   0
         bytes copied     =  2132664   std. dev.   0
MEM(K):  max phys mem     =     3184   std. dev.   0
PAGING:  page reclaims    =      415   std. dev.   3
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    22508   std. dev.   0
         CodeSize         =    88608   std. dev.   0

hoist:

TIME(s): total time       =   1.94   std. dev. 0.04
          user time       =   1.52   std. dev. 0.04
          sys time        =   0.02   std. dev. 0.00
          gc-all time     =   0.40   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.18   std. dev. 0.00
          gc-stack time   =   0.23   std. dev. 0.00
GC(k):   bytes alloced    = 46895360   std. dev.   0
         bytes copied     =  1768796   std. dev.   0
MEM(K):  max phys mem     =     2824   std. dev.   0
PAGING:  page reclaims    =      369   std. dev.   3
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    22304   std. dev.   0
         CodeSize         =    87064   std. dev.   0

invariant:

TIME(s): total time       =   2.21   std. dev. 0.06
          user time       =   1.73   std. dev. 0.06
          sys time        =   0.02   std. dev. 0.01
          gc-all time     =   0.46   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.19   std. dev. 0.01
          gc-stack time   =   0.27   std. dev. 0.01
GC(k):   bytes alloced    = 55088432   std. dev.   0
         bytes copied     =  1820072   std. dev.   0
MEM(K):  max phys mem     =     2880   std. dev.   0
PAGING:  page reclaims    =      376   std. dev.   3
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    21484   std. dev.   0
         CodeSize         =    85592   std. dev.   0

Ratios:

std opts + cse / std opts :

TIME(s): total time       =  0.748
          user time       =  0.718
          sys time        =  0.909
          gc-all time     =  0.865
          majorgc-copy    = n.d.
          gc-copy time    =  0.815
          gc-stack time   =  0.909
GC(k):   bytes alloced    =    0.918
         bytes copied     =    1.035
MEM(K):  max phys mem     =    1.005
PAGING:  page reclaims    =    1.000
          page faults     =   n.d.
MISC:    GCTableSize      =    0.989
         CodeSize         =    0.998

std opts + hoist / std opts :

TIME(s): total time       =  0.681
          user time       =  0.666
          sys time        =  0.901
          gc-all time     =  0.734
          majorgc-copy    = n.d.
          gc-copy time    =  0.680
          gc-stack time   =  0.782
GC(k):   bytes alloced    =    0.785
         bytes copied     =    0.858
MEM(K):  max phys mem     =    0.891
PAGING:  page reclaims    =    0.889
          page faults     =   n.d.
MISC:    GCTableSize      =    0.980
         CodeSize         =    0.980

std opts + invariant / std opts :

TIME(s): total time       =  0.776
          user time       =  0.756
          sys time        =  1.075
          gc-all time     =  0.844
          majorgc-copy    = n.d.
          gc-copy time    =  0.736
          gc-stack time   =  0.938
GC(k):   bytes alloced    =    0.922
         bytes copied     =    0.883
MEM(K):  max phys mem     =    0.909
PAGING:  page reclaims    =    0.906
          page faults     =   n.d.
MISC:    GCTableSize      =    0.944
         CodeSize         =    0.964

std opts + allopts / std opts :

TIME(s): total time       =  0.732
          user time       =  0.718
          sys time        =  1.263
          gc-all time     =  0.769
          majorgc-copy    = n.d.
          gc-copy time    =  0.709
          gc-stack time   =  0.821
GC(k):   bytes alloced    =    0.785
         bytes copied     =    0.858
MEM(K):  max phys mem     =    0.891
PAGING:  page reclaims    =    0.892
          page faults     =   n.d.
MISC:    GCTableSize      =    0.961
         CodeSize         =    0.964

Lexgen

allopts:

TIME(s): total time       =   2.06   std. dev. 0.06
          user time       =   1.95   std. dev. 0.05
          sys time        =   0.03   std. dev. 0.01
          gc-all time     =   0.07   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.07   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):   bytes alloced    =  8688968   std. dev.   0
         bytes copied     =   779436   std. dev.   0
MEM(K):  max phys mem     =     1872   std. dev.   0
PAGING:  page reclaims    =      252   std. dev.   4
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    54920   std. dev.   0
         CodeSize         =   203468   std. dev.   0

noloopopts:
```

```
TIME(s): total time      =   1.75   std. dev. 0.02
          user time      =   1.65   std. dev. 0.02
          sys time       =   0.03   std. dev. 0.00
          gc-all time    =   0.08   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.07   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8688980   std. dev.   0
         bytes copied    =  779448   std. dev.   0
MEM(K):  max phys mem    =    1920   std. dev.   0
PAGING:  page reclaims   =     258   std. dev.   4
          page faults    =       0   std. dev.   0
MISC:    GCTableSize     =   57828   std. dev.   0
         CodeSize        =  212152   std. dev.   0

cse:

TIME(s): total time      =   1.68   std. dev. 0.02
          user time      =   1.58   std. dev. 0.02
          sys time       =   0.03   std. dev. 0.00
          gc-all time    =   0.07   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.07   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8688968   std. dev.   0
         bytes copied    =  779436   std. dev.   0
MEM(K):  max phys mem    =    1872   std. dev.   0
PAGING:  page reclaims   =     252   std. dev.   4
          page faults    =       0   std. dev.   0
MISC:    GCTableSize     =   54872   std. dev.   0
         CodeSize        =  203780   std. dev.   0

hoist:

TIME(s): total time      =   2.03   std. dev. 0.01
          user time      =   1.93   std. dev. 0.01
          sys time       =   0.03   std. dev. 0.00
          gc-all time    =   0.07   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.07   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8688980   std. dev.   0
         bytes copied    =  779472   std. dev.   0
MEM(K):  max phys mem    =    1896   std. dev.   0
PAGING:  page reclaims   =     254   std. dev.   4
          page faults    =       0   std. dev.   0
MISC:    GCTableSize     =   57872   std. dev.   0
         CodeSize        =  210164   std. dev.   0

invariant:

TIME(s): total time      =   2.12   std. dev. 0.02
          user time      =   2.02   std. dev. 0.01
          sys time       =   0.03   std. dev. 0.01
          gc-all time    =   0.07   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.07   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 8688980   std. dev.   0
         bytes copied    =  779448   std. dev.   0
MEM(K):  max phys mem    =    1896   std. dev.   0
PAGING:  page reclaims   =     254   std. dev.   4
          page faults    =       0   std. dev.   0
MISC:    GCTableSize     =   57836   std. dev.   0
         CodeSize        =  210268   std. dev.   0

Ratios:
```

```
std opts + cse / std opts :

TIME(s): total time      =  0.958
          user time      =  0.959
          sys time       =  1.021
          gc-all time    =  0.904
          majorgc-copy   = n.d.
          gc-copy time   =  0.904
          gc-stack time  =  0.885
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.975
PAGING:  page reclaims   =    0.977
          page faults    =   n.d.
MISC:    GCTableSize     =    0.949
         CodeSize        =    0.961

std opts + hoist / std opts :

TIME(s): total time      =  1.159
          user time      =  1.173
          sys time       =  0.985
          gc-all time    =  0.894
          majorgc-copy   = n.d.
          gc-copy time   =  0.891
          gc-stack time  =  0.952
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.988
PAGING:  page reclaims   =    0.984
          page faults    =   n.d.
MISC:    GCTableSize     =    1.001
         CodeSize        =    0.991

std opts + invariant / std opts :

TIME(s): total time      =  1.211
          user time      =  1.228
          sys time       =  1.021
          gc-all time    =  0.893
          majorgc-copy   = n.d.
          gc-copy time   =  0.893
          gc-stack time  =  0.914
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.988
PAGING:  page reclaims   =    0.984
          page faults    =   n.d.
MISC:    GCTableSize     =    1.000
         CodeSize        =    0.991

std opts + allopts / std opts :

TIME(s): total time      =  1.174
          user time      =  1.184
          sys time       =  1.252
          gc-all time    =  0.931
          majorgc-copy   = n.d.
          gc-copy time   =  0.934
          gc-stack time  =  0.859
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.975
PAGING:  page reclaims   =    0.977
          page faults    =   n.d.
MISC:    GCTableSize     =    0.950
```

```
        CodeSize        =    0.959

Life

allopts:

TIME(s): total time      =   1.36   std. dev. 0.04
          user time      =   1.31   std. dev. 0.03
          sys time       =   0.03   std. dev. 0.01
          gc-all time    =   0.02   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.02   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20723992   std. dev.   0
         bytes copied    =    98780   std. dev.   0
MEM(K):  max phys mem    =      889   std. dev.   3
PAGING:  page reclaims   =      129   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    11672   std. dev.   0
         CodeSize        =    55008   std. dev.   0

noloopopts:

TIME(s): total time      =   1.43   std. dev. 0.00
          user time      =   1.32   std. dev. 0.00
          sys time       =   0.01   std. dev. 0.00
          gc-all time    =   0.10   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.09   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 21562840   std. dev.   0
         bytes copied    =   944480   std. dev.   0
MEM(K):  max phys mem    =     1745   std. dev.   3
PAGING:  page reclaims   =      236   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    12316   std. dev.   0
         CodeSize        =    57560   std. dev.   0

cse:

TIME(s): total time      =   1.33   std. dev. 0.07
          user time      =   1.29   std. dev. 0.07
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.02   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.02   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20723992   std. dev.   0
         bytes copied    =    98780   std. dev.   0
MEM(K):  max phys mem    =      897   std. dev.   3
PAGING:  page reclaims   =      129   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    11904   std. dev.   0
         CodeSize        =    55212   std. dev.   0

hoist:

TIME(s): total time      =   1.27   std. dev. 0.02
          user time      =   1.23   std. dev. 0.02
          sys time       =   0.02   std. dev. 0.00
          gc-all time    =   0.02   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.01   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20723992   std. dev.   0
         bytes copied    =    98780   std. dev.   0
MEM(K):  max phys mem    =      897   std. dev.   3
PAGING:  page reclaims   =      129   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    11832   std. dev.   0
         CodeSize        =    55576   std. dev.   0

invariant:

TIME(s): total time      =   1.29   std. dev. 0.06
          user time      =   1.26   std. dev. 0.06
          sys time       =   0.01   std. dev. 0.00
          gc-all time    =   0.02   std. dev. 0.00
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.01   std. dev. 0.00
          gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced   = 20723992   std. dev.   0
         bytes copied    =    98780   std. dev.   0
MEM(K):  max phys mem    =      905   std. dev.   3
PAGING:  page reclaims   =      131   std. dev.   2
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    12000   std. dev.   0
         CodeSize        =    55680   std. dev.   0

Ratios:

std opts + cse / std opts :

TIME(s): total time      =  0.927
          user time      =  0.976
          sys time       =  1.143
          gc-all time    =  0.223
          majorgc-copy   = n.d.
          gc-copy time   =  0.210
          gc-stack time  =  0.731
GC(k):   bytes alloced   =    0.961
         bytes copied    =    0.105
MEM(K):  max phys mem    =    0.514
PAGING:  page reclaims   =    0.547
          page faults    =   n.d.
MISC:    GCTableSize     =    0.967
         CodeSize        =    0.959

std opts + hoist / std opts :

TIME(s): total time      =  0.883
          user time      =  0.932
          sys time       =  1.158
          gc-all time    =  0.176
          majorgc-copy   = n.d.
          gc-copy time   =  0.154
          gc-stack time  =  0.886
GC(k):   bytes alloced   =    0.961
         bytes copied    =    0.105
MEM(K):  max phys mem    =    0.514
PAGING:  page reclaims   =    0.547
          page faults    =   n.d.
MISC:    GCTableSize     =    0.961
         CodeSize        =    0.966

std opts + invariant / std opts :

TIME(s): total time      =  0.900
          user time      =  0.952
          sys time       =  0.990
          gc-all time    =  0.181
          majorgc-copy   = n.d.
          gc-copy time   =  0.160
          gc-stack time  =  0.882
```

```
GC(k):    bytes alloced    =    0.961
          bytes copied     =    0.105
MEM(K):   max phys mem     =    0.519
PAGING:   page reclaims    =    0.555
           page faults     =   n.d.
MISC:     GCTableSize      =    0.974
          CodeSize         =    0.967

std opts + allopts / std opts :

TIME(s): total time       =  0.946
          user time       =  0.987
          sys time        =  2.073
          gc-all time     =  0.217
          majorgc-copy    = n.d.
          gc-copy time    =  0.190
          gc-stack time   =  1.209
GC(k):    bytes alloced    =    0.961
          bytes copied     =    0.105
MEM(K):   max phys mem     =    0.509
PAGING:   page reclaims    =    0.547
           page faults     =   n.d.
MISC:     GCTableSize      =    0.948
          CodeSize         =    0.956

Matmult

allopts:

TIME(s): total time       =   2.45   std. dev. 0.04
          user time       =   2.43   std. dev. 0.03
          sys time        =   0.02   std. dev. 0.01
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):    bytes alloced    =      344   std. dev.   0
          bytes copied     =        0   std. dev.   0
MEM(K):   max phys mem     =      600   std. dev.   0
PAGING:   page reclaims    =       93   std. dev.   2
           page faults     =        0   std. dev.   0
MISC:     GCTableSize      =     8904   std. dev.   0
          CodeSize         =    43084   std. dev.   0

noloopopts:

TIME(s): total time       =   2.45   std. dev. 0.02
          user time       =   2.44   std. dev. 0.02
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):    bytes alloced    =      344   std. dev.   0
          bytes copied     =        0   std. dev.   0
MEM(K):   max phys mem     =      600   std. dev.   0
PAGING:   page reclaims    =       93   std. dev.   2
           page faults     =        0   std. dev.   0
MISC:     GCTableSize      =     9052   std. dev.   0
          CodeSize         =    43788   std. dev.   0
cse:

TIME(s): total time       =   2.42   std. dev. 0.00
          user time       =   2.41   std. dev. 0.00
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):    bytes alloced    =      344   std. dev.   0
          bytes copied     =        0   std. dev.   0
MEM(K):   max phys mem     =      600   std. dev.   0
PAGING:   page reclaims    =       93   std. dev.   2
           page faults     =        0   std. dev.   0
MISC:     GCTableSize      =     8904   std. dev.   0
          CodeSize         =    43076   std. dev.   0

hoist:

TIME(s): total time       =   2.26   std. dev. 0.00
          user time       =   2.25   std. dev. 0.00
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):    bytes alloced    =      344   std. dev.   0
          bytes copied     =        0   std. dev.   0
MEM(K):   max phys mem     =      600   std. dev.   0
PAGING:   page reclaims    =       93   std. dev.   2
           page faults     =        0   std. dev.   0
MISC:     GCTableSize      =     9044   std. dev.   0
          CodeSize         =    43668   std. dev.   0

invariant:

TIME(s): total time       =   2.26   std. dev. 0.00
          user time       =   2.25   std. dev. 0.00
          sys time        =   0.01   std. dev. 0.00
          gc-all time     =   0.00   std. dev. 0.00
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.00   std. dev. 0.00
          gc-stack time   =   0.00   std. dev. 0.00
GC(k):    bytes alloced    =      344   std. dev.   0
          bytes copied     =        0   std. dev.   0
MEM(K):   max phys mem     =      600   std. dev.   0
PAGING:   page reclaims    =       93   std. dev.   2
           page faults     =        0   std. dev.   0
MISC:     GCTableSize      =     9044   std. dev.   0
          CodeSize         =    43596   std. dev.   0

Ratios:

std opts + cse / std opts :

TIME(s): total time       =  0.987
          user time       =  0.987
          sys time        =  0.939
          gc-all time     = n.d.
          majorgc-copy    = n.d.
          gc-copy time    = n.d.
          gc-stack time   = n.d.
GC(k):    bytes alloced    =    1.000
          bytes copied     =   n.d.
MEM(K):   max phys mem     =    1.000
PAGING:   page reclaims    =    1.000
           page faults     =   n.d.
MISC:     GCTableSize      =    0.984
          CodeSize         =    0.984

std opts + hoist / std opts :

TIME(s): total time       =  0.924
          user time       =  0.923
```

```
         sys time       =  1.047
         gc-all time    = n.d.
         majorgc-copy   = n.d.
         gc-copy time   = n.d.
         gc-stack time  = n.d.
GC(k):   bytes alloced  =    1.000
         bytes copied   =   n.d.
MEM(K):  max phys mem   =    1.000
PAGING:  page reclaims  =    1.000
         page faults    =   n.d.
MISC:    GCTableSize    =    0.999
         CodeSize       =    0.997
```

std opts + invariant / std opts :

```
TIME(s): total time     =  0.924
         user time      =  0.924
         sys time       =  0.963
         gc-all time    = n.d.
         majorgc-copy   = n.d.
         gc-copy time   = n.d.
         gc-stack time  = n.d.
GC(k):   bytes alloced  =    1.000
         bytes copied   =   n.d.
MEM(K):  max phys mem   =    1.000
PAGING:  page reclaims  =    1.000
         page faults    =   n.d.
MISC:    GCTableSize    =    0.999
         CodeSize       =    0.996
```

std opts + allopts / std opts :

```
TIME(s): total time     =  0.999
         user time      =  0.996
         sys time       =  1.493
         gc-all time    = n.d.
         majorgc-copy   = n.d.
         gc-copy time   = n.d.
         gc-stack time  = n.d.
GC(k):   bytes alloced  =    1.000
         bytes copied   =   n.d.
MEM(K):  max phys mem   =    1.000
PAGING:  page reclaims  =    1.000
         page faults    =   n.d.
MISC:    GCTableSize    =    0.984
         CodeSize       =    0.984
```

PIA

allopts:

```
TIME(s): total time     =   0.40   std. dev. 0.01
         user time      =   0.35   std. dev. 0.01
         sys time       =   0.02   std. dev. 0.00
         gc-all time    =   0.04   std. dev. 0.00
         majorgc-copy   =   0.00   std. dev. 0.00
         gc-copy time   =   0.04   std. dev. 0.00
         gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced  = 5664272   std. dev.   0
         bytes copied   =  344632   std. dev.   0
MEM(K):  max phys mem   =    1328   std. dev.   0
PAGING:  page reclaims  =     183   std. dev.   4
         page faults    =       0   std. dev.   0
MISC:    GCTableSize    =   48292   std. dev.   0
         CodeSize       =  175144   std. dev.   0
```

noloopopts:

```
TIME(s): total time     =   0.41   std. dev. 0.00
         user time      =   0.35   std. dev. 0.00
         sys time       =   0.02   std. dev. 0.00
         gc-all time    =   0.04   std. dev. 0.00
         majorgc-copy   =   0.00   std. dev. 0.00
         gc-copy time   =   0.04   std. dev. 0.00
         gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced  = 5664272   std. dev.   0
         bytes copied   =  344648   std. dev.   0
MEM(K):  max phys mem   =    1360   std. dev.   0
PAGING:  page reclaims  =     188   std. dev.   4
         page faults    =       0   std. dev.   0
MISC:    GCTableSize    =   49580   std. dev.   0
         CodeSize       =  179132   std. dev.   0
```

cse:

```
TIME(s): total time     =   0.40   std. dev. 0.00
         user time      =   0.34   std. dev. 0.00
         sys time       =   0.02   std. dev. 0.00
         gc-all time    =   0.04   std. dev. 0.00
         majorgc-copy   =   0.00   std. dev. 0.00
         gc-copy time   =   0.04   std. dev. 0.00
         gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced  = 5664272   std. dev.   0
         bytes copied   =  344640   std. dev.   0
MEM(K):  max phys mem   =    1344   std. dev.   0
PAGING:  page reclaims  =     185   std. dev.   4
         page faults    =       0   std. dev.   0
MISC:    GCTableSize    =   48244   std. dev.   0
         CodeSize       =  175232   std. dev.   0
```

hoist:

```
TIME(s): total time     =   0.40   std. dev. 0.00
         user time      =   0.34   std. dev. 0.00
         sys time       =   0.02   std. dev. 0.00
         gc-all time    =   0.04   std. dev. 0.00
         majorgc-copy   =   0.00   std. dev. 0.00
         gc-copy time   =   0.04   std. dev. 0.00
         gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced  = 5664272   std. dev.   0
         bytes copied   =  344632   std. dev.   0
MEM(K):  max phys mem   =    1320   std. dev.   0
PAGING:  page reclaims  =     182   std. dev.   4
         page faults    =       0   std. dev.   0
MISC:    GCTableSize    =   48280   std. dev.   0
         CodeSize       =  175400   std. dev.   0
```

invariant:

```
TIME(s): total time     =   0.40   std. dev. 0.00
         user time      =   0.35   std. dev. 0.00
         sys time       =   0.02   std. dev. 0.00
         gc-all time    =   0.04   std. dev. 0.00
         majorgc-copy   =   0.00   std. dev. 0.00
         gc-copy time   =   0.04   std. dev. 0.00
         gc-stack time  =   0.00   std. dev. 0.00
GC(k):   bytes alloced  = 5664272   std. dev.   0
         bytes copied   =  344648   std. dev.   0
MEM(K):  max phys mem   =    1320   std. dev.   0
PAGING:  page reclaims  =     182   std. dev.   4
         page faults    =       0   std. dev.   0
MISC:    GCTableSize    =   48372   std. dev.   0
         CodeSize       =  175464   std. dev.   0
```

Ratios:

```
std opts + cse / std opts :

TIME(s): total time      =  0.973
          user time      =  0.972
          sys time       =  1.033
          gc-all time    =  0.972
          majorgc-copy   = n.d.
          gc-copy time   =  0.975
          gc-stack time  =  0.824
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.988
PAGING:  page reclaims   =    0.984
          page faults    =   n.d.
MISC:    GCTableSize     =    0.973
         CodeSize        =    0.978

std opts + hoist / std opts :

TIME(s): total time      =  0.980
          user time      =  0.983
          sys time       =  0.993
          gc-all time    =  0.960
          majorgc-copy   = n.d.
          gc-copy time   =  0.958
          gc-stack time  =  1.033
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.971
PAGING:  page reclaims   =    0.968
          page faults    =   n.d.
MISC:    GCTableSize     =    0.974
         CodeSize        =    0.979

std opts + invariant / std opts :

TIME(s): total time      =  0.987
          user time      =  0.989
          sys time       =  1.029
          gc-all time    =  0.952
          majorgc-copy   = n.d.
          gc-copy time   =  0.951
          gc-stack time  =  1.022
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.971
PAGING:  page reclaims   =    0.968
          page faults    =   n.d.
MISC:    GCTableSize     =    0.976
         CodeSize        =    0.980

std opts + allopts / std opts :

TIME(s): total time      =  0.987
          user time      =  0.984
          sys time       =  1.093
          gc-all time    =  0.970
          majorgc-copy   = n.d.
          gc-copy time   =  0.970
          gc-stack time  =  0.934
GC(k):   bytes alloced   =    1.000
         bytes copied    =    1.000
MEM(K):  max phys mem    =    0.976
PAGING:  page reclaims   =    0.973
          page faults    =   n.d.
MISC:    GCTableSize     =    0.974
         CodeSize        =    0.978

SIMPLE

allopts:

TIME(s): total time      =  10.67   std. dev. 0.06
          user time      =   9.90   std. dev. 0.05
          sys time       =   0.13   std. dev. 0.02
          gc-all time    =   0.64   std. dev. 0.04
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.58   std. dev. 0.03
          gc-stack time  =   0.06   std. dev. 0.01
GC(k):   bytes alloced   = 402813284   std. dev.   0
         bytes copied    = 4218960   std. dev.   0
MEM(K):  max phys mem    =     9752   std. dev.   0
PAGING:  page reclaims   =     1238   std. dev.   4
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    37988   std. dev.   0
         CodeSize        =   168808   std. dev.   0

noloopopts:

TIME(s): total time      =  14.57   std. dev. 0.16
          user time      =  13.80   std. dev. 0.12
          sys time       =   0.12   std. dev. 0.02
          gc-all time    =   0.65   std. dev. 0.04
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.58   std. dev. 0.03
          gc-stack time  =   0.06   std. dev. 0.01
GC(k):   bytes alloced   = 414037196   std. dev.   0
         bytes copied    = 4384120   std. dev.   0
MEM(K):  max phys mem    =     9928   std. dev.   0
PAGING:  page reclaims   =     1262   std. dev.   4
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    40128   std. dev.   0
         CodeSize        =   167796   std. dev.   0

cse:

TIME(s): total time      =  11.72   std. dev. 0.10
          user time      =  10.98   std. dev. 0.08
          sys time       =   0.11   std. dev. 0.01
          gc-all time    =   0.62   std. dev. 0.03
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.57   std. dev. 0.03
          gc-stack time  =   0.06   std. dev. 0.01
GC(k):   bytes alloced   = 413114228   std. dev.   0
         bytes copied    = 4371264   std. dev.   0
MEM(K):  max phys mem    =     9888   std. dev.   0
PAGING:  page reclaims   =     1255   std. dev.   4
          page faults    =        0   std. dev.   0
MISC:    GCTableSize     =    39068   std. dev.   0
         CodeSize        =   170004   std. dev.   0

hoist:

TIME(s): total time      =  10.26   std. dev. 0.36
          user time      =   9.43   std. dev. 0.35
          sys time       =   0.13   std. dev. 0.02
          gc-all time    =   0.69   std. dev. 0.02
          majorgc-copy   =   0.00   std. dev. 0.00
          gc-copy time   =   0.64   std. dev. 0.02
          gc-stack time  =   0.05   std. dev. 0.01
GC(k):   bytes alloced   = 402813284   std. dev.   0
         bytes copied    = 4218992   std. dev.   0
MEM(K):  max phys mem    =     9752   std. dev.   0
```

```
PAGING:  page reclaims    =     1239   std. dev.   4
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    40548   std. dev.   0
         CodeSize         =   172788   std. dev.   0

invariant:

TIME(s): total time       =  10.42   std. dev. 0.03
          user time       =   9.61   std. dev. 0.03
          sys time        =   0.12   std. dev. 0.01
          gc-all time     =   0.69   std. dev. 0.01
          majorgc-copy    =   0.00   std. dev. 0.00
          gc-copy time    =   0.64   std. dev. 0.01
          gc-stack time   =   0.04   std. dev. 0.01
GC(k):   bytes alloced    = 403735964   std. dev.   0
         bytes copied     = 4265224   std. dev.   0
MEM(K):  max phys mem     =     9800   std. dev.   0
PAGING:  page reclaims    =     1245   std. dev.   4
          page faults     =        0   std. dev.   0
MISC:    GCTableSize      =    40788   std. dev.   0
         CodeSize         =   172652   std. dev.   0

Ratios:

std opts + cse / std opts :
TIME(s): total time       =  0.804
          user time       =  0.795
          sys time        =  0.918
          gc-all time     =  0.966
          majorgc-copy    = n.d.
          gc-copy time    =  0.973
          gc-stack time   =  0.897
GC(k):   bytes alloced    =    0.998
         bytes copied     =    0.997
MEM(K):  max phys mem     =    0.996
PAGING:  page reclaims    =    0.994
          page faults     =   n.d.
MISC:    GCTableSize      =    0.974
         CodeSize         =    1.013

std opts + hoist / std opts :

TIME(s): total time       =  0.704
          user time       =  0.683
          sys time        =  1.079
          gc-all time     =  1.068
          majorgc-copy    = n.d.
          gc-copy time    =  1.100
          gc-stack time   =  0.784
GC(k):   bytes alloced    =    0.973
         bytes copied     =    0.962
MEM(K):  max phys mem     =    0.982
PAGING:  page reclaims    =    0.982
          page faults     =   n.d.
MISC:    GCTableSize      =    1.010
         CodeSize         =    1.030

std opts + invariant / std opts :

TIME(s): total time       =  0.715
          user time       =  0.696
          sys time        =  0.955
          gc-all time     =  1.062
          majorgc-copy    = n.d.
          gc-copy time    =  1.106
          gc-stack time   =  0.668
GC(k):   bytes alloced    =    0.975
         bytes copied     =    0.973
MEM(K):  max phys mem     =    0.987
PAGING:  page reclaims    =    0.987
          page faults     =   n.d.
MISC:    GCTableSize      =    1.016
         CodeSize         =    1.029

std opts + allopts / std opts :

TIME(s): total time       =  0.732
          user time       =  0.718
          sys time        =  1.015
          gc-all time     =  0.982
          majorgc-copy    = n.d.
          gc-copy time    =  0.989
          gc-stack time   =  0.920
GC(k):   bytes alloced    =    0.973
         bytes copied     =    0.962
MEM(K):  max phys mem     =    0.982
PAGING:  page reclaims    =    0.981
          page faults     =   n.d.
MISC:    GCTableSize      =    0.947
         CodeSize         =    1.006
```

Bibliography

[1] *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991. ACM.

[2] *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993. ACM.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley Publishing Company, 1986.

[4] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[5] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software — Practice and Experience*, 19(2):171–184, February 1989.

[6] Andrew W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[7] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[8] Andrew W. Appel. Personal communication. March 22 1993.

[9] Andrew W. Appel and Trevor Jim. Making lambda calculus smaller, faster. *Journal of Functional Programming*, 1995. accepted for publication.

[10] Andrew W. Appel and Trevor Y. Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM.

[11] Andrew W. Appel, James S. Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[12] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *19th Symposium on Principles of Programming Languages*. ACM, January 1992.

[13] J.M. Barth. A practical interprocedural data flow analysis algorithm. *Journal of the ACM*, 21(9):724–736, September 1978.

[14] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 55–64, Orlando, Florida, June 1994. ACM.

[15] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Tur ner. The ML Kit, Version 1. Technical Report 93/14, DIKU, 1993.

[16] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* [2], pages 197–206.

[17] Dianne Ellen Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[18] David R. Chase. Safety considerations for storage allocation optimizations. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 23(7):1–10, July 1988.

[19] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Fourteenth Symposium on Operating System Principles*. ACM, December 1993.

[20] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[21] Frederick C. Chow. *A Portable Machine-Independent Global Optimizer — Design and Measurements*. PhD thesis, Computer Systems Laboratory, Stanford University, December 1983. Technical Report No. 83-254.

[22] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[23] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, February 1978.

[24] Cypress Semiconductor, Ross Technology Subsidiary. *SPARC RISC User's Guide*, second edition, February 1990.

[25] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In LFP '92 [57], pages 299–310.

[26] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the 17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 1990. ACM.

[27] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado, 1993.

[28] Digital Equipment Corporation. *DS5000/200 KN02 System Module Functional Specification.*

[29] Digital Equipment Corporation, Palo Alto, CA. *DECStation 3100 Desktop Workstation Function Specification*, 1.3 edition, August 1990.

[30] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. SIGPLAN, ACM Press.

[31] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[32] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[33] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* [2], pages 237–247.

[34] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. Fx-87 reference manual. Technical Report MIT/LCS/TR-407, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.

[35] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 177–186, Albuquerque, New Mexico, June 1993. ACM.

[36] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[37] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995. ACM.

[38] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU-CS-96-136R, School of Computer Science, Carnegie Mellon University, September 1996.

[39] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1992.

[40] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77. ACM, June 1990.

[41] Mark D. Hill. A case for direct mapped caches. *Computer*, 21(12):25–40, December 1988.

[42] M.D. Hill and A.J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[43] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

[44] Neil D. Jones. Flow analysis of lambda expressions.

[45] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, San Diego, California, May 1993.

[46] Pierre Jouvelot. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages* [1], pages 303–310.

[47] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[48] Richard A. Kelsey. *Compilation By Program Transformation*. PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, May 1989.

[49] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *Transactions on Computer Systems*, 10(4):338–359, November 1992.

[50] Philip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache behavior of combinator graph reduction. *Transactions on Programming Languages and Systems*, 14(2):265–277, April 1992.

[51] David Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, February 1988.

[52] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

[53] James R. Larus. Abstract Execution: A technique for efficiently tracing programs. *Software Practice and Experience*, 20(12):1241–1258, December 1990.

[54] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report Wis 1083, Computer Sciences Department, University of Wisconsin-Madison, March 1992.

[55] Xavier Leroy. Polymorphic type inference and assignment. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages* [1], pages 291–302.

[56] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, January 1992.

[57] *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California, June 1992. ACM.

[58] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[59] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1987.

[60] I. A. Mason and C. L. Talcott. Program transformation via contextual assertions. In N. D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language, and Computation: Festschrift in Honor of Satoru Takasu*, number 792 in Lecture Notes in Computer Science, pages 225–254. Springer-Verlag, 1994.

[61] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[62] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*. ACM, January 1996. To appear.

[63] J. Gregory Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1995. Published as Technical Report CMU-CS-95-226.

[64] Anne Neirynck. *Analysis of Side Effects in Higher-Order Languages*. PhD thesis, Cornell University, Cornell, New York, 1988.

[65] Anne Neirynck, Prakash Panangaden, and Alan Demers. Effect analysis of higher-order languages. *International Journal of Parallel Programming*, 18(119), 119.

[66] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[67] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK, 1991.

[68] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[69] Simon Peyton-Jones. Compilation by transformation: a report from the trenches. In *European Symposium on Programming (ESOP '96), Lecture Notes in Computer Science 1058*. Springer Verlag, 1996.

[70] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[71] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[72] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, Massachusetts, 1989.

[73] Mark B. Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. PhD thesis, Laboratory for Computer Science, MIT, September 1993.

[74] John H. Reppy. Asynchronous Signals in Standard ML. Technical Report 90-1144, Department of Computer Science, Cornell University, August 1990.

[75] Stephen Richardson and Mahadevan Ganapathi. Interprocedural optimization: experimental results. *Software — Practice and Experience*, 19(2):149–168, February 1989.

[76] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–226, May 1979.

[77] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuations-passing style. In LFP '92 [57], pages 288–298.

[78] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, November 1993.

[79] Andre Santos. *Compilation by transformation in non-strict functional languages*. PhD thesis, Department of Computing Science, Glasgow University, Glasgow, Scotland, 1995.

[80] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[81] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton University, Princeton, New Jersey, November 1994.

[82] Olin Shivers. Control Flow Analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. ACM, June 1988.

[83] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991.

[84] Michael Slater. PA workstations set price/performance records. *Microprocessor Report*, 5(6):1, April 1991.

[85] Guy L. Steele Jr. RABBIT: A Compiler for Scheme (A Study in Compiler Optimization). Master's thesis, AI Laboratory Technical Report AI-TR-474, Massachusetts Institute of Technology, May 1978.

[86] Peter Steenkiste. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford,CA 94305, March 1987.

[87] Darko Stefanović and Eliot Moss. Characterisation of object behavior in Standard ML of New Jersey. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, 1994.

[88] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region,and effect inference. *Journal of Functional Programming*, 2(3):245–272, 1992.

[89] David Tarditi and Andrew W. Appel. ML-YACC, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[90] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, FL, June 1994.

[91] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[92] David Ungar. *The design and evaluation of a high performance Smalltalk system*. ACM Distinguished Dissertation. MIT Press, Cambridge, Massachusetts, 1987.

[93] Mitchell Wand and Zheng-Yu Wang. Conditional lambda-theories and the verification of static properties of programs. *Information and Computation*, 113(2):253–277, 1994.

[94] Kevin G. Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, August 1990.

[95] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1982.

[96] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection: a case for large and set-associative caches. Technical Report EECS-90-5, University of Illinios at Chicago, December 1990.

[97] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, California, June 1992.

[98] Andrew K. Wright. simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[99] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.

[100] Benjamin G. Zorn. *Comparative Performance evaluation of garbage collection algorithms*. PhD thesis, University of California, Berkeley, CA 94720, December 1989.