COMPILING STANDARD ML FOR EFFICIENT EXECUTION ON MODERN MACHINES

Zhong Shao

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

November 1994

Abstract

Many language theoreticians have taken great efforts in designing higher-level programming languages that are more elegant and more expressive than conventional languages. However, few of these new languages have been implemented very efficiently. The result is that most software engineers still prefer to use conventional languages, even though the new higher-level languages offer a better and simpler programming model.

This dissertation concentrates on improving the performance of programs written in Standard ML (SML)—a statically typed functional language—on today's RISC machines. SML poses tough challenges to efficient implementations: very frequent function calls, polymorphic types, recursive data structures, higher-order functions, and first-class continuations. This dissertation presents the design and evaluation of several new compilation techniques that meet these challenges by taking advantage of some of the higher-level language features in SML.

Type-directed compilation exploits the use of compile-time type information to optimize data representations and function calling conventions. By inserting coercions at each type instantiation and abstraction site, data objects in SML can use the same unboxed representations as in C, even with the presence of polymorphic functions. Measurements show that a simple set of type-based optimizations improves the performance of the non-type-based compiler by about 19% on a DECstation 5000.

Space-efficient closure representations utilizes the compile-time control and data flow information to optimize closure representations. By extensive closure sharing and allocating as many closures in registers as possible, the new *closure conversion* algorithm achieves very good asymptotic space usage, and improves the performance of the old compiler by about 14% on a DECstation 5000, *even without using a stack*. Further empirical and analytic studies show that the execution cost of stack-allocated and heap-allocated activation records is similar, but heap allocation is simpler to implement and allows very efficient first-class continuations.

Unrolling lists takes advantage of the higher-level language abstraction in SML to support more efficient representations for lists. By representing each *cons* cell using multiple *car* fields and one *cdr* field, the *unrolled list* reduces the memory used for links and significantly shortens the length of control-dependence and data-dependence chains in operations on lists.

Acknowledgements

First and foremost, I would like to thank my advisor Andrew Appel for the huge amount of effort he spent in guiding my research. During the last five years, he has always been there to share his quick wit, to clarify and enhance my thinking, and to spark new ideas. Without his careful guidance, this work would be impossible. This dissertation should really be considered as a joint work with him. In addition, I thank him for helping improve my writing and hacking skills, and for teaching me everything on how to become a good scientist.

I would also like to thank my readers John Reppy and Doug Clark for providing timely suggestions and valuable comments on this document. John gave this dissertation the kind of close reading that one's work is rarely privileged to receive; I thank him for his time and patience in refining my thoughts and correcting my syntax. I also want to thank David Hanson and Anne Rogers for serving on my thesis committee and for their interests in my work.

Special thanks go to David MacQueen, John Reppy, and Lal George at AT&T Bell Laboratories for the encouragement and advice they provided at various stages of this work. Dave introduced me to the world of programming language theory in his spring 1990 course at Princeton, and I became greatly interested in Standard ML since then. John and Lal taught me much about other aspects of the SML/NJ compiler. John also actively participated in the research described in Chapter 6. I thank them for making the SML/NJ compiler be such a pleasant project to work on.

Many people in many places have helped in many ways: I am grateful to Kai Li and Doug Clark for their illuminating computer architecture classes; to Kai Li, David Dobkin, David Hanson, and Anne Rogers for helping me decide my thesis topics and teaching me what good research should really be like; to Hans Boehm and John Ellis for giving me the opportunity to work at Xerox PARC (in summer 1993) where I learned many things outside the SML/NJ world; to Gün Sirer and Marcelo Gonçalves for implementing and adapting the MIPSI instruction simulator that I used in the measurements in Chapter 5; to Hans Boehm, Scott Burson, Amer Diwan, Damien Doligez, Lorenz Huelsbergen, Trevor Jim, Xavier Leroy, Paul Wilson for many useful comments on the early drafts of Chapter 4 and 5; to all the people in the administrative offices at Princeton, particularly Melissa Lawson for helping with all the paper work, Sharon Rodgers for helping me settle down when I first arrived at Princeton, and Jim Roberts and Matthew Norcross for providing me with more memory and disk space for running the "memory-eating" SML/NJ job.

Many thanks to Matthias Blume, Dimitrios Gunopulos, and Jeno Torocsik for being my wonderful officemates and for playing the water-gun fight with me; to my friend and English tutor Michael Dorn for patiently improving my English writing skills and for sharing many interesting discussions; to many friends here at Princeton—Richard Alpert, Alvaro Campos, Pei Cao, Stefanos Damianakis, Xue Fang, Yan Huo, S. V. Krishnan, Chen Lin, Zicheng Liu, Qiang Ren, Bin Wei, Jenny Zhao, and Jiaping Zhong, to name just a few—for making the quiet and boring town a lot more fun to live and work.

Finally, I want to thank my parents, my sister Hong, and my brother-in-law Zhenyu for their constant love and support. My deepest thanks go to my wife Xiaoguang for giving me great emotional support through her love, patience, and understanding. Xiaoguang was always able to cheer me up when my paper got rejected. Without the happiness and self-confidence that she instilled in me, this work would be impossible.

This research was funded, in part, by the National Science Foundation under grants CCR-8914570, CCR-9002786, CCR 9200790.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The most important component in software development is *programming*—the formalization of ideas and their expression in forms suitable for interpretation by computers. In programming, we start with a mental picture of some computational behavior, refine and clarify the idea over and over as its ramifications are better understood, and finally write out the formalized detail in a *programming language*—a formal notation designed for specifying such computational behaviors. The resulting program, after being translated into machine instructions, can then be executed on real machines to perform the desired task.

The ultimate goal of software research is to find the right programming model in which one can write the "best quality" (reliable, efficient, portable, etc.) program with the least amount of time and effort. Many researchers in programming languages believe that the most effective way to achieve this goal is to write programs in higher-level languages (i.e., languages that use more abstract representations and allow greater distance from low-level machines). In fact, most higher-level languages support simpler and cleaner programming models—making it easier to develop large and complex software. However, because higher-level languages are abstracted further away from real machines, they are difficult to implement as efficiently as conventional languages. As a result, most software engineers—unwilling to sacrifice code efficiency of their products—still prefer to write programs using conventional languages such as C and C++.

I believe that new higher-level languages, if designed carefully, can be implemented very efficiently even on modern machines. This dissertation documents a set of new compilation techniques that significantly improve the performance of programs written in Standard ML (SML)—a statically typed functional language—on today's RISC machines. Like many other higher-level languages, SML poses tough challenges to efficient implementations: very frequent function calls, polymorphic types, recursive data structures, higher-order functions, and first-class continuations. This dissertation presents the design and evaluation of several new techniques that meet these challenges by taking advantage of some of the higher-level language features in SML.

1.1 Motivation

It has been nearly four decades since the first programming languages, such as Fortran and Lisp, came into the programming world. Thousands of programming languages have been designed and implemented since then. Among these languages, there are imperative languages such as Fortran, Algol, Pascal, Ada, and C; object-oriented languages such as C++, Modula-3, and Smalltalk; functional languages such as Lisp, Scheme, ML, and Haskell; logic programming languages such as Prolog. While all language designers and implementors agree that they have the same ultimate goal—finding a programming model in which one can write the "best quality" software with the least amount of time and effort, language research has been carried out in two rather different directions:

- At one end, language theoreticians design new higher-level languages that have cleaner semantics; they usually do not care much about very efficient implementations;
- At the other end, most software in today's industrial world is still written using conventional languages such as C; most software engineers still prefer to use C and C++ simply because they have much more efficient implementations (and easy interface to libraries and operating system) .

This difference is also reflected in the dilemma that many people encounter when choosing which languages to use. Conventional languages such as C and C++ are very popular; they run very fast and use little memory. They are "hacker's heaven" because one can arbitrarily manipulate the machine level data representations in the source program; because of this, programs written in these languages might core-dump, and they are difficult to debug and difficult to reason about. On the other hand, higher-level languages such as ML are relatively less well known; they run slower and consume much memory. But they are much cleaner and safer than C and C++. Moreover, they often have very well founded semantics, and are easy to reason about and easy to write code in. Clearly, we want to have the best of both worlds, that is, to find languages that have both clean programming models and also efficient implementations.

Perhaps the best way to settle this dilemma is to develop new compilation techniques to make higher-level languages run faster. In the long term, the research centered around this can at least provide three benefits:

- If higher-level languages can be compiled as efficiently as conventional languages such as C and C++, we will be a step closer towards the "ideal", that is, to achieve the ultimate goal of software research.

- Because programs written in higher-level languages have very different run time behaviors (e.g., more allocations, fewer side-effects) from those in conventional languages, new compiler technologies discovered for higher-level languages may offer new insights on what is the best way to achieve successful high-performance computer systems in the future.

- Doing research on how to compile higher-level languages efficiently can give language theoreticians more feedback on what language features are very difficult to compile and can give hardware designers more feedback on what support modern machines can offer in order to achieve the best performance.

This dissertation concentrates on improving the performance of programs written in Standard ML (SML) [MTH90] on modern RISC machines. SML may not be the most representative or the most elegant higher-level language in today's world, but it serves as a great test bed to explore modern compilation technologies. There are many reasons to believe that SML can be compiled to run efficiently:

- SML is a statically typed language, that is, all type checking is done at compile time. This means that type tags need not be carried around at run time, and operators need not to check the types of their arguments at run time.

- SML uses call-by-value evaluation semantics, so SML programs still have understandable control flow to do all conventional data flow and control flow optimizations [ASU86].

- Unlike pure functional languages such as Haskell [HJ*et al*92], SML is just a mostly functional language. Side-effects are still allowed in SML, making it possible to support very efficient mutable data structures such as arrays and hash tables.

- We already have an efficient compiler for SML—the Standard ML of New Jersey compiler (SML/NJ) [AM91]—with which to explore new compiler optimizations.

However, SML contains many contemporary language features that are rarely seen in conventional languages:

Frequent function calls Almost all control structures in SML are expressed using function applications. For example, loops in conventional languages are expressed as recursive functions in SML.

Higher-order functions As in Scheme and other languages derived from the λ-calculus, functions in SML are first-class values that may be passed as arguments, returned as values, and put into data structures.

Polymorphic types SML allows polymorphic functions and data structures; that is, a function may take arguments of arbitrary type if in fact the function does not depend on that type. For example, the same `map` function can operate on a list of anything, and similarly for other common list functions such as `cons`, `length`, and `append`.

Immutable recursive data structures Recursive data structures are declared using concrete data type definitions. Like many pure functional languages, SML encourages heavy use of immutable data structures. For example, in order to insert an element into the end of a list, in SML, one has to copy the list rather than directly attaching the new element at the end.

First-class continuations SML[1] supports *call-with-current-continuation* (call/cc)—a primitive often used to support tasking, coroutines, exceptions, and so on.

Because compiler technologies developed for conventional languages do not apply to these new language features, the major challenge for compiling SML efficiently is naturally to find new techniques that can successfully deal with these new features. This dissertation presents the design and evaluation of several such new techniques that significantly improve the performance of SML programs:

- *Type-directed compilation* exploits the use of compile-time type information to optimize data representations and function calling conventions. By inserting coercions at each type instantiation and abstraction site, data objects in SML can use the same unboxed representations as in C, even with the presence of polymorphic functions. Measurements show that a simple set of type-based optimizations improve the performance of the non-type-based compiler by about 19% on a DECstation 5000.

[1] It is really the Standard ML of New Jersey dialect that supports first-class continuations.

- *Space-efficient closure representations* utilizes compile-time control and data flow information to optimize closure representations. By extensive closure sharing and allocating as many closures in registers as possible, the new *closure conversion* algorithm achieves very good asymptotic space usage, and improves the performance of the old compiler by about 14% on a DECstation 5000, *even without using a stack.* Further empirical and analytic studies show that the execution cost of stack-allocated and heap-allocated activation records is similar, but heap allocation is simpler to implement and allow very efficient first-class continuations.

- *Unrolling lists* takes advantage of the higher-level language abstraction in SML to support more efficient representations for lists. By representing each *cons* cell using multiple *car* fields and one *cdr* field, the *unrolled list* reduces the memory used for links and significantly shortens the length of control-dependence and data-dependence chains in operations on lists.

1.2 Outline of this dissertation

The development of this dissertation may be easier to follow for readers with some background in Standard ML [Ull93, Har86, MTH90] and basic compilation techniques for functional languages [App92, Pey87].

Chapter 2 contains a survey of various functional languages, an introduction to SML (which may be skipped if the reader is familiar with SML notation), and a review of several important aspects in compiling functional languages. This chapter also explains the experimental methodology and the SML benchmarks used in the rest of the chapters.

Chapters 3 through 6 constitute the core of this dissertation. Chapter 3 describes the design, implementation, and evaluation of the *type-directed compilation* technique in the context of the Standard ML of New Jersey compiler [AM91]; the new technique allows data objects in SML to use efficient unboxed representations, even with the presence of polymorphic functions. Chapter 4 describes the design, implementation, and evaluation of a new heap-based environment allocation scheme that uses the *space-efficient closure representations*; this new scheme supports very efficient function calls and returns, even though all activation records are allocated on the heap. Chapter 5 does an empirical and analytic study of the heap-based scheme described in Chapter 4 and the conventional stack-based scheme, and shows that in compiling languages such as SML, the efficient heap-based scheme can be more attractive than a stack-based scheme. Chapter 6 describes the *unrolling* *lists* technique that supports a more efficient representation for lists; the *unrolled list* reduces the memory used for links and significantly shortens the length of control-dependence and data-dependence chains in operations on lists.

Finally, Chapter 7 describes areas for future research and summarizes the results of this dissertation.

History

The idea of allocating continuation closures in callee-save registers (described in Chapter 4) is first published as Reference [AS92]. The new closure conversion algorithm in Chapter 4 is developed recently, and published as Reference [SA94]. The measurement data used in Reference [SA94] is different from one in Chapter 4 because they are using different versions of the SML/NJ compiler. The comparison of stack-based and heap-based closure allocation scheme described in Chapter 5 previously appeared as Reference [AS94]. The *unrolling lists* technique described in Chapter 6 is previously published as Reference [SRA94]. Some of the ideas used in Chapter 3 are evolved from my work on *smartest recompilation* [SA93, SA92], which is not described in this dissertation.

Chapter 2

Background

This chapter sets the stage for the presentations in Chapters 3 through 6. First, we review the fundamental concepts and notations evolved in the development of functional languages; then we present an introduction to Standard ML; finally, we explain the space-usage assumption and the experimental methodology used in this dissertation.

2.1 Evolution of functional languages

In this section, we use four representative functional languages—namely, lambda calculus, Lisp, ML, and Haskell—to explain the fundamental concepts and notations evolved in the development of functional languages. A more complete survey of the history of functional languages can be found in Hudak [Hud89]. This section only describes the core ML language and its polymorphic type system; a more detailed description of Standard ML [MTH90] is presented later in Section 2.2.

2.1.1 Lambda calculus

The development of functional languages has been influenced from time to time by many sources, but none is as fundamental as the work of Church [Chu41] on the *lambda calculus*. The lambda calculus is often regarded as the first functional language, and all modern functional languages can be thought of as nontrivial embellishments of the lambda calculus. Interested readers are referred to the excellent book by Barendregt [Bar84] for more detailed explanations.

The abstract syntax of the *pure untyped lambda calculus* (a name chosen to distinguish it from other versions defined later) embodies what are called *lambda expressions*, defined by the following grammar:

$$e ::= x \mid \lambda x.e_1 \mid e_1\ e_2$$

where x denotes an arbitrary identifier. Expressions of the form $\lambda x.e$ are called *abstractions* and of the form $e_1\ e_2$ are called *applications*. The former captures the notion of a function and the latter captures the notion of application of a function. By convention, application is assumed to be left associative, so that $(e_1\ e_2\ e_3)$ is the same as $((e_1\ e_2)\ e_3)$.

The rewrite rules of the lambda calculus depend on the notion of substitution of an expression e_1 for all free occurrences of an identifier x in an expression e_2, which we write as $[e_1/x]e_2$. To understand substitution, we must first understand the notion of the free variables of an expression e, which we write as $fv(e)$ and define by the following simple rules:

$$fv(x) = \{x\};$$
$$fv(e_1e_2) = fv(e_1) \cup fv(e_2);$$
$$fv(\lambda x.e) = fv(e) \setminus \{x\}.$$

We say that x is free in e if and only if $x \in fv(e)$. The substitution "$[e_1/x]e_2$" can then be inductively defined as follows:

$$[e/x]x = e;$$
$$[e/x]y = y;$$
$$[e_1/x](e_2e_3) = ([e_1/x]e_2)([e_1/x]e_3);$$
$$[e_1/x]\lambda x.e_2 = \lambda x.e_2;$$
$$[e_1/x]\lambda y.e_2 = \lambda y.[e_1/x]e_2, \text{ if } x \notin fv(e_2) \text{ or } y \notin fv(e_1);$$
$$[e_1/x]\lambda y.e_2 = \lambda z.[e_1/x]([z/y]e_2), \text{ otherwise, where } z \notin (fv(e_1) \cup fv(e_2)).$$

The last rule is the subtle one, since it is where a name conflict could occur and is resolved by making a name change.

To complete the lambda calculus, we define three simple *conversion* rules on lambda expression:

α-**conversion** (renaming): $\lambda x.e \Longleftrightarrow_\alpha \lambda y.[y/x]e$ where $y \notin fv(e)$;

β-**conversion** (application): $(\lambda x.e_1)e_2 \Longleftrightarrow_\beta [e_2/x]e_1$;

η-**conversion** : if $x \notin e$, then $\lambda x.ex \Longleftrightarrow_\eta e$.

The notion of *reduction*, which is the same as conversion but restricted so that β-conversion and η-conversion only happen in one direction:

β-reduction : $(\lambda x.e_1)e_2 \Longrightarrow_\beta [e_2/x]e_1$;

η-reduction if $x \notin e$, then $\lambda x.ex \Longrightarrow_\eta e$.

We write $e_1 \stackrel{*}{\Longrightarrow} e_2$ if e_2 can be derived from zero or more β- or η-reductions or α-conversions; and $e_1 \stackrel{*}{\Longleftrightarrow} e_2$ if e_2 can be derived from zero or more α-, β-, or η-conversions. In summary, $\stackrel{*}{\Longrightarrow}$ captures the notion of reducibility, and $\stackrel{*}{\Longleftrightarrow}$ captures the notion of intraconvertibility.

A lambda expression is in *normal form* if it cannot be further reduced using β- or η-reduction. For example, $\lambda x \lambda y.y$, $\lambda x.x$, xy are in normal forms, while $(\lambda x.y)z$ is not in normal form because it can be β-reduced into y. Note that some lambda expressions have no normal form, such as

$$(\lambda x.(xx))(\lambda x.(xx)),$$

where the only possible β-reduction leads to an identical term, and thus the reduction process is nonterminating.

The famous Church-Rosser theorem says that if e_1 and e_2 are intraconvertible (i.e., $e_1 \stackrel{*}{\Longleftrightarrow} e_2$), then there exists a third term (possibly the same as e_1 or e_2) to which they can both be reduced. One corollary of this is that a lambda expression e can never be reduced to two distinct normal forms. Otherwise, suppose $e \stackrel{*}{\Longrightarrow} e_1$ and $e \stackrel{*}{\Longrightarrow} e_2$, and both e_1 and e_2 are in normal form, then according to the Church-Rosser theorem, there exists a third term e' to which both e_1 and e_2 can be reduced. But e_1 and e_2 cannot be both in normal form. This corollary essentially means that as long as the reduction finally reaches a normal form, the way how it is carried out (i.e., the evaluation order) is irrelevant.

Two most well known ways to carry out the reduction are *normal-order reduction* and *applicative-order reduction*: the normal-order reduction, corresponding to the *call-by-name* evaluation strategy, is a sequential reduction in which, whenever there is more than one reducible sub-expression (called a *redex*), the leftmost one is chosen first; the applicative-order reduction, corresponding to the *call-by-value* evaluation strategy, is a sequential reduction in which the leftmost innermost redex is chosen first. Given an expression e, applying the normal-order reduction on e will always yield the normal form of e if there is one. Applicative-order reduction, however, is not always adequate. Consider the following example:

$$(\lambda x.y)((\lambda x.(xx))(\lambda x.(xx))) \Longrightarrow (\lambda x.y)((\lambda x.(xx))(\lambda x.(xx))) \Longrightarrow \ldots;$$

normal order reduction will yield:

$$(\lambda x.y)((\lambda x.(xx))(\lambda x.(xx))) \Longrightarrow y\ .$$

One nice thing about the lambda calculus is the ability to express recursive functions nonrecursively. Given a lambda expression e, there is a fixpoint e' such that $(ee') \overset{*}{\Longleftrightarrow} e'$; One example of e' is just the expression (Ye) where the Y combinator is defined by

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

With this Y combinator, every recursive function in the form "$f \equiv \ldots f \ldots$" can now be written nonrecursively as

$$f \equiv Y(\lambda f.\ldots f \ldots).$$

For example, the factorial function (written in lambda calculus extended with conditional expressions and constants)

$$fac \equiv \lambda n.if(n = 0)\ then\ 1\ else\ (n * fac(n-1))$$

can be written nonrecursively as

$$fac \equiv Y(\lambda fac.\lambda n.if(n = 0)\ then\ 1\ else\ (n * fac(n-1))).$$

The ability of the lambda calculus to simulate recursion in this way is the key to its power and accounts for its persistence as a useful model of computation. Actually, it is shown (by Turing [Tur37]) that those functions computable on Turing machines are exactly those definable functions in the lambda calculus.

2.1.2 Lisp

Lisp [McC60, Ste84] was the first functional language in the world, developed by John McCarthy in 1950s. Although the core of Lisp is essentially the lambda calculus plus the constants, many new features in Lisp are now commonly used by almost all functional languages.

For example, Lisp is the first language that uses S-expressions. An S-expression is either a *symbol*, a *number*, or a *pair* of S-expressions. A restricted form of S-expression is the list data structure. Most list notations, such as the primitive *cons*, *car*, and *cdr* operations, are first introduced in Lisp.

Unlike the lambda calculus, Lisp is a strict call-by-value language. It allows side effect on S-expressions, using the primitive *rplaca* (for "replace the car") and *rplacd* (for "replace the cdr") operations.

Lisp is the first language that supports dynamic storage allocation. Whenever *cons* is applied, the *cons* cell must be allocated on the heap. When memory is used up, garbage collection is triggered to reclaim unused cells.

Lisp programs are untyped; that is, Lisp does not associate types with expressions. Type-checking in Lisp is done during program execution. This so-called "dynamic type-checking" is done by inserting extra code into the program to watch for impending errors. Because of this, every data object in Lisp must have a type tag attached.

2.1.3 ML

A fundamental difference between ML and Lisp is that ML is statically typed and Lisp is dynamically typed. The type checking in ML is done, once and for all, at compile time. Like Lisp, ML is a "mostly functional" language, meaning that most ML programs are side effect free but ML has facilities for creating and manipulating mutable objects. The evaluation order in ML is *call-by-value*, with function arguments strictly evaluated from left to right.

ML is most well-known for its powerful polymorphic type system. ML programs are checked for type correctness at compile time. The compiler can infer the types of identifiers during type checking, so that the programmer need not declare them explicitly. In the following, we briefly explain the ML type system using the following core ML language (which still looks like the lambda calculus):

$$e ::= x \mid \lambda x.e_1 \mid e_1\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

Here, the `let` statement is just a syntactic language construct equivalent to $[e_1/x]e_2$; this additional structure is used to support ML's powerful polymorphic type system, developed independently by Hindley [Hin69] and Milner [Mil78].

Suppose TyVar is an infinite set of type variables and TyCon is a set of nullary type constructors,

$$\pi \in \text{TyCon} = \{int, bool, \ldots\}$$

$$\alpha \in \text{TyVar} = \{\beta, \gamma, \alpha_1, \ldots\}$$

then the set of *types*, Type, ranged over by τ and the set of *type schemes*, TypeScheme, ranged over by σ are defined by

$$\tau ::= \pi \mid \alpha \mid \tau_1 \rightarrow \tau_2$$

$$\sigma ::= \tau \mid \forall\alpha.\sigma_1.$$

A *type environment* is a finite map from program variables to type schemes. $tyvars(\tau)$, $tyvars(\sigma)$ and $tyvars(TE)$ are the set of type variables that occur *free* in τ, σ and TE respectively. A type τ' is a *generic instance* of a type scheme $\sigma = \forall\alpha_1, ..., \alpha_n.\tau$, written as $\tau' \prec \sigma$, if there exists a substitution S with its domain being a subset of $\{\alpha_1, ..., \alpha_n\}$

$$\text{(VAR)} \qquad \frac{\tau \prec TE(x)}{TE \vdash x : \tau}$$

$$\text{(ABS)} \qquad \frac{TE \pm \{x \mapsto \tau'\} \vdash e : \tau}{TE \vdash \lambda x.e : \tau' \to \tau}$$

$$\text{(APP)} \qquad \frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1 e_2 : \tau}$$

$$\text{(LET)} \qquad \frac{TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto gen(TE, \tau_1)\} \vdash e_2 : \tau}{TE \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau}$$

Figure 1: The type deduction rules for core ML

and $\tau' = S(\tau)$. A type scheme σ_1 is more general than σ_2, denoted as $\sigma_2 \prec \sigma_1$, if all generic instances of σ_2 are also generic instances of σ_1. The generalization of a type τ in a type environment TE is denoted by $gen(TE, \tau)$, it is the type scheme $\forall\alpha_1, ..., \alpha_n.\tau$ where $\{\alpha_1, ..., \alpha_n\} = tyvars(\tau) \backslash tyvars(TE)$. The core ML type system, in the form of type deduction rules as $TE \vdash e : \tau$, is listed in Figure 1.

In general, an expression e can be given many different typings under the deduction rules and a given type environment TE. We are particularly concerned with the *principal type* of e under TE, namely the type τ such that if $TE \vdash e : \tau$ and $TE \vdash e : \tau'$, then $gen(TE, \tau) \prec \tau'$. Damas and Milner [DM82] have shown that any expression that has a type in a given environment has a principal type in that environment, which is unique except for choice of bound type-variable names and can be inferred using the well-known type assignment algorithm "W" [DM82].

Standard ML [MTH90] is just core ML extended with constants, pattern matching, data type definitions, references and exceptions, and a sophisticated module system [Mac84]. We'll explain the details of Standard ML in Section 2.2.

2.1.4 Haskell

Haskell is a nonstrict, purely functional programming language developed by Hudak, Peyton Jones, Wadler *et al* [HJ*et al*92]. Core Haskell is very much like the core ML described in the last section; it is a statically typed language with the same Hindley-Milner type system. The main differences between ML and Haskell are as follows:

- ML is a mostly functional language; side effects in ML are allowed. Haskell is a purely functional language; no side effects are permitted.
- ML is a "strict" call-by-value language (i.e., uses applicative-order reduction). Haskell, on the other hand, is a "non-strict" language that uses *lazy evaluation.*
- ML supports a more expressive and powerful module system than Haskell. For example, ML supports parametrized modules (i.e., functors) and module abstractions while Haskell does not.
- The type system in Haskell also supports *type-classes*—a more general form of overloading [WB89].

The details on how to efficiently compile pure functional languages such as Haskell are discussed by Peyton Jones [Pey87, Pey92]. Although some of the techniques described in this dissertation also apply to lazy languages such as Haskell, the rest of dissertation will mainly focus on compiling strict functional languages such as Standard ML.

2.2 Introduction to Standard ML

This section provides an introduction to Standard ML (SML) that should allow the reader to follow the examples and notations used in this dissertation easily. Readers are referred to References [Har86, Ull93, Pau91] for a more complete introduction. The formal definition and commentary for SML can be found in References [MTH90, MT91].

2.2.1 Basic expressions, values, and types

SML is an expression language: the traditional statement constructs, such as blocks, conditionals, case statements, and assignment, are packaged as expressions. Every expression has a statically determined type and will only evaluate to values of that type. In the following, we illustrate the basic values and types of SML by example:

Unit The type `unit` consists of a single value, written (). This type is used whenever an expression has no interesting value, or a function is to have no arguments.

Booleans The type `bool` consists of the values `true` and `false`. The ordinary boolean negation is available as `not`. Booleans are most commonly used as the first argument of the conditional expression

$$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2.$$

In SML, both the `then` and `else` clauses must have the same type.

Integers The type `int` is the set of integers. Integers are written in the usual way, except that negative integers are written with the tilde character "~" rather than a minus sign. ML supports the usual arithmetic operators, `+`, `-`, `*`, `div`, and `mod`; and the usual relational operators, `<`, `<=`, `>`, `>=`, `=`, and `<>`.

Strings The type `string` consists of the set of finite sequences of characters. Strings are written in the conventional fashion as characters between double quotes, e.g., `"hello world!\n"`, `"foo"`.

Real Numbers The type of floating point numbers is known in SML as `real`. Real numbers written in more or less the usual fashion for programming languages, e.g., `3.1415`, `0.03`.

Tuples The type $\tau_1 * \tau_2$, where τ_1 and τ_2 are types, is the type of ordered pairs whose first component has type τ_1 and whose second component has type τ. Ordered pairs are written (e_1, e_2), where e_1 and e_2 are expressions. An ordered *n-tuple*, where $n \geq 2$, is written as n comma-separated expressions between parentheses. For example, `(1,true)` is a pair of `int` and `bool`; `("bar", 17, 3.14*3.14)` is a triple whose SML type is `string * int * real`.

Records The *record type* is quite similar to Pascal records and to C structures. A record consists of a finite set of labelled fields, each with a value of any type (as with tuples, different fields may have different types). Record values are written by giving a set of equations of the form $l = e$, where l is the label and e is an expression, enclosed in curly braces. The type of a record is a set of pairs of the form $l : \tau$ where l is a label and τ is a type, aslo enclosed in curly braces. For example, `{name="bar", used=true}` is a record whose type is `{name:string, used:bool}`.

2.2.2 Value bindings, functions, and polymorphism

The principal mechanism for associating values with variables in SML is to use value declarations or function declarations. A *value binding* starts with a leading key word `val`, e.g.,

```
val x = 4 * 5
```

it is evaluated by first evaluating the expression on the right-hand side, and then setting the value of the variable on the left-hand side to this value. In the above example, `x` is bound to `20`, an integer.

Function declaration uses the leading key word `fun`; for example, the factorial function can be defined as:

```
fun fac n = if (n = 0) then 1 else n * fac(n-1)
```

The type of function `fac` is `int` $\rightarrow$ `int`.

Functions in SML are first-class values; they can be passed as arguments, embedded in data structures and returned as results. For example, in the following,

```
fun compose (f,g) = fn x => (f (g x))
```

the function `compose` takes two functions `f` and `g` as arguments; it returns the composition of `f` and `g` as the result.

As described in Section 2.1.3, SML also supports polymorphic types. The `compose` function above has a polymorphic type

$$\forall\alpha\forall\beta\forall\gamma.((\alpha \rightarrow \beta) * (\beta \rightarrow \gamma)) \rightarrow (\alpha \rightarrow \gamma);$$

it can be applied to any pair of functions as long as the application satisfies the ML type deduction rules (see Figure 1).

2.2.3 Datatypes and pattern matching

One very important feature in SML is that programmer can declare new concrete data types using the `datatype` constructs. The most commonly used data type is *list*, defined as

```
datatype 'a list = nil
                 | :: of 'a * 'a list
```

here `'a` is a type variable used to denote the Greek symbol α. This data type declaration defines a list to be either empty (`nil`), or the cons of an element and a list (`::`). In SML, the cons operator "::" is pre-declared as an infix operator; lists are often represented syntactically as

$$[e_1, e_2, \ldots, ..., e_n],$$

which really denotes

$$(e_1 :: (e_2 :: \ldots :: (e_n :: \texttt{nil}))).$$

Another important predefined data type in SML (actually, only in SML/NJ [AM91]) is the polymorphic option type:

```
datatype 'a option = NONE | SOME of 'a
```

Structured data type values are decomposed using a powerful *pattern matching* notation. A *pattern* is a data template. If a datum *matches* a pattern, the variables in the pattern are bound to the corresponding components of the datum. A match *fails* if the datum and pattern do not concur. Pattern matching is useful for defining functions—a series of n patterns can select from a function's n cases. The | symbol separates pattern-case pairs. Matching proceeds serially from left to right. A successful match causes evaluation of the corresponding function case. The matching process faults if no pattern matches the function's argument value. Using patterns, a function to compute the lengths of lists is:

```
fun length nil = 0
  | length (x::xs) = 1 + (length xs)
```

When `length` is applied to the empty list, the first pattern (`nil`) matches and `length` returns 0; otherwise, `length`'s argument matches the cons cell (with the `::` list constructor) in the second pattern (`x::xs`). This match binds `x` to the head element of the list and `xs` to the tail. Since the length function does not require a binding for a list's head element, the pattern `(x::xs)` is more informatively written as `(_::xs)` where the *wildcard* (`_`) matches—but does not bind—anything.

2.2.4 Reference values

Although SML is mostly functional, it does allow side effects. References are cells whose contents may be changed after creation by assignment. The `ref` "datatype" constructor, and its corresponding value constructor, are almost as if defined by the declaration

```
datatype 'a ref = ref of 'a
```

A reference whose initial contents are string `"foo"` may be created and later altered as follows:

```
let val r = ref "foo"
 in (r := ("bar"^(!r))
end
```

Here, "`^`" is the string concatenation operator (infix); "`:=`" is the assignment operator (infix); "!" is the dereferencing operator. The final content for `r` is the string `"barfoo"`.

2.2.5 Exceptions

SML has an exception mechanism for signaling run-time errors and other exceptional conditions. For example,

```
exception Head

fun head(nil) = raise Head
  | head(x::xs) = x

fun head2 l = head(l) handle Head => 0
```

the first line is an *exception binding* that declares `Head` to be an exception. The function `head` is defined in the usual waay by pattern matching on the constructors of the `list` type. In the case of a non-empty list, the value of `head` is simply the first element. But for `nil`, the function `head` is unable to return a value, and instead *raises* an exception. To be complete there is a way of doing something about an error; in SML, this is called an *exception handler*. The expression

$$e \texttt{ handle } exn \texttt{ => } e'$$

is evaluated as follows: first, evaluate e; if it returns a value v, then the value of the whole expression is v; if it raises the exception exn, then returns the value of e'; if it raises any other exception, then raise that exception. The above function `head2` will return 0 if the argument is an empty list.

2.2.6 First-class continuations

An extension in Standard ML of New Jersey (SML/NJ) [AM91] is the typed first-class continuation [DHM91], defined as follows:

```
type 'a cont
val callcc : ('1a cont -> '1a) -> '1a
val throw : 'a cont -> 'a -> 'b
```

Here, `cont` is an abstract type constructor; "`'1a`" denotes a weak type variable (also used for typing references) [Mac88]; `callcc` is used to capture the current state (representing the "rest of the program"), just like the *call-with-current-continuation* function in Scheme [RC86]; the captured continuation can be later applied by using the `throw` function. First-class continuations can be used to support tasking, coroutines, exceptions, and so on.

2.2.7 Modules

SML provides a powerful module system, which can be used to partition programs along clean interfaces.

In its simplest form, a module is (syntactically) just a collection of declarations viewed as a unit, or (semantically) the environment defined by those definitions. This is one form of a *structure expression*: "`struct` *dec* `end`." For example, the following structure expression represents an implementation of stacks:

```
struct
  datatype 'a stack = Empty | Push of 'a * 'a stack
  exception Pop and Top
  fun empty(Empty) = true | empty _ = false
  val push = Push
  fun pop(Push(v,s)) = s | pop(Empty) = raise Pop
  fun top(Push(v,s)) = v | top(Empty) = raise Top
end
```

Structure expressions and ordinary expressions are distinct classes; structure expressions may be bound using the `structure` keyword to structure identifiers. For example, we might make a structure `Stack` using the structure expression shown above:

```
structure Stack = struct
                    datatype 'a stack = ...
                    ....
                  end
```

It is often useful to explicitly constrain a structure binding to limit the visibility of its fields. This is done with a *signature*, which is to structure binding as a type constraint is to a value binding. For example, we might write a signature for the `Stack` as

```
sig type 'a stack
    exception Pop and Top
    val Empty : 'a stack
    val push : 'a * 'a stack -> 'a stack
    val empty : 'a stack -> bool
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
end
```

The signature mentions the structure components that will be visible outside the structure. Signatures may be bound to identifiers by a signature declaration using the key word `signature`, for example, the signature above could be bound to the identifier `STACK` by the declaration

```
signature STACK = sig type 'a stack
                      ..........
                  end
```

A signature can be used to constrain a structure (also called *signature matching*) by including it in a structure declaration:

```
structure Stack1 : STACK = Stack
```

Now the constructor `Push` is not a visible component of the `Stack1` structure, since it does not appear in the signature. Since the constructor `Empty` is mentioned as a `val` in the signature, but not as a constructor (i.e., as part of a data type specification), then `Stack1.Empty` may be applied as a function but not matched in a pattern.

SML also supports parametrized modules, called *functors*. Functors, which are functions on structures, are used to manage the dynamics of program development in SML. Functors are defined using *functor declarations*, using the key word `functor`. The syntax of a functor declaration is similar to the clausal form of function definition, for example,

```
functor F(S : STACK) =
  struct fun init() = S.Empty
         fun pushlist(nil,s) = s
           | pushlist(x::xs,s) = S.push(x,pushlist(xs,s))
         val pop = S.pop
         val top = S.top
  end
```

this functor `F` defines a function that, given any structure matching `STACK`, returns another structure, which contains four components, `init`, `pushlist`, `pop`, and `top`.

2.2.8 Summary

In summary, SML is a functional language that contains many features frequently seen in the modern higher-level programming languages:

- SML is safe: programs cannot corrupt the runtime system so that further execution of the program is not faithful to the language semantics.
- SML supports first-class functions: functions can be passed as arguments, returned as results, and stored in variables. The principal control mechanism in SML is recursive function application.
- SML is statically typed with a powerful polymorphic type system. Every legal expression in SML has a uniquely determined most general typing which is determined automatically by the compiler.
- SML has a module system supporting abstract data types, hiding of representations, and type-checked interfaces.
- SML is a *mostly* functional language. Although SML programs are mostly written using immutable data structures only, side effects are still allowed.

2.3 Compiling functional languages

As mentioned in Chapter 1, the major challenges in compiling functional languages are on how to compile function call and return efficiently and how to optimize the runtime data representations. On many other aspects, compiling functional languages can be attacked using the same techniques described in the "Dragon" book [ASU86].

This section reviews the basic issues involved in compiling function call and return for functional languages. The background about efficient data representations is discussed later in Section 3.2 and Section 6.1.

2.3.1 Stack allocation and heap allocation

Stack allocation is the most commonly used method to implement function call and return. In the stack scheme, a contiguous region of memory is used as a runtime stack. An *activation record* is pushed onto the stack at each function call and then popped off the stack when the function returns. Each activation record normally contains formal parameters, returned values, access link, saved registers, temporaries, and local data. The access link is a pointer from the activation record of a function to the activation record of its enclosing function; this can be decided at compile time. By following the access links, each function can access non-local variables stored in the activation records of outer functions. Another region of memory is for static data and code segment; this area is of fixed size (for each program) and is used to store code segment and global variables. A third region is the "heap;" all the dynamically allocated data are stored here. In languages such as C and Pascal, programmers are responsible for freeing the dead data objects by explicitly calling the "`free`" or "`dispose`" library function in their programs. In functional languages, memory allocation and deallocation are often done implicitly; dead data objects are reclaimed by a garbage collector.

Function call and return can be implemented using either the stack-based scheme or the heap-based scheme. The stack-based scheme pushes an activation record on the stack at each function call and pops the activation record off the stack when the function returns. The heap-based scheme allocates an activation record in the heap at each function call and does nothing at the function return; the activation record is left on the heap, and later reclaimed by the garbage collector.

```
fun f(v,w,x,y,z) =
  let fun g() =
        let val u = hd(v)
            fun h() =
              let fun i() = w+x+y+z+3
               in (i,u)
              end
         in h
        end
   in g
  end

fun big(n) = if n<1 then [0] else n :: big(n-1)

fun loop (n,res) =
  if n<1 then res
  else (let val s = f(big(N),0,0,0,0)()
         in loop(n-1,s::res)
        end)

val result = loop(N,[])
```

Figure 2: Higher-order functions and space safety

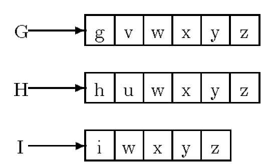
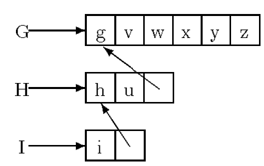
2.3.2 Higher-order functions and closures

Because functions in functional languages are first-class values that may be passed as arguments, returned as results, and put into data structures, implementing function call and return is more complicated than in imperative languages such as C and Pascal.

For example, Figure 2 presents an example in SML that uses higher-order functions. The function f returns as its result a nested function g, which returns a nested function h, which returns a nested function i and a value u computed by selecting the head of the list v. The function $big(n)$ makes a list of length n, and *loop* makes a list of n copies of function h, that is, the result of the function application $f(big(N), 0, 0, 0, 0)()$.

Higher-order functions such as g, h, i cannot be implemented just using a LIFO stack because they may be called later even though their environments (i.e., activation records) have been popped off the stack. The usual solution to this problem is to represent functions as *closures* [Lan64]. A function with free variables is said to be *open*; a *closure* is a data structure containing both the machine code address of an open function, and bindings for all the non-local variables (i.e., free variables) of that function. The machine-code implementation of the function knows to find the values of non-local variables in the closure data structure.

Table 1: Flat closures and linked closures

Flat Closures	**Linked Closures**
	

For example, Table 1 shows two commonly used closure representations: *flat closure* and *linked closure*. A *flat closure* [Car84b] is a record that holds only the free variables needed by the function. For example, the flat closure for h (denoted as H) contains just the code pointer (denoted by h) plus the values for variable u, w, x, y, and z. A *linked closure* [Lan64] is a record that contains the bound variables of the enclosing function, together with a pointer to the enclosing function's closure. For example, the linked closure for h contains the code pointer h, the locally bound variable u, and the pointer to the enclosing function g's closure.

2.3.3 Space safety

In any language, it is common for the programmer to have variables in scope that are "dead;" that is, their current values will never again be needed. In a garbage-collected language, the garbage collector need not use such variables as "roots" of live data. Several implementors have independently discovered that this is really important: if the collector traverses too many dead variables, the memory use of the program can increase by a large factor [Bak76, Cha88, RW93, App92, Jon92].

In fact, a collector that starts from only the (statically determinable) live variables can often keep *asymptotically less* data live than a less careful collector; that is, one system might use $O(N)$ space where another uses $O(N^2)$ space, where N is the size of the input. This theorem, examples, and a description of compiler techniques that are "safe for space complexity" are described by Appel [App92, Chapter 12].

We can use the example shown in Figure 2 to illustrate this problem. With flat closures, each evaluation of $f(\ldots)()$ yields a closure s for h that contains just a few integers u, w, x,

y, and z; the final result (i.e., *result*) contains N copies of the closure s for h, thus it uses at most $O(N)$ space.

With linked closures, each closure s for h contains a pointer to the closure for g, which contains a list v of size N. Since the final *result* keeps N closures for different instantiations of h simultaneously—each with a different (large) value for the variable v—it requires $O(N^2)$ space consumption instead of $O(N)$. This space leak is caused by inappropriately retaining some "dead" objects (v) that should be garbage collected earlier.

Such space leaks are unacceptable. Closure (and frame) representations must not cause space leaks (see Chapter 4). In 1992, we found several instances of real programs whose live data size (and therefore memory use) was unnecessarily large (with factors of 2 to 80) when compiled by early versions of our compiler that introduced this kind of space leak. All recent versions of SML/NJ have obeyed the "safe for space complexity" (SSC) rule, and users really did notice the improvement. The SSC rule is stated as follows: *any local variable binding must be unreachable after its last use within its scope* (see Appel [App92] for a more formal definition).

Assumption: all discussions in this dissertation are based on the assumption that the compiler must satisfy the "safe for space complexity" rule.

2.4 Experimental measurements

All measurements in this dissertation are done on a DEC5000/240 workstation with 128 mega-bytes of memory (under Ultrix 4.3). Table 2 shows the set of benchmarks we use; for each benchmark, we also show the source program size (in number of lines) and the degree of modularity (in number of modules and files[1]). In our measurements, a file is a basic separate compilation unit. Among these 12 benchmarks, **MBrot**, **Nucleic**, **Simple**, **Ray**, and **BHut** involves intensive floating-point operations; **Sieve** and **KB-Comp** frequently use first-class continuations or exceptions; **VLIW** and **KB-Comp** make heavy use of higher-order functions.

The execution time is measured using the standard UNIX timer facility (using the interface provided by the SML/NJ compiler). Each benchmark is run 20 times by a specific compiler. Since the performance of different runs are rather different,[2] we pick the fastest

[1] In all the measurements, we treat each file (not module) as a separate compilation unit.

[2] According to our experience, the difference between the fastest run and the slowest run of the same executable can be as much as a factor of two, depending on where and how the executable is put into the main memory.

Table 2: General information about the benchmark programs

Program	Lines	Modules	Files	Description
BHut	1258	9	9	"Barnes-Hut" N-body problem solver [BH86], translated from C into Standard ML by John Reppy.
Boyer	919	4	3	Standard theorem-prover benchmark [BM72], translated from the Gabriel benchmark [Gab85].
Sieve	1356	4	5	CML implementation of prime number generator written by John Reppy [Rep91].
KB-Comp	655	1	1	An implementation of the Knuth–Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.
Lexgen	1185	5	1	A lexical-analyzer generator, implemented by James S. Mattson and David R. Tarditi [AMT89], processing the lexical description of Standard ML.
Life	148	1	1	The game of Life, written by Chris Reade and described in his book [Rea89], running 50 generations of a glider gun.
Ray	874	5	5	A simple ray tracer written in C by Don Mitchell, translated into Standard ML by John Reppy.
Simple	990	2	1	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [CHR78], translated into ID [EA87], and then translated into Standard ML by Lal George.
VLIW	3658	24	2	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
YACC	7432	56	26	A LALR(1) parser generator, implemented by David R. Tarditi [TA90], processing the grammar of Standard ML.
MBrot	60	1	1	The Mandelbrot curve construction written by John Reppy.
Nucleic	3307	1	1	The pseudoknot program that computes the three-dimensional structure of part of a nucleic acid molecule [FTL94], translated from Scheme into Standard ML by Peter Lee.

run as its final performance data (as recommended, for example, by the SPEC benchmark consortium [Sta89]).

All compilers mentioned in this dissertation use a simple two-generation copying garbage collector [App89]. Available memory is divided into two half-spaces, and allocation occurs at the low end of the upper space (called *new space*). When the new space becomes full, all live data in the new space is appened into the other, *old space*, but it typically does not fill up the old space. The allocator then reset to begin filling the new space again. After some number of these *minor collections*, the old space occupies half the total available memory, though much of its data will no longer be live.

Garbage collection time can be very much dependent on heap size. In our measurements, for each benchmark, every version of the compiler is run in the same amount of memory (not the same ratio) so that improvements in space usage become improvements in garbage collection time. The amount of memory used for each benchmark is based on a ratio of 5 for the "base case" version.

Chapter 3

Type-Directed Compilation

Compile-time type information should be valuable in efficient compilation of statically typed functional languages such as Standard ML. But how should type-directed compilation work in real compilers, and how much performance gain will type-based optimizations yield? In order to support more efficient data representations and gain more experience about type-directed compilation, we have implemented a new type-based middle end and back end for the Standard ML of New Jersey compiler. This chapter describes the basic design of the new compiler, identify a number of practical issues, and then compare the performance of our new compiler with the old non-type-based compiler. Our measurement shows that a combination of several simple type-based optimizations reduces heap allocation by 36%; and improves the already-efficient code generated by the old non-type-based compiler by about 19% on a DECstation 5000.

3.1 Introduction

Compilers for languages with run-time type checking, such as Lisp and Smalltalk, must often use compilation strategies that are oblivious to the actual types of program variables, simply because no type information is available at compile time. For statically typed languages such as Standard ML (SML) [MTH90], there is sufficient type information at compile time to guarantee that primitive operators will never be applied to values of the wrong type. But, because of SML's parametric polymorphism, there are still contexts in which the types of (polymorphic) variables are not completely known. In such cases, the program can still manipulate values without inspecting their internal representation. But to manipulate them (pass them as arguments, store them in data structures, etc.), it is necessary to know their size. The usual solution is to discard all the static type information and adopt the approach used for dynamically typed languages, that is, to represent all program variables using a *standard boxed representation.* This means that every variable, every function closure, and every argument to a function, is represented in exactly one word. If the natural representation of a value does not fit into one word (as with a floating-point number, etc.), a pointer to a heap-allocated object is used instead. This is a source of great inefficiency.

Leroy [Ler92] has recently presented a *representation analysis* technique (for core-ML) that does not always require variables be boxed in one word. In his scheme, data objects whose types are not polymorphic can be represented in multiple words or in machine registers; only those variables that have polymorphic types must use boxed representations. When polymorphic functions are applied to monomorphic values, the compiler automatically inserts appropriate coercions (if necessary) to convert polymorphic functions from one representation to another. For example, in the following SML code:

```
fun quad (f, x) = (f(f(f(f(x)))))

fun h x =  x * x * 0.50 + x * 0.87 + 1.3

val res = h(3.14) + h(3.84) + quad(h , 1.05)
```

quad is a polymorphic function with type $\forall\alpha.(((\alpha \rightarrow \alpha) * \alpha) \rightarrow \alpha)$; all of its four calls to f must use the standard calling convention—passing its argument in a general-purpose register. On the other hand, h is a monomorphic function with type $real \rightarrow real$, so all monomorphic applications of h (e.g., in $h(3.14)$ and $h(3.84)$) can use a more efficient calling convention—passing its argument x in a floating-point register. When h is being passed to the polymorphic function *quad* (e.g., in $quad(h, 1.05)$), h has to be wrapped to use the standard calling convention so that f will be called correctly inside *quad*.

Representation analysis makes possible many interesting type-based compiler optimizations. But, since no existing compiler has fully implemented representation analysis for the complete SML language, many practical implementation issues are still unclear. For example, while Leroy [Ler92] has shown in detail how to insert coercions for core-ML, he does not address the issues in the ML module system, that is, how to insert coercions for functor application and signature matching (see Section 2.2.7). Propagating type information into the middle end and back end of the compiler can also incur large compilation overhead if it is not done carefully, because all the intermediate optimizations must preserve the type consistencies. In order to answer these questions and to gain more experience with type-directed compilation, we have implemented a new type-based middle end and back end for the Standard ML of New Jersey compiler (SML/NJ) [AM91]. In this chapter, we describe the basic design of our new compiler, identify and solve a number of practical problems involved in the implementation, and then present a detailed performance evaluation of various type-based compilation techniques. The major contributions of this chapter are as follows:

- Our new compiler is the first type-based compiler for the entire Standard ML language.
- We extend Leroy's representation analysis to the SML module language to support module-level abstractions and functor applications.
- We improve compilation speed and code size by using partial types at module boundaries, and by memo-izing coercions.
- We evaluate the utility of *minimum typing derivations* [Bjo94]—a method for eliminating unnecessary "wrapper" functions introduced by representation analysis.
- We show how the type annotations can be simplified in successive phases of the compiler, and how representation analysis can interact with the *Continuation-Passing Style* used by the SML/NJ compiler's optimizer.
- We compare representation analysis with the crude (but effective) known-function parameter specialization implemented by Kranz [Kra87].
- Our measurements show that a combination of several type-based optimizations reduces heap allocation by 36%, and improves the already-efficient code generated by the old non-type-based compiler by about 19%.

3.2 Data representations

The most important benefit of type-directed compilation is to allow data objects with specialized types to use more efficient data representations. In this section, we explain in detail what the *standard boxed representations* are, and what other more efficient alternatives one can use in type-based compilers.

Non-type-based compilers for polymorphic languages, such as the old SML/NJ compiler [AM91], must use the standard boxed representations for all data objects. More specifically, primitive types such as integers and reals are always tagged or boxed; every argument and result of a function, and every field of a closure or a record, must be either a tagged integer or a pointer to other objects that use the standard boxed representations. For example, in Figure 3, the value x is a four element record containing both real numbers and strings, each field of x must be boxed separately before being put into the top-level

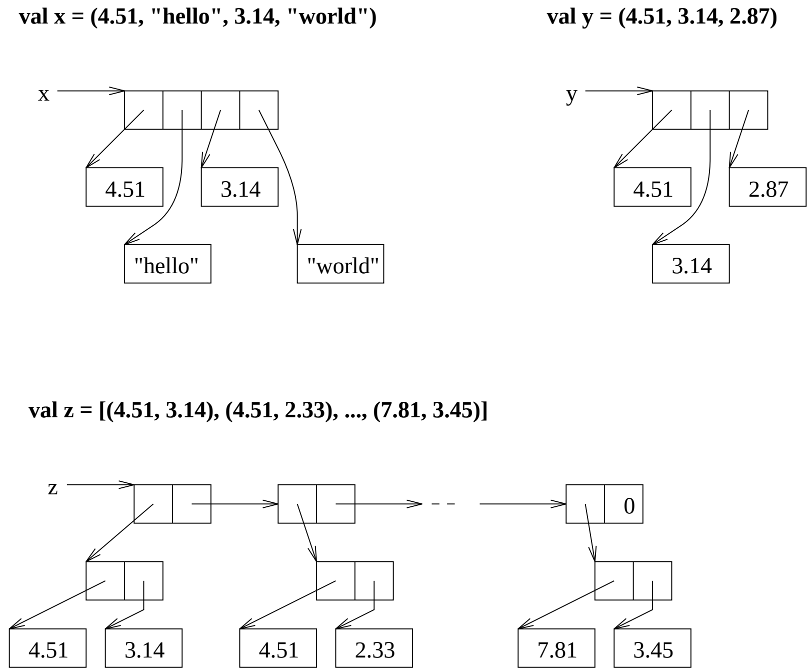


Figure 3: Standard boxed representations

record. Similarly, y is record but containing only real numbers, but each field still has to be separately boxed under standard boxed representations. For concrete data types such as variable z in Figure 3, each element of z must again be boxed using standard boxed representations.

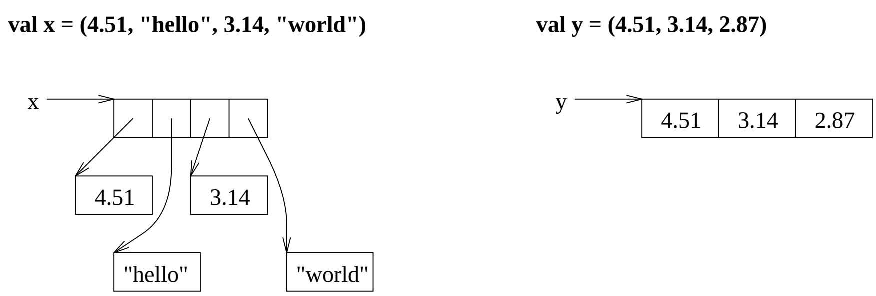
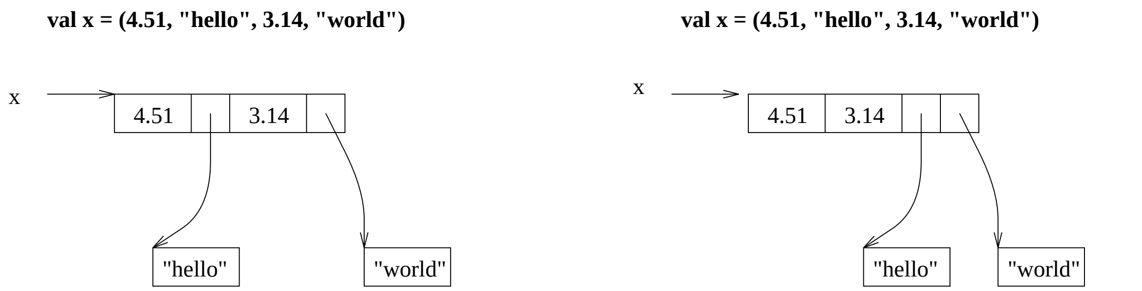


Figure 4: Flat unboxed representations with simple descriptors

In a type-based compiler, because we know the types of program variables, we can use much more efficient data representations, depending on how complicated a descriptor we want to support at runtime[1]. For example, in the SML/NJ compiler [App90, Rep93, AM91], the descriptor for runtime objects is just a kind tag plus the length of the object; each object may contain tagged/boxed objects, or untagged/unboxed objects, but not both. Under this scheme, records that contain only real numbers can be represented as real vectors (e.g., y in Figure 4); records that contain both unboxed values (i.e., real numbers) and boxed values (i.e., pointer to a string, etc.) must be still represented using two layers, with an extra box around each real number (e.g., x in Figure 4).



(a) naive way with bitmap descriptor (b) reordering fields, with more compact descriptor

Figure 5: Flat unboxed representations with sophisticated descriptors

If more sophisticated object descriptors are used, x can be represented even more efficiently, as shown in Figure 5. A naive way is to use a bitmap descriptor to indicate the boxity of each field, then the unboxed real numbers can just be mixed arbitrarily with other pointers (see Figure 5a); the problem with this is that it is quite expensive for the garbage collector to interpret the descriptor. Because we know the type of each element statically at compile time, a better way is to reorder all the fields to put all unboxed fields ahead of boxed fields (see Figure 5b); the descriptor for this kind of object is just two integers: one indicating the length of the unboxed part, another indicating the length of the boxed part.

For concrete data types such as the list z in Figure 3, more efficient data representations are also possible. If we know z has type $(real*real)\ list$, z can be represented more compactly as shown in Figure 6a or Figure 6b. The major problem with these is that when z is passed to a polymorphic function such as the *unzip* function:

```
fun unzip l = let fun h((a,b)::r,u,w) = h(r,a::u,b::w)
                    | h([],u,w) = (rev u, rev w)
               in h(l,[],[])
              end
```

[1]For statically typed languages such as SML, the descriptors kept at runtime are only used by the garbage collector to trace pointer data structures.

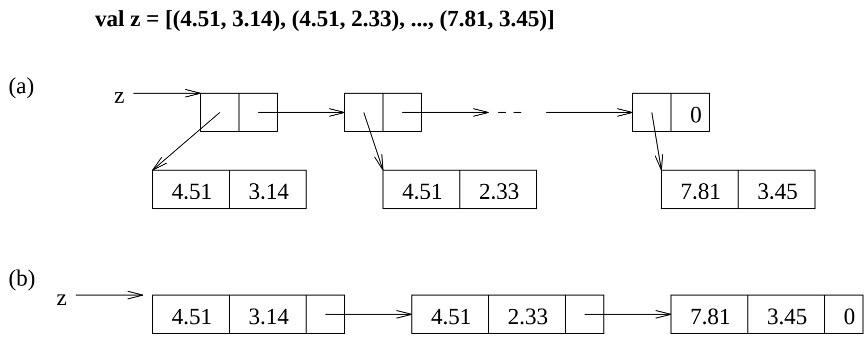


Figure 6: More compact representations for concrete data type such as list

the list z needs to be coerced from the more efficient representations (shown in Figure 6a and 6b) into the standard boxed representation (shown in Figure 3). These coercions can be very expensive because their costs are proportional to the length of the list. There are two solutions to solve this problem:

- The first approach—proposed by Leroy [Ler92]—is to use standard boxed representations for concrete data type objects. In other words, even though we know z has type (*real* $*$ *real*) *list*, we still represent z in the way shown in Figure 3, with each *car* cell pointing to an object that uses standard boxed representation. For example, if pairs such as `(4.51,3.14)` are normally represented as flat real vectors, when they are being added to (or fetched from) a list, they must be coerced from flat representations to (or from) standard boxed representations. The type-based compiler described in this paper also uses this approach. The LEXP language described later in Section 3.5.1 has special type called `RBOXEDty` to express such requirements.

- Another approach, described by Harper and Morrisett [HM95], is to represent concrete data types using more efficient representations as shown in Figure 6. When z is being passed to *unzip*, we also pass a type descriptor to *unzip* indicating how to extract each *car* field of z. How well this approach behaves in practice is still not clear (see more discussion in Section 3.8).

3.3 Overview of the compiler

This section gives an overview of our new type-based compiler. The overall organization of the new compiler is very similar to the old Standard ML of New Jersey compiler described by Appel and MacQueen [AM91]. Compilation of an SML program is grossly divided into

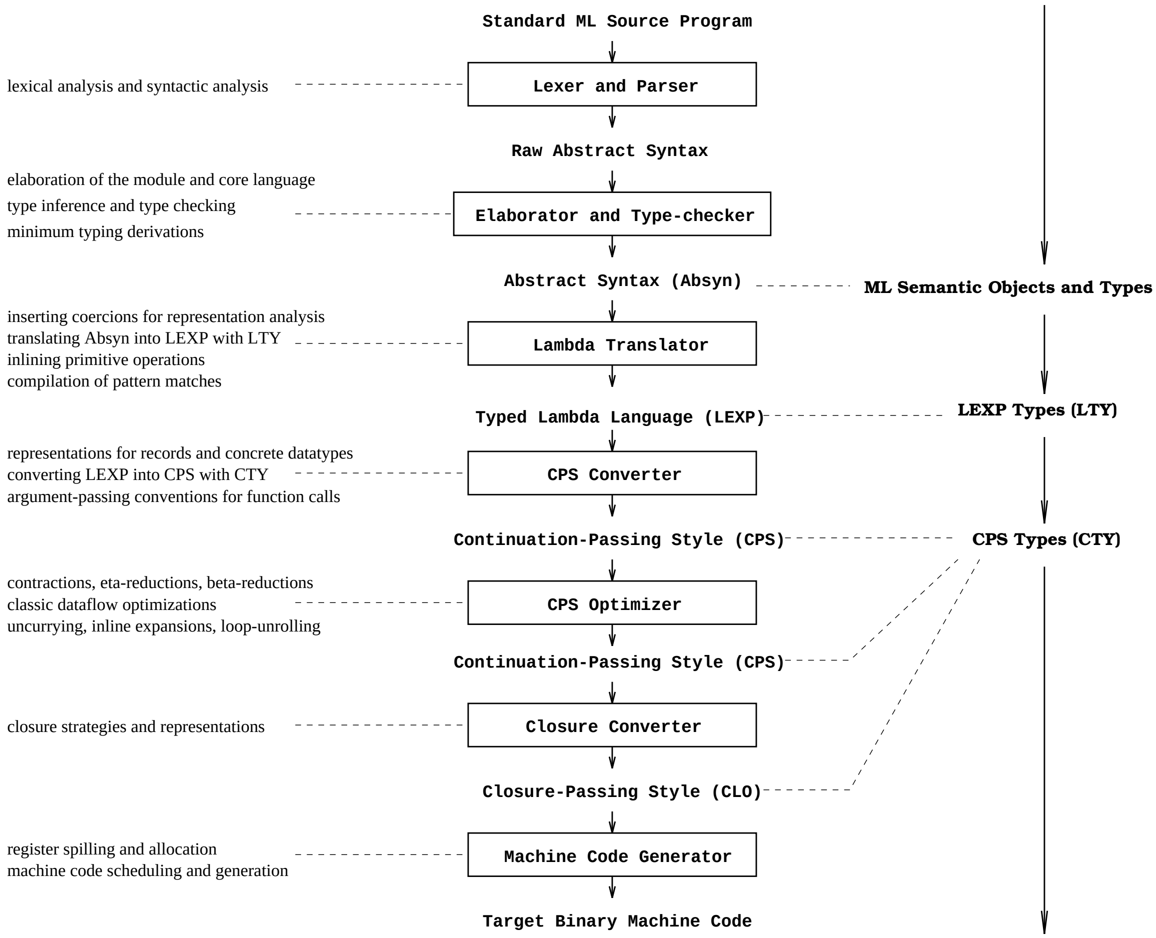


Figure 7: Overview of the new type-based SML/NJ compiler

the following seven phases (as shown in the center column of Figure 7), with each phase denoting certain set of program transformations:

Parsing The input stream of the SML source program is broken into tokens by a lexical analyzer, and then parsed according to a context-free grammar. This phase rewrites the source program into the *raw abstract syntax tree* (Ast)—an intermediate language that looks very much like the *concrete syntax* except that all of its punctuation tokens are discarded.

Elaboration and Type-Checking The raw abstract syntax tree does not contain any static semantic information; furthermore, it contains syntactic sugar and derived forms. This phase elaborates all program declarations and specifications (e.g., modules, module interfaces, type and value declarations, etc.) into semantic objects according to the static semantics [MTH90]; the types of all program variables are inferred and checked using the ML type inference algorithm [DM82, MTH90]. The raw abstract syntax tree is rewritten into a more compact form called *abstract syntax* (Absyn). Each declaration in Absyn is annotated with its corresponding static semantic or type information calculated during elaboration. To support representation analysis, the Absyn in our new compiler also remembers the details[2] of each abstraction and instantiation in the program. The details are described in Section 3.4.

Lambda Translation In this phase, the *abstract syntax* Absyn, annotated with static semantic information, is translated into a strict call-by-value lambda calculus (LEXP) augmented with data constructors, records, and primitive operators. Unlike the untyped lambda language used in the older SML/NJ compiler [AM91], LEXP is explicitly typed using a simple monomorphic type system (to be described in Section 3.5). The type information in LEXP is converted directly from the static semantic information attached to the Absyn. Coercion functions (in the same style as Leroy's [Ler92]) are inserted at each abstraction and instantiation site to correctly support abstraction and polymorphism. In addition, this phase also inserts the proper implementation of each equality test and assignment operator, and does pattern-match compilation. The details are described in Section 3.5.

CPS Conversion In this phase, the typed lambda language LEXP is converted into *continuation-passing style* (CPS). The CPS language is designed to match the execution model of a von Neumann register machine: functions in CPS can have multiple arguments, and variables (and function arguments) correspond closely to machine registers. Like the LEXP language, the CPS language here is also typed, but with an even simpler set of types (i.e., CTY, described in Section 3.6). With all the LTY information available in LEXP, this phase also determines the argument-passing convention for all function calls and returns, and the representation for all records and concrete data types. The details are described in Section 3.6.

[2] More specifically, everytime a polymorphic object is referenced, the compiler remembers its type instantiation in the Absyn; everytime a type abstraction occurs (e.g., in functor application or signature matching), both the actual type and the abstract type are recorded in the Absyn.

CPS optimizations The resulting CPS expression is then fed into many rounds of CPS optimizations [App92] such as contraction, eta-reduction, beta-reduction, inlining expansion, loop unrolling, and so on. Type correctness must be preserved during all optimizations and transformations.

Closure Conversion This phase makes explicit the access to nonlocal variables by converting CPS expressions into *closure-passing style* (CLO) [AJ89, App92]. CLO is almost the same as CPS, except that all functions in CLO do not contain free variables, so they can be translated into machine code directly. The new closure conversion algorithm described in Chapter 4 can be extended to utilize the CPS type information to support even more efficient closure representations (see Section 3.6.4).

Machine Code Generation Finally, based on the CLO expression and the CTY information, the compiler does register allocation and instruction scheduling, and then writes out the machine code.

In summary, the new type-based compiler achieves type-directed compilation by performing a sequence of program transformations on several typed intermediate languages. First, the source program (Absyn) annotated with static semantics is translated into an intermediate lambda language (LEXP) annotated with simple monomorphic types (LTY); all module constructs and polymorphic functions are translated into simple lambda functions and records using Leroy's *representation analysis* [Ler92]. The lambda expression is then converted into continuation-passing style (CPS) annotated with an even simpler CPS types (CTY); this conversion also determines the representation of records and concrete data types, and the argument-passing convention for function calls and returns, based on the LTY information. The very back end of the compiler then uses the CTY information to help generate more efficient code.

```
fun square (x : real) = x * x

fun sumsquare (l : real list) = let fun h ([], s : real) = s
                                      | h (a::r, s) = h(r, a+s)
                                 in h(map square l, 0.0)
                                end
```

Figure 8: Front end issues in core language

3.4 Front end issues

The main task of the front end (the first two phases in Figure 7) is parsing and elaboration. Elaboration determines whether the source program is well-typed and well-formed according to the Definition [MTH90], and records relevant semantic or type information in the static environment. In a type-based compiler, elaboration must also remember the details of each abstraction and instantiation (for both types and modules) in the program in order to implement module constructs and polymorphic functions.

```
signature SIG = sig type t
                    val p : t
                    val f : t -> (t * t)
                end

structure S = struct type t = real * real
                     val p = (3.0, 4.0)
                     fun f x = (x, p)
                     val q = 1.0
              end

functor F(A : SIG) = struct val r = (A.f(A.p), A.p)
                     end

structure U : SIG = S
abstraction V : SIG = S
structure W = F(S)
```

Figure 9: Front end issues in module language

3.4.1 Core language

Previous ML compilers cannot take advantage of static type information because they do not know how to deal with polymorphism. Leroy's representation analysis technique [Ler92] solves this problem by memorizing the actual instantiation of every polymorphic type and then inserting proper coercions later. Our compiler does the same. Each polymorphic variable and data constructor expression (or pattern) in the *abstract syntax* (Absyn) is annotated with two ML types (gathered during type inference): one is the polymorphic type itself, the other is its actual instantiation at this particular use. For example, in the SML program shown in Figure 8, where *map* is the standard map function on lists with polymorphic type $\forall\alpha\forall\beta.(\alpha \rightarrow \beta) \rightarrow (\alpha\ list \rightarrow \beta\ list)$; *map* is annotated with this polymorphic type, plus its instantiation $(real \to real) \to (real\ list \to real\ list)$. Similarly, the data constuctor, "::", is also annotated with its original polymorphic type $\forall\alpha.(\alpha * \alpha\ list) \to \alpha\ list$ plus its instantiation $(real * real\ list) \to real\ list$.

Table 3: Signature matching is transparent

	type in structure U	type in structure S
f	$(real * real) \to ((real * real) * (real * real))$	$\forall\alpha.\alpha \to (\alpha * (real * real))$
p	$real * real$	$real * real$

Table 4: Abstraction matching is opaque

	type in structure V	type in structure S
f	$t \to (t * t)$	$\forall\alpha.\alpha \to (\alpha * (real * real))$
p	t	$real * real$

3.4.2 Module language

The SML module language contains many instances of type abstractions and type instantiations, all of which must be dealt with carefully in order to correctly support type-directed compilation. In SML, basic modules, called *structures*, are encapsulated environments; module interfaces, called *signatures*, are environments associating specifications with component names and are used to both describe and constrain structures. Parameterized modules, called *functor*s, are functions from structures to structures. A functor's argument is specified by a signature and the result is given by a structure expression, which may optionally be constrained by a result signature. There are four module constructs where abstraction and instantiation may occur: *signature matching*, *abstraction declaration*, *functor application*, and *functor signature matching* (used by higher-order modules [Tof92, MT94]). In the following, we use the example in Figure 9 to explain what information must be recorded in the abstract syntax tree during the elaboration phase:

- Signature matching checks that a structure fulfills the constraints specified by a signature, and creates a new *instantiation structure* that is a restricted "view" of the original structure. The elaboration phase automatically generates a *thinning* function that specifies all the visible components and their types (or thinning functions if there are substructures) in the original structure, and their new types in the instantiation structure. For example, in Figure 9, U is bound to the result of matching structure S against signature SIG. Because signature matching in SML is *transparent* [MT91, Ler94, HL94], f and p in the instantiation structure U respectively have type $(real * real) \rightarrow ((real * real) * (real * real))$ and $real * real$ (see Table 3). These new types and their old types in structure S (shown in Table 3) will be recorded in its thinning function.

- Abstraction is treated the same as signature matching. Because matching for abstraction is *opaque* [MT91, Ler94, HL94], the elaboration phase also records the result signature in addition to the thinning function. For example, in Figure 9, V is an abstraction of structure S on signature SIG. V remembers the thinning function generated for doing signature matching of S against SIG, plus the actual signature SIG. During the elaboration of this abstraction, the type of f in S, $\forall\alpha.\alpha \rightarrow (\alpha*(real*real))$, is first instantiated into $(real*real) \rightarrow ((real*real)*(real*real))$ in signature matching, and then abstracted into $t \rightarrow (t * t)$ (see Table 4).

- Each functor application must remember its argument thinning function and its actual instantiation functor. For example, functor F in Figure 9 takes SIG as its argument signature, and returns a body structure that contains a value declaration r; the type of r is $(A.t * A.t) * A.t$. When F is applied to structure S, first, S is matched against the argument signature SIG to get the actual argument instance, say S'; then, the elaborator reconstructs a new body structure (for F), say B', assuming S' is the argument A. The component r in B' has type $((real * real) * (real * real)) * (real * real)$. The elaborator records both the thinning function generated when S is matched against SIG, and the actual functor instance of F, which has S' as its argument, B' as its result.

- Functor signatures are essentially "types" of functors. Given a functor signature $FSIG = (X : ASIG) : RSIG$, and a functor F, elaborating functor signature matching "functor G : $FSIG = F$" is equivalent to elaborate the functor declaration "functor $G(X\!: ASIG) : RSIG = F(X)$." Therefore, for each functor signature matching, the elaborator memorizes everything occurred in functor application $F(X)$ (see the case for functor application) plus the thinning function generated for matching $F(X)$ against the result signature $RSIG$.

3.4.3 Minimum typing derivation

Like the Damas-Milner type assignment algorithm W [DM82], The elaboration phase in our compiler also infers the most general type schemes for all SML programs. As a result, local variables are always assigned the most polymorphic types, even though they are used much less polymorphically. For example,

```
fun f(u,v) = let fun equal(x, y, z) = ((x=y) andalso (y=z))
              in equal(u*2.0, v*3.0, u+v)
             end
```

variable f has type $real * real \rightarrow bool$, the let-bound function *equal* is assigned a polymorphic type, $\forall\alpha.(\alpha * \alpha * \alpha) \rightarrow bool$ (assuming α is an equality type variable), even though it is only used monomorphically with type $(real * real * real) \rightarrow bool$.

We have implemented a "minimum typing derivation" phase in our compiler to give all local variables "least" polymorphic types. The derivation is done after the elaboration so that it is only applied to type-correct programs. Our algorithm, which is similar to Bjørner's algorithm M [Bjo94], does a bottom-up traversal of the abstract syntax Absyn. During the traversal, we mark all variables that are local (e.g., let-bound) or hidden because of signature matching. For each marked polymorphic variable v, we gather all of its actual type instantiations, say $\tau_1, ..., \tau_n$, and reassign v a new type—the *least general* type scheme that generalizes $\tau_1, ..., \tau_n$. The new type assigned to v is then propagated into v's declaration d, constraining other variables referenced by d.

In the above example, the let-bound *equal* function will be reassigned a new type $(real * real * real) \rightarrow bool$. Thus, the "=" operator can now be implemented as the primitive equality function on real numbers, which is much more efficient than the polymorphic equality operator. Moreover, because *equal* is no longer polymorphic, no coercion is necessary when it is applied to monomorphic values.

3.5 Translation into LEXP

The middle end of our compiler translates the abstract syntax Absyn into a simple typed lambda language called LEXP. During the translation, all the static semantic objects in Absyn, including types, signatures, structures, and functors, are translated into simple Lambda types (LTY); coercions are inserted at each abstraction and instantiation site (marked by the front end) to correctly support representation analysis. In this section, we explain the details of our translation algorithm, and present solutions to several practical implementation problems.

```
datatype lty = INTty | REALty | RECORDty of lty list
             | ARROWty of lty * lty | BOXEDty | RBOXEDty

type var = int
datatype 'a option = NONE | SOME of 'a
type dataconstr = conrep * lty

datatype con = DATAcon of dataconstr | INTcon of int
             | REALcon of string | STRINGcon of string

datatype lexp = VAR of var | INT of int | REAL of string
              | STRING of string | PRIM of primop * lty
              | FN of var * lty * lexp | APP of lexp * lexp
              | FIX of var list * lty list * lexp list * lexp
              | SWITCH of lexp * conrep list * (con * lexp) list * lexp option
              | CON of dataconstr * lexp | DECON of dataconstr * lexp
              | RECORD of lexp list | SELECT of int * lexp
              | RAISE of lexp * lty | HANDLE of lexp * lexp
              | WRAP of lty * lexp | UNWRAP of lty * lexp
```

Figure 10: The typed lambda language LEXP

3.5.1 The typed lambda language LEXP

An Lambda expression (LEXP) (i.e., the data type *lexp* shown in Figure 10) can be any of the following:

- a variable (`VAR`), a integer (`INT`), a real number (`REAL`), a string (`STRING`), or a primitive operator (`PRIM`);
- a lambda abstraction `FN(v,t,e)` where v is the argument, t is the argument type, and e is the function body;
- a function application (`APP`);
- a set of mutually recursive function definitions (`FIX`) where where `var list` denotes the function names, `lty list` denotes the types of these functions, and `lexp list` denotes the corresponding function definitions;
- a `SWITCH` expression that detects which constant or data constructor (from the `(con*lexp) list`) was used to build a data type value, and then evaluates the corresponding expression (see Appel [App92] for details);

- a data constructor applied to an argument (`CON`, the injection), or a value-carrying data constructor removed from an argument (`DECON`, the projection), where each data constructor (`dataconst`) is essentially a constructor representation (`conrep`) plus its Lambda type information;
- a `RECORD` expression with each field being an evaluated expression;
- a `SELECT(i,e)` expression that selects the ith field of an evaluated expression e;
- the `RAISE`-ing of an exception (annotated with the result type), or the evaluation of an expression in the scope of an exception `HANDLE`r;
- a primitive wrapper expression `WRAP(t,e)` that wraps an evaluated expression e with type t into exactly one word;
- a primitive unwrapper expression `UNWRAP(t,e)` that unwraps an evaluated expression e into the natural unboxed representation with type t.

A Lamda-type LTY (i.e., the data type *lty* shown in Figure 10) can be:

- a primitive integer type `INTty`, or a primitive real type `REALty`;
- a record type `RECORDty(l)` where the types of its fields are specified by l;
- a functor type `ARROWty(t,s)` where t is the argument type and s is the result type;
- a machine-level pointer type `BOXEDty` that points to objects with arbitrary representations;
- a special machine-level pointer type `RBOXEDty` that can only point to objects with standard boxed representations; the details of how and where to use `RBOXEDty` are discussed in Section 3.5.2

The LTY information essentially characterizes all the possible data representations used at runtime. In order to support coercion of data objects from one representation to another, we define a *coerce* operation on our lambda language, just like the "wrap" and "unwrap" functions used by Leroy [Ler92]. More specifically, *coerce* is a compile-time operation; given two LTYs t_1 and t_2, $coerce(t_1, t_2)$ returns a coercion function that coerces one *lexp* with type t_1 into another *lexp* with type t_2:

- If t_1 and t_2 are equivalent, no coercion is necessary, $coerce(t_1, t_2)$ returns the identity function.

- If one of t_1 and t_2 is `BOXEDty`, this requires coercing an arbitrary unboxed object into a pointer (or *vice versa*); the coercion here is a primitive `WRAP` or `UNWRAP` operation, written as $coerce(\texttt{BOXEDty}, t_2) = \lambda e.\texttt{UNWRAP}(t_2, e)$ and $coerce(t_1, \texttt{BOXEDty}) = \lambda e.\texttt{WRAP}(t_1, e)$.

- If one of t_1 and t_2 is `RBOXEDty`, this requires coercing an arbitrary unboxed object into a pointer (or *vice versa*); moreover, the object itself must be coerced into standard boxed representation (or *vice versa*); this coercion is similar to the *recursive wrapping* operations defined by Leroy [Ler92]. It is defined as $coerce(\texttt{RBOXEDty}, t_2) = coerce(dup(t_2), t_2)$, and $coerce(t_1, \texttt{RBOXEDty}) = coerce(t_1, dup(t_1))$, where the function *dup* is defined as follows:

  - $dup(\texttt{RECORDty}[x_1, ..., x_n]) = \texttt{RECORDty}[\texttt{RBOXEDty}, ..., \texttt{RBOXEDty}]$;
  - $dup(\texttt{ARROWty}(x_1, x_2)) = \texttt{ARROWty}(\texttt{RBOXEDty}, \texttt{RBOXEDty})$;
  - $dup(x) = \texttt{BOXEDty}$, for all other LTY x.

- If $t_1 = \texttt{RECORDty}[a_1, ..., a_n]$ and $t_2 = \texttt{RECORDty}[r_1, ..., r_n]$, we first build a list of coercions $[c_1, ..., c_n]$ for every field, where $c_i = coerce(a_i, r_i)$ for $i = 1, ..., n$. Assume v is a new lambda variable that corresponds to the old record, then the coercion from t_1 to t_2 is $\lambda e.\texttt{LET}(v, e, \texttt{RECORD}[f_1, ..., f_n])$ where each field of the new record is defined as $f_i = c_i(\texttt{SELECT}(i, \texttt{VAR}\ v))$ for $i = 1, ..., n$. Here the `LET(x,y,z)` expression is an LEXP idiom, equivalent to the expression `APP(FN(x,BOXEDty,z),y)`.

- If $t_1 = \texttt{ARROWty}(a_1, r_1)$ and $t_2 = \texttt{ARROWty}(a_2, r_2)$, we first build the coercions c_a and c_r for both the argument and the result, that is, $c_a = coerce(a_2, a_1)$ and $c_r = coerce(r_1, r_2)$; then assume u and v are two new lambda variables, the coercion from t_1 to t_2 is $\lambda e.\texttt{FN}(u, a_2, \texttt{LET}(v, \texttt{APP}(e, c_a(\texttt{VAR}\ u)), c_b(v)))$.

3.5.2 Translating static semantic objects into LTY

The abstract syntax (Absyn) is translated into the lambda language LEXP through a simple top-down traversal of the Absyn tree. During the traversal, all static semantic types used in Absyn must be translated into LTYs. A signature or structure object s is translated into `RECORDty` where each field is the LTY translated from the corresponding component in s; a functor object is translated into `ARROWty` with the argument signature being the argument LTY, and the body structure being the result LTY. The translation of an ML type t into LTY is done using the following algorithm (see the function `ty2lty` in Figure 11 for the pseudo code):

```
fun ty2lty(t) =
   (mark all type variables in t that ever appear in a constructor type;
    return lty(t))

fun lty(∀α.σ) = lty(σ)
  | lty({l1 : τ1, ..., ln : τn}) = RECORDty[lty(τ1), ..., lty(τn)]
  | lty(τ1 → τ2) = ARROWty(lty(τ1),lty(τ2))
  | lty(int) = INTty
  | lty(bool) = INTty
  | lty(unit) = INTty
  | lty(real) = REALty
  | lty(α) = if α is a marked type variable then RBOXEDty else BOXEDty
  | lty(t) = if the constructor type t is rigid then BOXEDty else RBOXEDty
```

Figure 11: Translating ML type into LTY

```
signature VECTOR = sig type 'a vec
                       val tabulate : (int -> 'a) -> 'a vec
                       val vdotvp : real vec * real vec -> real
                   end

functor F(V : VECTOR) = struct val v = V.tabulate(fn _ => 1.0)
                               val z = V.vdotvp(v,v)
                        end

structure V3 = struct
                 type 'a vec = 'a * 'a * 'a
                 fun tabulate f = (f 0, f 1, f 2)
                 val vdotvp((x1:real,x2,x3),(y1,y2,y3)) = (x1*y1+x2*y2+x3*y3)
               end

structure S = F(V3)
```

Figure 12: Flexible constructor type must be recursively boxed

- Type variables in t are divided into two kinds: those that ever appear in constructor types,[3] such as α in type $(\alpha * \alpha\ list) \rightarrow \alpha\ list$ or β in type $(\beta\ ref * \beta) \rightarrow unit$, are translated into `RBOXEDty`; all other type variables, such as γ in $\gamma * \gamma \rightarrow \gamma$, are translated into `BOXEDty`;

[3] Record type constructors and function type constructors ("→") are not counted here.

- If t is a polymorphic type, with the form as $\forall\alpha_1 ... \forall\alpha_n.\tau$, all quantifications are just ignored; the LTY for t is just the LTY translated from τ.
- Primitive types `int`, `bool` and `unit` are translated into `INTty`; `real` is translated into `REALty`; all other primitive types are translated into `BOXEDty`;
- The arrow type constructor "$\rightarrow$" for functions is translated into `ARROWty`; the record type constructor is translated into `RECORDty`, with its fields by symbolic label.
- All *rigid* constructor types,[4] such as *ByteArray*, α *list*, and $(real * real)$ *ref*, are translated into `BOXEDty`;
- All *flexible* constructor types are translated into `RBOXEDty`.

For example, the signature *VECTOR* in Figure 12 is translated into the LTY t_1 = `RECORDty` $[t_2, t_3]$, in which the component *tabulate* has the LTY t_2 and the component *vdotvp* has the LTY t_3, where

$$t_2 = \texttt{ARROWty (ARROWty (INTty,RBOXEDty),RBOXEDty)},$$

$$t_3 = \texttt{ARROWty (RECORDty [RBOXEDty, RBOXEDty],REALty)}.$$

Functor F is translated into the LTY t_4 = `ARROWty` $(t_1,\ t_5)$, where the result type is the LTY t_5:

$$t_5 = \texttt{RECORDty [RBOXEDty, REALty]}.$$

Structure *V3* has the LTY t_6 = `RECORDty` $[t_7, t_8]$, in which the component *tabulate* has the LTY t_7 and the component *vdotvp* has the LTY t_8 where

$$t_7 = \texttt{ARROWty (ARROWty (INTty,BOXEDty), RECORDty[BOXEDty, BOXEDty, BOXEDty])},$$

$$t_8 = \texttt{ARROWty (RECORDty } [t_9, t_9]\texttt{, REALty)},$$

$$t_9 = \texttt{RECORDty [REALty, REALty, REALty]}.$$

Finally, the result structure S from functor application *F(V3)* has the LTY t_{10} where

$$t_{10} = \texttt{RECORDty } [t_9, \texttt{REALty}].$$

To understand why flexible constructor types (with arity > 0) must be translated into `RBOXEDty`, let us look at the functor F in Figure 12 more closely. Here, the type `'a vec` in the argument signature is flexible, and we do not know its exact data representation

[4] Following the Definition and Commentary [MTH90, MT91], all type constructor names, defined as type specification in signatures, are *flexible*; all other type constructor names are *rigid*.

until we know the actual instantiation of `'a vec` at functor application time. But, in the body of functor F, the type `'a vec` may be instantiated into `real vec`, as is done in the function application "`V.tabulate(fn _ => 1.0)`." Clearly, since we do not know how to coerce between `'a vec` and `int vec`, they must use exactly the same data representation, that is, the standard boxed representation.

Flexible constructor types are abstract; during the signature matching or functor application, they may be instantiated into any type. For example, type `'a vec` in the functor signature of F is instantiated intto a tuple type "`'a * 'a * 'a`" (see the definition of structure *V3* in Figure 12) during functor application *F(V3)*.

Rigid constructor types, however, can only be concrete data types. This is because if a rigid constructor type is bound to a non-data-type SML type, it must be done through type abbreviation. But all type abbreviations are expanded during elaboration (see Definition [MTH90]).

As we discussed in Section 3.2, concrete data types—whether they are polymorphic or not—always use the same data representations. If a data object needs to be put into or fetched out of a certain concrete data type representation, it must be recursively "wrapped" or "unwrapped." This is why type variables that appear in constructor types are translated into `RBOXEDty`. By doing this, expensive coercions between polymorphic and monomorphic data type objects can be avoided.

3.5.3 Translating Absyn into LEXP

Now that we have explained how to translate static semantic objects into LTY and how to coerce from one LTY to another, the translation of Absyn into LEXP is straightforward:

polymorphic variables: Given a polymorphic variable v in Absyn, the front end has annotated every use of v with its polymorphic type σ plus its actual instantiation τ. Assume that σ and τ are translated into LTYs s and t, variable v is then translated into the LEXP expression $\mathit{coerce}(s,t)(\texttt{VAR}\ v)$.

pattern matching: Pattern matches are compiled in the same way as in the old compiler [AR92, AM91, App92]. The only difference is that we have to insert coercions around polymorphic data constructor projections (`DECON`) (see the next item).

polymorphic data constructors: Polymorphic data constructors are treated the same as polymorphic variables except that coercions are applied to data constructor injections (`CON`) and projections (`DECON`). For example, the projection function for the list *cons* constructor "::" appearing in function h in Figure 8 has the polymorphic type $\forall\alpha.\alpha\ list \rightarrow (\alpha * \alpha\ list)$; this is translated into the LTY:

$$s = \quad \texttt{ARROWty(BOXEDty,RECORDty}\ [\texttt{RBOXEDty}, \texttt{BOXEDty}]).$$

But this projection function for :: is instantiated into type $real\ list \rightarrow (real * real\ list)$, which is translated into the LTY:

$$t = \quad \texttt{ARROWty(BOXEDty,RECORDty}\ [\texttt{REALty}, \texttt{BOXEDty}]),$$

so the :: projection must be coerced from s to t before being applied to the actual argument list.

polymorphic primitive operators: Polymorphic primitive operators whose implementations are known at compile time can be specialized based on their actual type instantiations. For example, polymorphic equality, if used monomorphically, can be translated into primitive equality; integer assignments and updates can use unboxed update;[5] the function composition operator can be nicely specialized, avoiding many expensive coercions.

signature matching: Suppose structure S is matched against signature *SIG*, and U is the result instantiation structure; then the thinning function generated by the front end is translated into a coercion c, which fetches every component from S, and coerces it to the type specified in U. If S is denoted by v, then the translation of this signature matching is simply $c(v)$.

abstraction: Abstraction is translated in the same way as signature matching, except that the result $c(v)$ must also be coerced into the LTY for the signature *SIG*. Assume that the LTYs for U and *SIG* are respectively u and s, then the abstraction of structure S under *SIG* is $(coerce(u,s))(c(v))$.

functor application: Suppose the argument signature of functor F is *SIG* and F is applied to structure S. The front end has recorded the thinning function for matching S against *SIG* and the actual functor instance F' for F. As before, assume the result of matching S against *SIG* is $c(v)$, and F is denoted by the LEXP expression f, and the LTYs for F and F' are respectively s and t, then the LEXP expression for F' is $f' = (coerce(s,t))(f)$, and functor application $F(S)$ is translated into $\texttt{APP}(f', c(v))$.

[5] In order to support generational garbage collection [LH83, Ung86], most compilers must do some bookkeeping at each update so that the pointers from older generation to youngest generation are correctly identified. *Unboxed update* is a special operator that always assigns an unboxed value into a reference cell; unlike other updates, unboxed updates will not introduce any new pointers, thus no extra bookkeeping is necessary.

3.5.4 Practical issues

In practice, a naive implementation of the above translation algorithm can lead to particularly large LEXP expressions, because of large LTYs and excessive coercion code. This problem is extremely severe for programs that contain many of the functor applications, and large structure and signature expressions. For example, the top-level linking program for the SML/NJ compilation manager CM [HLPR94] contains only 78 lines of source code and 9 functor applications, but many signatures involved (they are externally defined) are extremely large. As a result, its LEXP expression is several orders of magnitude larger than its Absyn form. This makes the compilation of such programs become several orders of magnitude slower, and makes it consume an intolerably large amount of memory.

The simplest way to decrease the size of coercion code is to avoid unnecessary *eta redexes* introduced by the *coerce* procedure. We make sure that coercing two equivalent LTYs never introduces any coercion code. For example, suppose $t_1 = \texttt{RECORDty}[a_1, ..., a_n]$ and $t_2 = \texttt{RECORDty}[r_1, ..., r_n]$, then $coerce(t_1, t_2)$ first builds a list of coercions for each field $c_i = \ coerce(a_i, r_i)$ where $i = 1, ..., n$. If all of the c_is are identity functions, then no coercion code is built; otherwise everything proceeds as usual. This check is also done for coercing two `ARROWty`s.

In certain cases, naive translation may drag in some large LTYs that are mostly useless. For example, to compile the following code,

```
val _ = (Compiler.Control.CG.calleesaves := 3;
         Compiler.AllocProf.reset())
```

we really only need know that variable `calleesaves` has type *int*, and variable `reset` has type $unit \rightarrow unit$. However, our translation algorithm will have to include the type of structure `Compiler` which may contain hundreds of components. So we extend the *lty* and *lexp* language in Figure 10 with the following new constructs:

```
datatype lty = ...... | GRECty of (int * lty) list | SRECORDty of lty list

datatype lexp = ...... | SRECORD of lexp list
```

Here, `SRECORDty` and `SRECORD` are same as `RECORDty` and `RECORD`, except that they are used particularly for module constructs (i.e., structure and signature objects). The `GRECty` LTY is used to type external structures such as the above `Compiler` structure; each `GRECty` specifies a subset of its actual fields (and their corresponding LTYs) that are interesting to the current compilation unit. The LTYs for all external structure identifiers are inferred during the translation phase, rather than being translated from their corresponding static semantic objects. For example, the LTY for structure `Compiler` in the above example will be:

`GRECty`[(3, `GRECty`[(0, `GRECty`[(43,`INTty`)])]), (7, `GRECty`[(0, `ARROWty`(`INTty`,`INTty`))])]

where we assume that `Control` and `AllocProf` are respectively the 3th and 7th fields of `Compiler`, `CG` is the 0th field of `Control`, `calleesaves` is the 43th field of `CG`, and `reset` is the 0th field of `AllocProf`.

The most effective way to avoid excessive coercions is to share coercion code between equivalent pair of LTYs. More specifically, we build a hash table during the translation phase, with a pair of LTYs (s, t) as the index; the contents of the hash table is a lambda variable that corresponds to a "shared" coercion code from s to t, expressed in an LEXP expression as `FN`$(x, s, (\mathit{coerce}(s, t))(x))$. Every time we need to insert a coercion from s and t, we check the hash table, and retrieve the corresponding lambda variable if it is in the table; otherwise we add a new entry for this (s, t) to the table. Finally, at the end of the translation phase, all "shared" coercions are defined in the top-level of the resulting LEXP expression.

Coercions introduced in the *coerce* procedure are normally inlined in the CPS optimization phase, because they are applied just once. "Shared" coercions are often not inlined, because they may cause excessive code explosion. Because "shared" coercions can be more expensive than general "inlined" coercions, we only use this hashing approach for coercions between module objects. This compromise works extremely well in practice, mainly because it is large module objects that are causing the "excessive coercion code" problem. Moreover, since module-level coercions are not executed often, the generated code is not noticeably slower.

3.6 Typed CPS back end

One nice thing about our compiler is that all seven phases (as shown in Figure 7) are completely independent of each other. Each phase is a collection of transformations from one intermediate form to another, thus the internal interfaces are very clean. For example, the front end checks the source program and verify that it is well typed and well formed according to the static semantics. The middle end, which translate Absyn into LEXP, makes the implementation decision for pattern matching, module constructs, and polymorphic functions; this requires translating static semantic objects into LTYs and inserting proper coercions at all instantiation and abstraction sites, but it does not need

```
datatype value = VAR of var | INT of int | REAL of string | STRING of string

datatype accesspath = DIRp | SELp of int * accesspath

datatype cexp
  = RECORD of record_kind * (value * accesspath) list * var * cexp
  | SELECT of int * value * var * cty * cexp
  | APP of value * value list
  | FIX of function list * cexp
  | SWITCH of value * var * cexp list
  | BRANCH of branch_prim * value list * var * cexp * cexp
  | SETTER of setter_prim * value list * cexp
  | LOOKER of looker_prim * value list * var * cty * cexp
  | ARITH of arith_prim * value list * var * cty * cexp
  | PURE of pure_prim * value list * var * cty * cexp
withtype function = fun_kind * var * var list * cty list * cexp

datatype cty = INTt | PTRt of int option | FUNt | FLTt | CNTt
```

Figure 13: The typed CPS language

to worry about how records and functions (i.e. those with LTYs such as `RECORDty` and `ARROWty`) are actually represented at runtime.

The CPS back end of our compiler contains four phases: conversion of LEXP into CPS, CPS optimization, closure conversion, and machine code generation. The CPS conversion phase not only converts the LEXP expression into *continuation-passing style* (CPS), but also makes the implementation decisions for records, data constructors, function applications, and switch statements.[6] The resulting CPS expression, annotated with CPS types (CTY), is fed into many rounds of CPS optimization (described by Appel [App92]), and then converted into *closure-passing style* (CLO) by the closure converter. Finally, based on the CLO expression and the CTY information, the compiler does the register allocation and instruction scheduling, and writes out the machine code.

3.6.1 The typed CPS language

Figure 13 presents the new typed CPS language (defined as the data type *cexp*) and the new CPS types "CTY" (defined as the data type *cty*). As in the old CPS language [App92], the

[6] See Appel [App92] for details on how to CPS-convert data constructors and switch statements.

arguments to a function (or primitive operators) are always values (variables or constants, defined as the data type *value*). A CPS expression can be:

- a `RECORD(k,ul,v,ce)` expression; here, k specifies the record kind (i.e., vector, closure, etc.); ul is a list of its elements, each can be a *direct* value (`DIRp`), or a value accessible through certain path (`SELp`); the result binds to a CPS variable v, which can be referenced in the rest of the CPS expression ce; notice that the CTY of v is not specified here, but it can always be reconstructed;

- a `SELECT(i,u,v,t,ce)` expression which binds the i-th field of u to v; the CTY of v is t;

- a function (or a continuation) application (`APP`);

- a set of function definitions (`FIX`); each function (the type *function*) specifies its function kind `fun_kind` (i.e. continuation, escaping function, etc.), its function variable name, its list of arguments, and the CTYs of these arguments, and the function body;

- a primitive `SWITCH` expression (switches on integers);

- a set of branch primitive operations (`BRANCH`);

- a set of primitive operations with side-effects (`SETTER`);

- a set of access (`LOOKER`), arithmetic (`ARITH`), and other miscellaneous (`PURE`) primitive operations; each of these operations takes several arguments (`value list`), and binds the result to a CPS variable with a CTY specified. The `PURE` operators include special primitive "wrapper" and "unwrapper" operators for integers (`iwrap` and `iunwrap`), reals (`fwrap` and `funwrap`), and other pointer objects (`wrap` and `unwrap`).

The CPS type language CTY is even simpler than LTY. It only specifies whether a CPS variable is an integer (`INTt`), a real (`FLTt`), a function (`FUNt`), a continuation (`CNTt`), or a pointer to another heap record (`PTRt`). The pointer type `PTRt` carries a length value option, which denotes the length of the heap object it points to; if the object length is unknown at compile time (e.g., for objects such as concrete data types or polymorphic objects), the length field contains nothing. The translation from LTY to CTY is quite straight-forward, as shown by the following `lty2cty` function:

```
fun lty2cty(INTty) = INTt
  | lty2cty(REALty) = FLTt
  | lty2cty(ARROWty _) = FUNt
  | lty2cty(BOXEDty) = PTRt(NONE)
  | lty2cty(RBOXEDty) = PTRt(NONE)
  | lty2cty(RECORDty l) PTRt(SOME(length l))
```

More specifically, `INTty` is translated into `INTt`, `REALty` into `FLTt`, `ARROWty` into `FUNt`, `BOXEDty` and `RBOXEDty` into `PTRt(NONE)`, and `RECORDty l` to `PTRt(SOME k)` where k is the length of the record. Notice since the CPS conversion phase has made implementation decisions for records and functions, the CTY is not concerned with the details of `RECORDty` and `ARROWty`.

3.6.2 Converting LEXP into CPS

The overall structure and algorithm of our CPS conversion phase is almost same as the one described by Appel [App92]. The conversion function $\mathcal{F}$ takes two arguments: an LEXP expression E and a "continuation" function c of type $value \rightarrow cexp$; and returns a CPS expression as the result. Unlike Appel [App92], during the conversion process, we also gather the LTY information for each LEXP expression, and maintain an LTY environment for all CPS variables. The LTY information is not only used to help make implementation decisions for records and function calls, but also is translated into CTY to annotate CPS variables.

Converting LEXP records is the most interesting case. Given an LEXP expression `RECORD`$[u_1, u_2, ..., u_n]$, suppose the LTY for each u_i is t_i ($i = 1, ..., n$), we can represent the record using virtually any layout:

- The simplest way is to box every field, with integers tagged and reals boxed (see Figure 3). This is the approach used in the old non-type-based SML/NJ compiler [AM91, App92]. This approach is is clearly inefficient.

- If all t_is are `REALty`, the record can be represented as a "flat" real vector (as shown in Figure 4 in Section 3.2).

- If the runtime system supports descriptors such as bitmaps that specify the boxity of each field, the boxed and unboxed field can be simply mixed; the bitmap descriptor for this record can be inferred from its LTY (as shown in Figure 5a in Section 3.2).

- Another way to layout boxed and unboxed values is to reorder the fields so that all unboxed values are put ahead of all boxed values (as shown in Figure 5b in Section 3.2).

> The advantage is that we can use a simple descriptor that specifies the length of the unboxed chunk and the boxed chunk. This kind of descriptor is also cheaper for the garbage collector to interpret than a bitmap.

Clearly, the LEXP `SELECT` expressions must be converted following the layout convention used for records.

The CPS conversion also decides the argument-passing convention for all function calls and returns. Given an LEXP function $f = \texttt{FN}(v, t, e)$, depending on what t is, we can convert f into a multi-argument function; these arguments essentially correspond to target-machine registers. If t is a record type `RECORDty` l, and the length of the record (say n) is sufficiently small,[7] we pass all arguments of f in n registers. Assume v_1, ..., v_n are n new CPS variables, where each v_i denotes the i-th field of v with type t_i, and suppose $vl = [v_1, ..., v_n]$, $cl = [t_1, ..., t_n]$, and that k is the return continuation for f, then

$$\mathcal{F}(\texttt{FN}(v,t,e), c) = \texttt{FIX}([(fk,f,k{::}vl,\texttt{CNTt}{::}cl,\ ce)],\ c(\texttt{VAR}\ f)),$$

where ce is the CPS conversion of the lambda expression `LET`(v,`RECORD` [`VAR` v_1, ... `VAR` v_n], e) (with return continuation k), and fk is f's function kind.

Similarly, if f is the function defined as above, v_1, ..., v_n are n new CPS variables, and ul = [`VAR` v_1, ..., `VAR` v_n], converting the function application of f is defined as

$$\mathcal{F}(\texttt{APP}(f,e),c) = \texttt{FIX}(kl,(\mathcal{F}(e,\ \lambda u.hdr(\texttt{APP}(f, k :: ul)))))$$

where k is the return continuation defined by kl from c, and hdr is a series of `SELECT` expressions that fetch out all n fields from u into v_1, ..., v_n.

CPS conversion of function definition and application may introduce redundant record creation and selection code. Fortunately, they can be eliminated by the CPS optimization phase [App92].

Finally, the primitive coercion operations, `WRAP`(t,e) and `UNWRAP`(t,e), are converted into corresponding CPS primitive operations. Based on whether t is `INTty`, `REALty`, or other pointer types, `WRAP` and `UNWRAP` are translated into `iwrap` and `iunwrap`, `fwrap` and `funwrap`, or `wrap` and `unwrap`.

3.6.3 CPS optimizations

When the CPS conversion phase is finished, the compiler has made most of the implementation decisions for almost all program features and objects: structures and functors are compiled into records and functions; polymorphic functions are coerced properly if they are being used less polymorphically; pattern matches are compiled into switch statements;

[7] A compilation parameter serves as a threshold value; the current value we use is 10.

concrete data types are compiled into tagged data records or constants; records are laid out appropriately based on their types; and the function calling conventions are mostly decided.

The resulting CPS program, however, is very inefficient. The job of the CPS optimizer is to apply many simple transformations (such as β-reduction, constant folding, η-reduction, uncurrying), and to rewrite the CPS program into a smaller and more efficient one with the same semantics.

Because CPS is now annotated with CTY information, the CPS optimizer must faithfully preserve and propagate the CTY information in all transformations. This turns out to be very easy, since all CPS optimizations are naturally carried out in a "type-consistent" fashion. The only extra work is to copy the CTY information, when a function is being inlined or unrolled; this overhead is minimal since we have such a simple set of CPS types.

Besides those described by Appel [App92], two new CPS optimizations are performed:

- One is to cancel pairs of "wrapper" and "unwrapper" operations as long as they are of the same kind, that is, between `iwrap` and `iunwrap`, `fwrap` and `funwrap`, or `wrap` and `unwrap`. This optimization is very useful in eliminating extra cocercions introduced in the lambda translation phase.

- Another is to eliminate record copying operations; for example, suppose u is a CPS value, v_i is `SELECT`ed from the i-th field of u where $i = 1, ..., n$, and w is a record built from these v_i with all fields in the same order, then if u has the CTY `PTRt (SOME n)`, we can tell that w is just another copy of v. Because there is no object identity for records in ML, we can replace all uses of w by v and eliminate all copying operations. This is impossible in the old compiler [AM91, App92] where the length information for each record is not known in CPS.

3.6.4 Closure conversion

The representations of CPS functions (`FIX`es) are not exactly as a von Neumann machine would like them, since functions are nested with lexical scope. The closure conversion phase makes explicit the access to nonlocal variables by converting CPS expressions into *closure-passing style* (CLO). CLO is really a subset language of CPS, but each CLO function contains extra arguments that denote its environment for all non-local variables.

Our compiler uses the new space-efficient closure conversion algorithm described in Chapter 4. The only new problem caused by the typed CPS language is to decide where and how to put unboxed values (e.g., untagged integers or unboxed reals) in heap-allocated closures or machine registers. This can be divided into the following three cases:

- For all *known functions* (i.e., those whose call sites are all known at compile time), their environment can be allocated in both general purpose and floating-point registers, depending on the CTY information of all free variables.

- For all *continuation functions*, we assign several *floating-point* registers as callee-save registers, using the same approach described in Chapter 4. The floating-point portions of the closure (e.g., those with type `FLTt`) can be allocated in floating-point callee-save registers.

- If a closure containing both boxed and unboxed values has to be allocated on the heap, we can use any layout scheme used for normal program records (described in Section 3.6.2).

One optimization commonly used in closure conversion is to share closures among several functions. Unfortunately, closure sharing, if not done carefully, is not *safe for space complexity* [App92] (see Section 2.3.3). In our typed CPS back end, the object size information for each CPS variable is known most of the time (from its CTY, except those with type `PTRt (NONE)`), so we can easily identify whether it is safe to share two closures or not. As long as the extra objects being held are of constant size, sharing two closures would never cause asymptotic increase of the space usage.

3.6.5 Machine code generation

Since every variable in CLO is annotated with proper type information, register allocation and spilling are straightforward. If real numbers use unboxed representations, all variables with CTY `FLTt` will be assigned a floating-point register. If the number of free variables (of the same register class) at a certain point of the program exceeds the number of available registers, spilling is necessary.

We also believe that the CTY information can benefit the instruction scheduler, but we have not yet explored this in our current implementation.

3.7 Performance evaluation

Type-directed compilation should support much more efficient data representations. In order to find out how much performance gain we can get for different type-based optimizations, we have measured the performance of six different compilers on twelve SML benchmarks (see Section 2.4).

The six compilers we use are all simple variations of the Standard ML of New Jersey compiler version 1.03z. All of these compilers use the new closure conversion algorithm (described in Chapter 4), and with three general purpose callee-save registers [AS92], and all use tagged 31-bit integer representations. Other aspects of these compilers are close to those described by Appel [App92].

sml.nrp This version does not support representation analysis. No type information is propagated into the compiler middle end and back end. All data objects use uniform standard boxed representations. All functions take exactly one argument and return one result.

sml.fag This is the **sml.nrp** compiler with the *argument flattening* optimization turned on [App92]. More specifically, given a function f, if all of its call sites are known at compile time, the CPS optimizer can check if its argument is always a length-n record (or tuple); if this is the case, the argument record for f is flattened, and f is transformed into a n-argument function.

sml.rep This is our new type-based compiler that supports very basic representation analysis. This version does *not* use *minimum typing derivations* (see Section 3.4.3 and Bjørner [Bjo94]). Moreover, all floating point numbers still use boxed representations. Functions do *not* pass arguments and return results in floating-point registers.

sml.mtd This is the **sml.rep** compiler plus the implementation of *minimum typing derivations.*

sml.ffb This is the **sml.mtd** compiler extended with support for unboxed representations for floating point numbers. Function call and return now pass floating-point arguments in floating-point registers. Records that contain just float-point numbers are represented as "flat" real vectors (see Figure 4).[8] Records that contain both boxed and unboxed values are still represented as two layers, with each unboxed value being boxed separately (see Figure 4).

sml.fp3 This version is completely same as the **sml.ffb** compiler except three floating-point callee-save registers are used.

[8]Unfortunately, the SML/NJ version 1.03z still uses the old runtime system [App90]. Memory fetch (or store) of a floating-point number is implemented using two normal (one-word) memory-load (memory-store) instructions. Because of this, the "flat" real vectors we used need not be aligned at the double-word boundary.

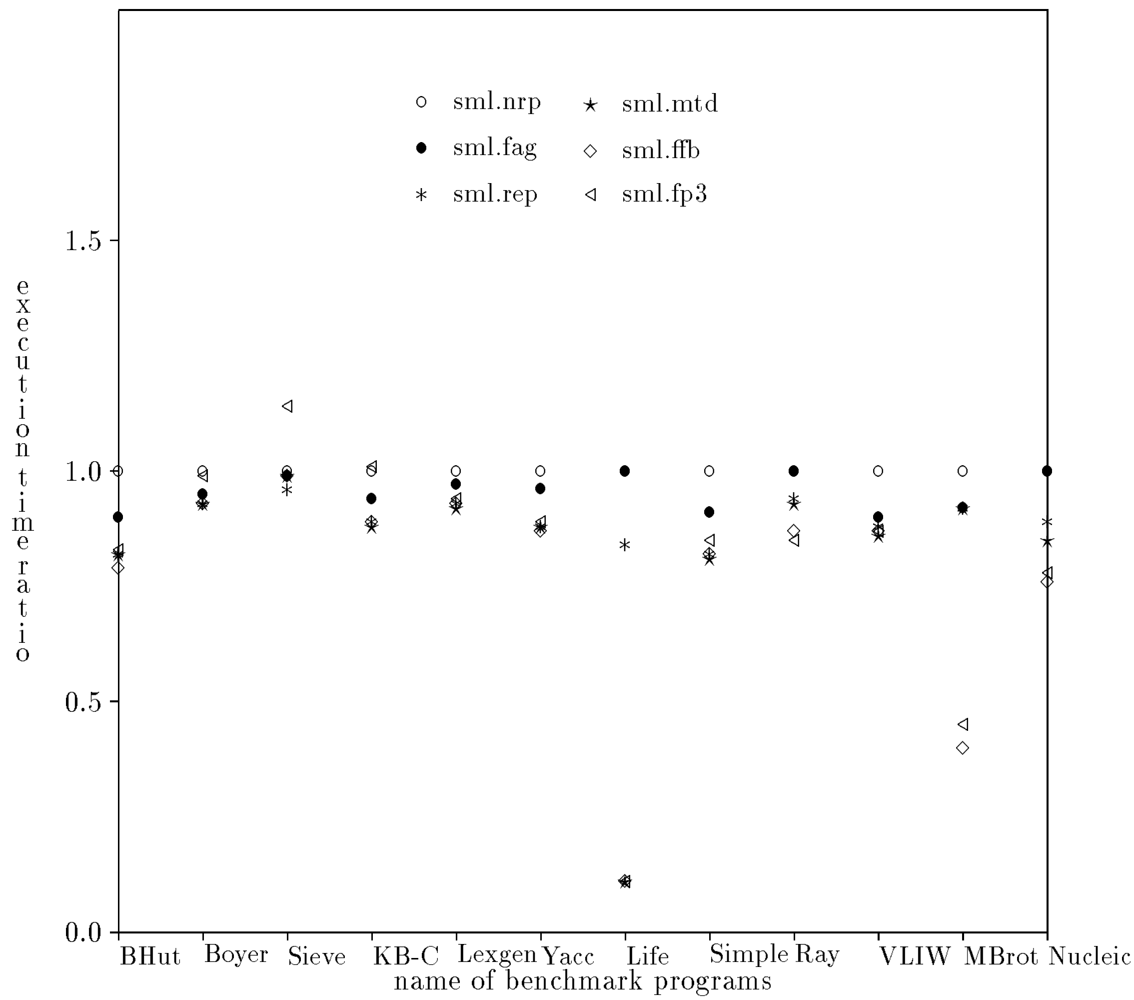


Figure 14: A comparison of execution time (illustration)

All measurements are done on a DEC5000/240 workstation with 128 mega-bytes of memory, using the methodology described in Section 2.4. In Figure 14 and Table 5, we list and compare the execution time of running all the benchmarks using the above six compilers. Only the execution time (user time plus system time, in seconds) for the **sml.nrp** compiler is shown; the performance for all other compilers are denoted by a relative ratio to the **sml.nrp** compiler. Similarly, Table 6, Table 7, and Table 8 respectively compares the total amount of heap allocation (in mega-bytes), the compilation time (in seconds), and the code size (in kilo-bytes). We can draw the following conclusions from these comparisons:

Table 5: A comparison of execution time

Program	**Base** (seconds)	**sml.nrp** (ratio)	**sml.fag** (ratio)	**sml.rep** (ratio)	**sml.mtd** (ratio)	**sml.ffb** (ratio)	**sml.fp3** (ratio)
BHut	28.0	1.00	0.90	0.82	0.82	0.79	0.83
Boyer	2.4	1.00	0.95	0.93	0.93	0.93	0.99
Sieve	30.6	1.00	0.99	0.96	0.99	0.99	1.14
KB-Comp	6.8	1.00	0.94	0.89	0.88	0.89	1.01
Lexgen	11.3	1.00	0.97	0.93	0.92	0.93	0.94
Yacc	4.8	1.00	0.96	0.88	0.88	0.87	0.89
Life	10.7	1.00	1.00	0.84	0.11	0.11	0.11
Simple	22.6	1.00	0.91	0.82	0.81	0.82	0.85
Ray	23.1	1.00	1.00	0.94	0.93	0.87	0.85
VLIW	14.0	1.00	0.90	0.88	0.86	0.87	0.87
MBrot	15.8	1.00	0.92	0.92	0.92	0.40	0.45
Nucleic	5.0	1.00	1.00	0.89	0.85	0.76	0.78
Average		1.00	0.95	0.89	0.83	0.77	0.81

Table 6: A comparison of total heap allocation

Program	**Base** (Mbytes)	**sml.nrp** (ratio)	**sml.fag** (ratio)	**sml.rep** (ratio)	**sml.mtd** (ratio)	**sml.ffb** (ratio)	**sml.fp3** (ratio)
BHut	506.0	1.00	0.76	0.57	0.57	0.54	0.54
Boyer	30.8	1.00	0.83	0.75	0.75	0.75	0.88
Sieve	186.0	1.00	0.97	0.89	0.96	0.96	1.13
KB-Comp	122.1	1.00	0.87	0.77	0.76	0.76	1.01
Lexgen	102.5	1.00	0.97	0.60	0.60	0.60	0.65
Yacc	63.1	1.00	0.94	0.70	0.70	0.70	0.71
Life	148.6	1.00	1.00	0.58	0.05	0.05	0.05
Simple	378.9	1.00	0.77	0.47	0.47	0.55	0.49
Ray	512.6	1.00	0.99	0.78	0.78	0.60	0.60
VLIW	172.9	1.00	0.84	0.65	0.65	0.66	0.66
MBrot	596.8	1.00	0.86	0.86	0.87	0.00	0.00
Nucleic	84.5	1.00	1.00	0.74	0.74	0.76	0.86
Average		1.00	0.90	0.70	0.66	0.58	0.63

- The type-based compilers perform uniformly better than older compilers that do not support representation analysis. The **sml.ffb** compiler gets nearly 19% speedup in execution time and decreases the total heap allocation by 36% (on average) over the older SML/NJ compiler (i.e., **sml.fag**) that uses uniform standard boxed representations. This comes with an average of 6% increase in the compilation time. The generated code size remains about the same.

Table 7: A comparison of compilation time

Program	**Base** (seconds)	**sml.nrp** (ratio)	**sml.fag** (ratio)	**sml.rep** (ratio)	**sml.mtd** (ratio)	**sml.ffb** (ratio)	**sml.fp3** (ratio)
BHut	49.1	1.00	1.01	1.00	1.12	1.16	1.20
Boyer	21.9	1.00	1.03	1.06	1.12	1.16	1.20
Sieve	28.5	1.00	1.02	1.04	1.08	1.08	1.13
KB-Comp	18.0	1.00	1.04	1.01	1.04	1.06	1.14
Lexgen	32.3	1.00	1.03	1.01	1.00	1.01	1.15
Yacc	114.1	1.00	1.04	1.07	1.08	1.10	1.20
Life	4.3	1.00	1.02	1.05	1.14	1.10	1.18
Simple	35.3	1.00	1.17	1.40	1.45	1.49	1.55
Ray	14.6	1.00	1.04	0.97	1.01	1.04	1.13
VLIW	161.7	1.00	1.02	1.06	1.09	1.05	1.12
MBrot	0.5	1.00	1.06	0.98	1.06	1.00	1.02
Nucleic	122.1	1.00	1.01	1.11	0.90	0.97	1.06
Average		1.00	1.04	1.06	1.09	1.10	1.17

Table 8: A comparison of code size

Program	**Base** (Kbytes)	**sml.nrp** (ratio)	**sml.fag** (ratio)	**sml.rep** (ratio)	**sml.mtd** (ratio)	**sml.ffb** (ratio)	**sml.fp3** (ratio)
BHut	85.1	1.00	0.92	0.95	0.95	0.98	1.01
Boyer	97.4	1.00	1.00	1.00	1.01	1.01	1.02
Sieve	66.0	1.00	0.96	0.94	0.93	0.93	0.98
KB-Comp	41.6	1.00	0.99	0.97	0.97	0.98	1.00
Lexgen	87.3	1.00	0.98	0.95	0.95	0.96	0.97
Yacc	320.0	1.00	0.99	0.98	0.98	0.97	0.98
Life	13.1	1.00	0.99	0.98	0.96	0.97	0.98
Simple	91.3	1.00	0.97	1.12	1.12	1.19	1.16
Ray	49.3	1.00	0.96	0.91	0.91	0.92	0.94
VLIW	349.5	1.00	0.98	0.95	0.95	0.95	0.96
MBrot	2.0	1.00	1.00	0.95	0.95	0.80	0.85
Nucleic	238.4	1.00	1.00	0.98	0.98	1.23	1.23
Average		1.00	0.98	0.97	0.97	0.99	1.01

- The simple, non-type-based argument flattening optimization in the **sml.fag** compiler gives an impressive 5% speedup. The optimization itself does slow down the compilation a little bit (4%).

- The **sml.rep** compiler, which supports passing argument in registers (but not floating-point registers), only improves the performance of the non-typed-based **sml.fag** compiler by about 6%. It does decrease heap allocation by an impressive 20%. We believe that most computation intensive parts (i.e., loops and recursions) are often *known functions*, so argument flattening can get most performance benefits of a type-based compiler such as **sml.rep**.

- The *minimum typing derivation* technique was intended to be useful in eliminating coercions; however, the major speedup of the **sml.mtd** compiler over **sml.rep** is from the **Life** benchmark where with *minimum typing derivation*, the polymorphic equality in a tight loop (testing membership of an element in a set) is successfully transformed into a monomorphic equality operator. Because the polymorphic equality implementation in SML/NJ is extremely slow (it uses runtime intepretation), **Life** is about 10 times faster. For all other benchmarks, **sml.mtd** performs about the same as **sml.rep**. Although there are much larger number of coercions in the lambda language in **sml.rep**, most these coercions are eliminated by the CPS optimization phase [App92] (through eta-reduction, inlining, constant folding, etc.).

- Another observation is that the **sml.mtd** compiler did not make the compilation any faster than **sml.rep** either. This is probably because the extra pass of deriving minimum types took away all the gains from coercion eliminations (in **sml.mtd**).

- Using three floating-point callee-save registers (i.e., the **sml.fp3** compiler) is not any better than using no floating-point callee-save registers. The slowdown mostly comes from benchmarks such as **Sieve** and **KB-Comp** whose tight loops frequently use *first-class continuations* and exception handlers, because more callee-save registers make the register state bigger (see Section 4.6 for more discussions). In addition, certain floating benchmarks such as **BHut**, **MBrot** and **Nucleic** also gets slower. We believe this is because the current implementation of closure conversion heuristic is not well-tuned for floating-point callee-save registers.

3.8 Related work

Statically typed languages with Hindley-Milner polymorphism have long been compiled using the uniform boxed representations, just like the dynamically typed languages such as Lisp and Scheme. The representation analysis technique, first proposed by Leroy [Ler92] (for ML-like languages) and Peyton Jones and Launchbury [PL91] (for Haskell-like languages), allows data objects whose types are not polymorphic to use more efficient unboxed representations. Leroy [Ler92] has also implemented representation analysis in his Gallium compiler for the Caml Light dialect of ML, and shown that it can result in important speedups on certain benchmarks. The work described in this chapter is a re-implementation of Leroy's techniques in the Standard ML of New Jersey compiler [AM91]. Unlike Leroy [Ler92], we concentrate more on practical issues such as how to implement type-directed compilation for the entire SML language (Caml has a much simpler module system than SML), and how to efficiently propagate type information through many rounds of transformations and optimizations.

Many people have worked on eliminating unnecessary "wrapper" functions introduced by representation analysis. Poulsen [Pou93] proposes a way to tag each type with a *boxity* annotation, and then statically determine when to use boxed representations and when to use unboxed representations. The major problem of his technique is that it is not easy to extend to the SML module system. Henglein and Jorgensen [HJ94] presents a term-rewriting method that translates a program with many coercions into one that contains "minimum" number of coercions (statically). Once again, it is not clear how their technique can easily be extended to the SML module language. Our compiler uses a simple *minimum typing derivation* [Bjo94] round in the front end to decrease the degree of polymorphism for all local and hidden functions. This is very easy to extend to the module system. We believe our approach can almost achieve the same result as "formally optimal unboxing" [HJ94]. Actually, we have shown that "wrapper" eliminations do not have much effect on performance in a highly optimizing compiler such as SML/NJ, simply because a simple compile-time contraction can eliminate most of the wrap and unwrap pairs (see Section 3.6.3).

Peterson [Pet89] presents a way to decide when to use boxed and unboxed representations using data flow analysis. It is not clear how much performance gain we can get from this kind of expensive analysis.

Harper and Morrisett [HM95] have recently proposed a type-based compilation framework called *Compiling with Intensional Type Analysis* for the core-ML language. They use a typed lambda calculus with explicit type abstractions and type applications as the intermediate language. Their scheme avoids recursive coercions by passing explicit type descriptors whenever a monomorphic value is passed to a polymorphic function. Since they have not implemented their scheme yet, it is unclear how well it would behave in practice. Because their proposal only addresses the core-ML language, we still do not know how easily their scheme can be extended to the SML module language.

3.9 Summary

We believe that type-based compilation techniques will be widely used in compiling statically typed languages such as ML in the future. The beauty of type-based representation analysis is that it places no burdens on the user: the source language does not change, programmers do not need to write coercions, and separate compilation works cleanly because interfaces are speficied using types.

By implementing a fully working type-based compiler for the entire SML language, we have gained experience with type-directed compilation, and solved many practical problems involved in the implementations. Our performance evaluation shows that type-based compilation techniques can achieve significant speedups on a range of benchmarks.

The number of ways to use type information to help generate better code seems to be endless; this chapter only discusses and evaluates a very small number of these optimizations. I intend to evaluate other optimizations in the future.

Chapter 4

Space-Efficient Closure Representations

Many modern compilers implement function calls (or returns) in two steps: first, a *closure* environment is properly installed to provide access for free variables in the target program fragment; and second, control is transferred to the target by a "jump with arguments (or results)." *Closure conversion*, which decides where and how to represent closures at runtime, is a crucial step in compiling functional languages. We have a new algorithm that exploits the use of compile-time control and data-flow information to optimize closure representations. By extensive closure sharing and by allocating as many closures in registers as possible, our new closure conversion algorithm reduces heap allocation by 32% and memory fetches for local/global variables by 46%, and improves the already-efficient code generated by the Standard ML of New Jersey compiler by about 14% on a DECstation 5000. Moreover, unlike most other approaches, our new closure allocation scheme satisfies the strong "safe for space complexity" rule, thus achieving good asymptotic space usage.

4.1 Introduction

Many compilers of functional languages take great efforts to optimize function calls and returns because they are the fundamental control structure. Before a function call, context information is saved from registers into a "frame." In a compiler based on Continuation-Passing Style (CPS), this "frame" is the closure of a continuation function [Ste78].

In a CPS-based compiler, a *closure* environment is constructed at each function (or continuation) definition site; it provides runtime access to bindings of variables free in the function (or continuation) body. Each function call is then implemented by first installing the corresponding closure environment, setting up the arguments (normally in registers), and then jumping to the target. Function returns are implemented in the same way, because in CPS, they are represented as calls to continuation functions.

A closure can be any combination of registers and memory data structures that gives access to the free variables [KKR*86, AS92]. The compiler is free to choose a closure representation that minimizes stores (closure creation), fetches (to access free variables), and memory use (reachable data).

We have developed a new algorithm for choosing good closure representations. As far as we know, our new closure allocation scheme is the first to satisfy all of the following important properties:

- Unlike stack allocation and traditional linked closures, our shared closure representations are safe for space complexity (see Section 4.2); at the same time, they still allow extensive closure sharing.
- Our closure allocation scheme exploits extensive use of compile-time control and data flow information to determine the closure representations.
- Source-language functions that make several sequential function calls can build one shared closure for use by all the continuations, taking advantage of callee-save registers.
- Because activation records (i.e., frames) are also allocated in the heap, they can be freely shared with other heap-allocated closures. Under stack allocation, this is impossible since stack frames normally have shorter lifetimes than heap-allocated closures.
- Tail recursive calls—which are often quite troublesome to implement correctly on a stack [Han90]—can be implemented very easily.
- All of our closure optimizations can be cleanly represented using continuation-passing and closure-passing style [AJ89] as the intermediate language.
- Once a closure is created, no later writes are made to it; this makes generational garbage collection and *call/cc* efficient, and also reduces the need for alias analysis in the compiler (since there are less side-effect operations).
- Because all closures are allocated either in the heap or in registers, first class continuations *call/cc* are very efficient, requiring no complicated stack hackery [HDB90].

Our new closure allocation scheme does not use any runtime stack. Instead, all closure environments are either allocated in the heap or in registers. This decision may seem controversial, because stack allocation is widely believed to have better reference locality, and deallocation of stack frames can be cheaper than garbage collection. Moreover, because heap allocated closures are not contiguous in memory, an extra memory write and read (of the frame pointer) is necessary at each function call. These assumptions do not hold in our algorithm for three reasons:

1. As we will show in Section 4.4, because most parts of continuation closures are allocated in callee-save registers [AS92], the extra memory write and read at each call can often be avoided. With the help of compile-time control and data flow information, the combination of shared closures and callee-save registers can often be comparable to or even better than stack allocation.

2. In Chapter 5, we show that stacks do not have a significantly better locality of reference than heap-allocated activation records, *even in a modern cache memory hierarchy.* Stacks do have a much better *write miss ratio*, but not a much better *read miss ratio.* But on many modern machines, the *write miss penalty* is approximately zero [Jou93, DTM94, Rei94].

3. The amortized cost of collection can be very low [App87] (also see Chapter 5), especially with modern generational garbage collection techniques [Ung86].

The major contribution of this chapter is a "safe for space" *closure conversion* algorithm that integrates and improves most previous closure analysis techniques [Kra87, AS92, Ste78, Roz84, Han90, Joh85], using a simple and general framework expressed in continuation-passing and closure-passing style [AJ89, AS92]. Our new algorithm extensively exploits the use of compile-time control and data flow information to optimize closure allocation strategies and representations. Our measurements show that on a DECstation 5000/240 the new algorithm reduces heap allocation by 32% and memory fetches for local/global variables by 46%; and improves the already-efficient code generated by the Standard ML of New Jersey compiler by about 14%.

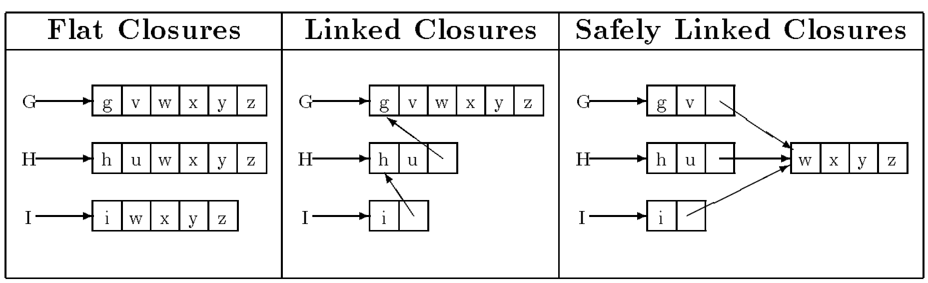
4.2 Safely linked closures

Optimization of closure representations is sometimes dangerous and unsafe for space usage (see Section 2.3.3). Traditional stack allocation schemes and linked closures obviously

```
fun f(v,w,x,y,z) =
  let fun g() =
        let val u = hd(v)
            fun h() =
              let fun i() = w+x+y+z+3
               in (i,u)
              end
         in h
        end
   in g
  end

fun big n = if n<1 then [0] else n :: big(n-1)

fun loop (n,res) =
  if n<1 then res
  else (let val s = f(big(N),0,0,0,0)()
         in loop(n-1,s::res)
        end)

val result = loop(N,[])
```

Figure 15: An example in Standard ML

Table 9: A comparison of three closure representations



violate the SSC rule (see Section 2.3.3) because local variable bindings are live until the function exits its scope, which may be after their last use. Flat closures do satisfy the SSC rule, but they require that variables be copied many times from one closure to another. Many of the closure strategies described by Appel and Jim [AJ88] and most stack-frame implementations also violate SSC, since dead variables remain in the frame until a function returns.

Obeying SSC can require extra copying of pointer values from an old closure that contains them (but also contains values not needed in a new context) into a new closure. One cannot simply "zap" the unneeded values in the old closure, since it is not known whether there are other references to the old closure. The challenge is to find efficient closure strategies that obey SSC while minimizing copying.

Our new algorithm uses *safely linked closures* (the 3rd column in Figure 9), which contain only those variables actually needed in the function, but avoids closure copying by grouping variables with same *lifetime* into a sharable record.

For example, consider the SML program in Figure 15 (this is same as the program shown in Figure 2 in Section 2.3.3). In Figure 9, we use G, H and I to denote the closure, and g, h, and i for code pointers. With flat closures (the 1st column in Figure 9), variables `w`, `x`, `y`, and `z` must be copied from the closure of `g` into the closure of `h`, and then into the closure of `i`, this is very expensive. With traditional linked closures (the 2nd column in Figure 9), closures for `h` and `i` are unsafely re-using the closure for `g`, retaining the variable `v` that is not free in `h` or `i`; moreover, accessing variables `w`, `x`, `y` and `z` inside I is quite expensive because at least two links needs to be traversed. By noticing that `w`, `x`, `y`, and `z` have same lifetime, the *safely linked closure* for `g` puts them into a separate record, which is later shared by closures for `h` and `i`. Unlike linked closures, the nesting level of safely linked closures never exceeds two, so they still enjoy very fast variable access time.

4.3 Continuations and closures

We will illustrate CPS-conversion (which is not new [Plo75, Ste78, Kra87, App92]), and our new closure analysis algorithm, on the example in Figure 16. The function `iter` iteratively applies function f to argument x until it converges to satisfy predicate p.

```
fun iter(x,p,f) =
  let fun h(a,r) = if p(a,r) then a
                   else h(f(a),a)
   in h(x,1.0)
  end
```

Figure 16: Function `iter` in Standard ML

$$
\begin{array}{lcl}
V & ::= & \textit{variable} \\
I & ::= & \textit{integer constant} \\
R & ::= & \textit{real constant} \\
P & ::= & \textit{arithmetic operator} \\
A & ::= & V \mid I \mid R \\
F & ::= & V_0(V_1,V_2,\ldots,V_n) = E \mid F_1 \texttt{ and } F_2 \\
 & & \\
D & ::= & D_1\ D_2 \mid \texttt{fun } F \mid \texttt{val } V = \texttt{select}(I,A) \\
 & \mid & \texttt{val } V = (A_1,A_2,\ldots,A_n) \\
 & \mid & \texttt{val } V = P(A_1,A_2,\ldots,A_n) \\
 & & \\
E & ::= & \texttt{if } V \texttt{ then } E_1 \texttt{ else } E_2 \\
 & \mid & \texttt{let } D \texttt{ in } E_1 \texttt{ end} \mid A_0(A_1,A_2,\ldots,A_n)
\end{array}
$$

Figure 17: Abstract syntax of CPS

4.3.1 Continuation-passing style

Continuation-passing style (CPS) is a subset of λ-calculus with certain syntactic properties. Unlike the λ-calculus, the order of evaluation in CPS is pre-determined. For the purposes of this chapter, we express CPS using ML notation, albeit severely constrained — see Figure 17. An atom A is a variable or a constant; a *record* is constructed from a sequence $(A_1, A_2, \ldots, A_n)$ of atoms. If v is bound to an n-element record, then the i^{th} field is fetched using `select`(i,v); The syntax for building records, selecting fields, applying primitive arithmetic operators, and defining mutually recursive functions (`fun` and F) must specify a continuation expression E that will use the result (via `let` expressions).[1] On the other hand, function application (shown in the last line on the right of Figure 17) does not specify a continuation expression — functions never *return* in the conventional sense. Instead, it is expected that many functions will pass *continuation functions* among their arguments. This function can be defined in the ordinary way (by `fun`), and will presumably be invoked by the callee in order to continue the computation.

Figure 18 shows the code of the function `iter` after translation into CPS, and after the loop-invariant continuation argument of `h` has been hoisted out of the loop [App94b]. Such

[1]Later in this chapter , we use `let` E_1 E_2 ... E_n `in` ... `end`. to denote a sequence of `let` expressions, e.g., `let` E_1 `in` (`let` E_2 `in` ... (`let` E_n `in` ... `end`) ... `end`) `end`.

```
fun iter(C,x,p,f) =
  let fun h(a,r) =
        let fun J(z) = if z then C(a)
                       else (let fun Q(b) = h(b,a)
                              in f(Q,a)
                             end)
         in p(J,a,r)
        end
   in h(x,1.0)
  end
```

Figure 18: Function `iter` after CPS-based optimizations

optimizations are performed after CPS-conversion, but before the closure analysis that is the subject of this chapter.

To ease the presentation, we use capital letters to denote continuations (e.g., `C`, `J`, and `Q`). We call those functions declared in the source program *user functions* (e.g., `iter`, `h`), and those introduced by CPS conversion *continuation functions* (e.g., `J`, `Q`). *Continuation variables* are those formal parameters (commonly placed as the first argument) introduced in CPS conversion to serve as return continuations (e.g., `C`). Functions such as `iter`, `p` and `f` are called *escaping functions*, because they may be passed as arguments or stored in data structures, which means that the compiler cannot identify all the places where they are called. All functions that do not escape are called *known functions* (e.g., `h`). We can do extensive optimizations on known functions since we know all of their call sites at compile time.

4.3.2 Closure-passing style

Continuation-passing style is meant to approximate the operation in machine language; a "function" in machine language is just an address in the executable program, perhaps with some convention about which registers hold the parameters—very much like a "jump with arguments." The notion of function in CPS is almost the same, except that CPS have nested lexical scope and may contain *free variables*. This problem is solved by adding a *closure* which makes explicit the access to all nonlocal variables.

Kranz [KKR*86, Kra87] showed that different kinds of functions should use different closure allocation strategies. For example, the closure for a *known function* (e.g., `h` in Figure 18) can be allocated in registers, because we know all of its call sites at compile time and can require that the caller always pass its free variables as extra arguments; on the other hand, the closure for an *escaping function* may have to be allocated as a heap record that contains both the machine code address of the function plus bindings for its free variables.

Conventional compilers use *caller-save* registers, which may be destroyed by a procedure call, and *callee-save* registers, which are preserved across calls. Variables that are not live after the call should be allocated to *caller-save* registers, which cuts down on register-saving.

```
fun iter(I,C0,C1,C2,C3,x,p,f) =
  let fun h(a,r,CR,p) =
        let fun J0(J1,J2,J3,z) =
               if z then
                 (let val C0 = select(0,J1)
                      val C1 = select(1,J1)
                      val C2 = select(2,J1)
                      val C3 = select(3,J1)
                   in C0(C1,C2,C3,J2)
                  end)
               else
                 (let fun Q0(Q1,Q2,Q3,b)
                                = h(b,Q2,Q1,Q3)
                      val f = select(4,Q1)
                      val f0 = select(0,f)
                   in f0(f,Q0,J1,J2,J3,J2)
                  end)
            val p0 = select(0,p)
         in p0(p,J0,CR,a,p,a,r)
        end
      val CR = (C0,C1,C2,C3,f)
   in h(x,1.0,CR,p)
  end
```

Figure 19: Function `iter` in after closure conversion

We wanted to adapt this idea to our continuation-passing intermediate representation. We did so as follows [AS92]: each CPS-converted user function f is passed its ordinary arguments, a continuation function c_0, and k extra arguments $c_1, ..., c_k$. The function "returns" by invoking c_0 with a "result" argument r and the additional arguments $c_1, ..., c_k$. Thus, the "callee-save" arguments $c_1, ..., c_k$ are handed back to the continuation. When this CPS code is translated into machine instructions, $c_1, ..., c_k$ will stay in registers throughout the execution of f, unless f needed to use those registers for other purposes, in which case

f must save and restore them. One could also say that the continuation is represented in $k+1$ registers $(c_0, ..., c_k)$.

In our previous work [AS92], we outlined this framework and demonstrated that it could reduce allocation and memory traffic. But, we did not have a really good algorithm to exploit the flexibility that callee-save registers provide.

Closure creation and use can also be represented using the CPS language itself [AJ89, KH89]. We call this *closure-passing style* (CLO). The main difference between CLO and CPS is that functions in CLO do not contain free variables, so they can be translated directly into machine code. In CLO, the formal parameters of each function correspond to the target machine registers, and heap-allocated closures are represented as CPS records.

Figure 19 lists the code of function `iter` after translation into CLO. Each continuation function and variable (e.g., `C,J,Q`) is now represented as a machine code pointer (e.g., `C0,J0,Q0`) plus three extra callee-save arguments (e.g., `C1-C3,J1-J3,Q1-Q3`).

The original function `J` (in Figure 18) had free variables `C,f,a,h`. With three callee-save registers, `C` becomes the four variables `C0,C1,C2,C3`, for an effective total of seven. When `J` is passed to `p` (line 19), these seven free variables—plus the machine code pointer for `J`'s entry point—must be squeezed into four formal parameters `J0,J1,J2,J3`. Where there are more than three free variables, some of the callee-save arguments must be heap-allocated records containing several variables each; thus, the `CR` closure-record appears as `J1` in the call on line 19.

Previous closure conversion algorithms [Ste78, KKR*86, AJ89] require memory stores for each continuation function. An important advance in our new work is that we allocate (in this example) only one record `CR` for the functions `J,Q,h`, *and this record is carefully chosen to contain loop-invariant components, so that it can be built outside the loop.*

An escaping user function (`iter,p,f`) is now represented as a closure record (`I,p,f`), with its 0^{th} field being the machine code pointer (`iter,p0,f0`). An escaping function call is implemented as first selecting the 0^{th} field, placing the closure itself in a special register (the first formal parameter), and then doing a "jump with arguments" (lines 15-16, 18-19).

4.4 Closure conversion

In this section, we present our new *closure conversion* algorithm using the framework defined in Section 4.3. Our algorithm takes a CPS expression E as the argument, determines the closure representation for each function definition in E, and then converts E into a CLO expression E' where each function definition is closed. The presentation of our algorithm is organized in five steps:

1. Construct an *extended* CPS *call graph* that captures the control flow information in the CPS expression.

2. Gather the set of *raw free variables* and their lifetime information for each CPS function.

3. Use *closure strategy analysis* to determine where in the machine to allocate each closure.

4. Use *closure representation analysis* to determine the actual structure of each closure at runtime.

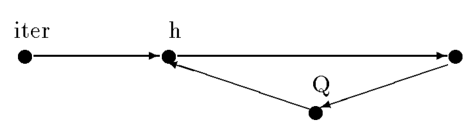
5. Find out the variable access path for all non-local variables of each CPS function.

Each step here does not necessarily correspond to a separate pass in the real implementation, for example, steps 3 through 5 are actually done in a single pass.

4.4.1 Extended CPS call graph

Given a CPS expression E, we can divide the set of function definitions in E into four categories: *escaping user functions*, *known user functions*, *escaping continuation functions*, and *known continuation functions* (see the last paragraph of Section 4.3.1 for definitions). Given two CPS variables v and w, v *directly calls* w if w is possibly the first function call inside the function definition of v. For example, in Figure 18, `J` directly calls `C` and `f` but not `h`, because `h` cannot be the first call inside `J`.

The *extended* CPS *call graph* G of E is a directed graph with the set of function definition variables in E as nodes; there is an edge from v to w in G if v directly calls w, or v directly calls some function with w as its return continuation. For example, the *extended* CPS *call graph* for the function `iter` in Figure 18 is as follows:



Although `J` is not directly called by `h`, we conservatively assume that the function `p` will always call its return continuation, i.e., `J`.

The extended CPS call graph G of E essentially captures a very simple set[2] of control flow information in E. Cycles in the graph imply loops or recursions (e.g., the path from `h` to `J` to `Q`). The nested hierarchies of loops and recursions in E can be revealed by running the Tarjan interval analysis algorithm [Tar74, RP86] on G, assuming G is a reducible flow graph. Given a flow graph G, a Tarjan interval is essentially a single-entry, strongly connected subgraph of G; the interval analysis [RP86] partitions the set of nodes in G into disjoint intervals, with each interval representing a proper loop (or recursion) layer.

For the purpose of our closure analysis, this control flow information is used to choose closure representations that allow more efficient variable accesses in frequently executed program fragments (e.g., loops).

For every function definition v in E, we define its *loop level* $L(v)$ as the nesting depth of its interval in the extended CPS call graph for E, with the outmost interval at depth 0. For variables that are not actually defined in E (e.g., `C,f,p` in Figure 18), their *loop levels* are defined as 0. The *loop level* of each call from v to w is defined as $L(v,w) = min(L(v), L(w))$. The *loop level* for an arbitrary CPS expression inside a function definition v is inductively defined as follows:

- $L(\texttt{if}\ V\ \texttt{then}\ E_1\ \texttt{else}\ E_2) = max(L(E_1), L(E_2))$;
- $L(\texttt{let}\ D\ \texttt{in}\ E_1\ \texttt{end}) = L(E_1)$;
- $L(A_0(A_1,A_2,\ldots,A_n)) = L(v, A_0)$ if A_0 is a continuation, and $max(L(v, A_0), L(v, A_1))$ if A_0 is a user function and A_1 is its return continuation.

The loop level number can be used as a guide for static branch prediction of control flow in E. For example, in function `iter`, the loop level of `h,J,Q` is 1, and the loop level of `iter,C,f,p` is 0. In the CPS expression "`if z then ... else ...`" in `J`'s definition, `J` either calls `f` with the return continuation `Q`, or it calls the continuation variable `C`. Clearly, the call to `f` and `Q` is inside a loop because $L(\texttt{J},\texttt{Q}) = 1$, while the call to `C` is not, because $L(\texttt{J},\texttt{C}) = 0$. The closure representations for `J` and `h` should be biased towards the "`else`" branch, because it is more likely to be taken at runtime.

For each function definition w in the expression E, we also define *pred*(w) as its predecessor set; i.e., the set of all variables v such that there is an edge from v to w in E's extended CPS call graph.

[2]Shivers [Shi91] presents more sophisticated techniques that can find even better approximations of control flow information.

Table 10: Raw free variables and closure strategies

Function	Stage Number	Raw Free Variables	Closure Strategy
`iter`	1	$\emptyset$	1 slot
`h`	2	$\{(\mathtt{p}, 2, 2), (\mathtt{C}, 3, 3), (\mathtt{f}, 3, 3), (\mathtt{h}, 4, 4)\}$	2 slots
`J`	3	$\{(\mathtt{C}, 3, 3), (\mathtt{f}, 3, 3), (\mathtt{a}, 3, 4), (\mathtt{h}, 4, 4)\}$	3 slots
`Q`	4	$\{(\mathtt{a}, 4, 4), (\mathtt{h}, 4, 4)\}$	3 slots

4.4.2 Raw free variables with lifetime

To implement the *safely linked closures* described in Section 4.2, we want to group variables into closure records if they have similar lifetimes. If v is defined much later than w, x, y, then we may not have enough registers to hold w, x, y while waiting for v. If y's last use is much earlier than w's or x's, then the record (w, x, y) might not obey the SSC rule.

Most closure conversion algorithms [App92, Kra87, Ste78] start with a phase to gather the set of raw free variables for each function definition in E. These free variables are called *raw free variables* because some of them may be substituted by a set of other free variables later during the closure representation analysis phase; we use the term *true free variables* to denote the set of variables that are finally put in the closure environment.

Our algorithm gathers the raw free variables together with their *lifetime* information. To define the lifetime for a variable, we first assign a *stage number* (denoted as *SN*) for each function definition w using the following method:

- if w is the outmost function definition, then $SN(w) = 1$;
- if w is a user function, then $SN(w) = 1 + SN(f)$ where f is the nearest enclosing function definition;
- if w is a continuation function, then $SN(w) = 1 + max\{\ SN(v) \mid v \in pred(w)\ \}$; this definition is valid because continuation functions are never recursive.

Intuitively, the stage number is meant to characterize the temporal relation among different CPS functions. The definition of *SN* is based on the observation that each user function is often called after its enclosing function is called. In addition, we also make sure (by using the call graph) that within each particular user function, continuations that have higher stage numbers are always called later than those with lower stage numbers.

With the stage number, we can define the *use time* for each use of every CPS variable v as $SN(f)$ where f is the nearest enclosing function definition for this use of v. The set of free variables for each function definition f is now a set of triples (v, fut, lut) where v is the variable, *fut* is the *first use time* of v denoting the smallest stage number of all uses of v inside f, and *lut* is the *last use time* of v denoting the largest stage number of all uses of v inside f.

To reflect the control flow, the *lut* and *fut* numbers of v can also be calculated based on the (predicted) execution frequency of each use of v. For example, for a CPS expression `if` V `then` E_1 `else` E_2, we can ignore all uses of v in E_1 (or E_2) if $L(E_1) > L(E_2)$ (or $L(E_2) > L(E_1)$) during the calculation. The higher preference for those uses inside a loop body would likely lead to more efficient closure representation at runtime.

For example, the stage number and the set of raw free variables for all function definitions in Figure 18 are shown in Table 10. Notice that a variable can have different *lut* and *fut* numbers inside different function definitions (e.g., `a` in `J` and `Q`).

4.4.3 Closure strategy analysis

Closure strategy analysis essentially determines where in the machine to allocate each closure. Unlike previous CPS compilers [Kra87, Ste78], we do not do any *escape analysis*,[3] because we simply do not use a runtime stack. Our *closure strategy analysis* only decides how many *slots* (i.e., registers) each closure is going to use, denoted by $S(f)$ for each function f. We calculate $S(f)$ using the following simple algorithm (see Figure 20 for the pseduo code):

If f is an escaping user function, then $S(f) = 1$. This essentially means that all its free variables must be put in the heap. The closure for f is a pointer to a linked data structure in the heap.

If f is an escaping continuation function, then $S(f) = k$ where k is the number of callee-save registers. Because their call sites are not known at compile time, most continuation functions have to use the uniform convention; that is, always use k callee-save registers [AS92]. In special cases, some escaping continuation functions can be represented differently; this is discussed in Section 4.5.3.

For known functions, since their call sites are known at compile time, their closures (or environments) may be allocated completely in registers. The number of registers on the target machine can be limited, however, and it may not always be desirable to allocate all free variables in registers (see Section 4.5.2). We run the following iterative algorithm to calculate the appropriate number of slots (registers) used for each known function:

[3] The *escape analysis* here refers to the analysis that decides whether a function's environment can be allocated on the stack or not.

Notations:

$free(v)$ is the set of raw free variables of v

$T(v, f) \quad = \quad max(1, \;\; S(v) \;\; - \;\; |\, free(v) \setminus free(f)\, |)$

Algorithm:

all $S(f)$ are initialized to m where m is the number of target machine registers;

```
repeat
  foreach f do
    V  is the set of variables in pred(f) that do not enclose f's definition;
    foreach v ∈ V do
      S(f)  =   min(T(v, f), S(f));
    end
  end
until (all S(f) reach a fixpoint)
```

Figure 20: Closure strategy analysis for known functions

K1 Initially, each known function f is assigned m slots, that is, $S(f) = m$, where m is the maximum number of available registers on the target machine minus the number of formal parameters of function f (assuming they will be passed in registers);

K2 Then, for each known function f, we substitute $S(f)$ by $min(\{T(v_1, f), \ldots, T(v_n, f), S(f)\})$. Here $v_1, ..., v_n$ are the subset of the functions in $pred(f)$ that do not enclose f's definition, that is, f must be free in these $v_1, ..., v_n$. The value $T(v, f)$ is $max(1, S(v) - j)$, where j is the number of variables that are free in v but not in f. This substitution process is then repeated until $S(f)$ no longer changes and a fixed point[4] is reached.

Here step **K2** is based on the observation that if f is called inside a function v, and f is also free in v, then the number of slots assigned to f should not be bigger than the number of slots available for v's environment, otherwise, some kind of spilling will be inevitable.

When choosing which subset of v_i to use in calculating $S(f)$ at step **K2**, we again take advantage of the control flow information in the extended CPS call graph. More specifically, we want to favor those program fragments that are likely executed more often than others, so we always choose those v_i that have a higher $L(v_i, f)$ value (i.e., the call from v_i to f is within a loop).

Let us apply this algorithm to the function `iter` in Figure 18. Suppose we use 3 callee-save registers, then both $S($`Q`$)$ and $S($`J`$)$ are 3; $S($`h`$)$ is initially 14. Assuming that there are

[4]This iterative process clearly terminates because $T(v, f) \geq 1$, $S(f) \geq 1$, and the sum of $S(f)$ (for all functions) gets smaller in each round.

16 available registers on the target machine; then since `Q` calls `h`, and `a` is free in `Q` but not in `h`, $S(\mathtt{h})$ should be $min(3-1, 14)$, which is 2, as shown in Table 10. Notice that the call from `iter` to `h` is not considered here because `h` is not free in `iter`.

4.4.4 Closure representation analysis

Closure representation analysis solves the following problem: "Given a function f, if f contains m free variables and is assigned n slots, how to place these m values into n slots?"

Given a CPS expression E, the closure representation analysis is done by processing each function definition through a preorder traversal of E; during the traversal, we maintain and update the following three data structures:

whatMap A static environment that maps every function definition processed so far to its closure representation.

whereMap A list of currently visible closures and variables.

baseRegs The current contents of callee-save registers.

When traversing and processing each function definition f, we do the following:

1. Suppose the set of *raw free variables* of f found in the last step (i.e., Section 4.4.2) is RFV. If f is recursive or mutually recursive with some other function, and then we compute the transitive closure RFV^* of f's raw free variables. For example, as shown in Table 10, function `h` is recursive and its RFV is $\{(\mathtt{p},2,2),(\mathtt{C},3,3),(\mathtt{f},3,3),(\mathtt{h},4,4)\}$; we remove `h` and replace it by its raw free variables. We also propagate `h`'s *fut* and *lut* numbers into each of its free variables by taking the minimum of their *fut* numbers and the maximum of their *lut* numbers. As the result, the transitive closure RFV^* of h is $\{(\mathtt{p},2,4),(\mathtt{C},3,4),(\mathtt{f},3,4)\}$.

2. Next, we find the set of *true free variables* (TFV) of f by replacing each continuation variable in RFV^* by its corresponding callee-save variables, and each function definition by its closure contents (or slot variables). For example, suppose we use three callee-save registers, each continuation variable `C` is then represented by a code pointer `C0` and its three callee-save variables `C1,C2,C3`. The set of true free variables TFV for `h` is $\{(\mathtt{p},2,4)$, $(\mathtt{C0},3,4)$, $(\mathtt{C1},3,4)$, $(\mathtt{C2},3,4)$, $(\mathtt{C3},3,4)$, $(\mathtt{f},3,4)\}$. Notice that `C0,C1,C2,C3` here naturally inherit `C`'s *fut* and *lut* numbers.

3. Now assume that TFV of f contains m variables, and f is assigned n slots by closure strategy analysis in Section 4.4.3. If $m = n$, then we are done. If $m < n$, f is

assigned more slots than its number of free variables,[5], the unused slots are just filled with integer zeros. If $m > n$, we search through the current list of visible closures maintained in the **whereMap** data structure, and see if there is any closure record that we can reuse (or share). The SSC rule mentioned in Section 4.1 is satisfied by making sure that we only reuse those closures whose contents are a subset of TFV. Because all closures in the heap are *safely linked closures*, certain closure sharings had already been anticipated while processing the enclosing function definitions.[6] If there are multiple sharable closures, we use a "best fit" heuristic to decide which one to reuse. In the example of function `iter`, the closure `CR` (line 21 in Figure 19) is sharable by the continuations `J` and `Q`.

4. If the size m of TFV after closure sharing is still larger than n, we have to heap allocate part of the closure. We do this by putting $n-1$ variables into individual slots, and packing the remaining $m-n+1$ variables into the heap closure. The criteria in choosing these $n-1$ variables is based on the following priorities: the first priority is smaller *lut* number (i.e., variables that die earlier); the second is smaller *fut* number (i.e., variables that are referenced earlier); the third is whether the variable is already in the current callee-save registers (i.e., **baseRegs**) or not. We also use the contents of **baseRegs** to decide which variable goes to which slot to reduce register moves. For example, the function `h` is assigned 2 slots but `h` has 6 true free variables, we put the free variable `p` in the register because it has the smallest *fut* number (all variables have the same *lut* number).

5. Finally, we decide the actual layout of the spilled heap closure of the above $m-n+1$ variables based on each variable's *lut* number. To satisfy SSC with shared closures, each distinct *lut* number requires a separate record. For example, the closure for G in Figure 9 was split into two records because `v`'s *lut* number was different from those of `w,x,y,z`.

We finish processing the function definition f by updating the **whatMap**, **whereMap** and **baseRegs** environments based on f's closure representation.

Not only is `CR` shared in Figure 19, but its creation is outside the `h` loop. Thus, each iteration of `h` manages to call two unknown (escaping) functions *without any memory traffic!* This is one of the most important strengths of our new algorithm.

[5]This case is only possible for escaping continuation functions.

[6]More specifically, suppose g is the function that encloses f, the safely linked closure built for g must contain separate chunks for those free variables of g that are also free in f; these chunks are very likely shared by f later.

4.4.5 Access path for non-local free variables

Computing the access path for each non-local free variable v is done by a breadth-first search of v in the **whereMap** environment. We use the "lazy display" technique [Kra87] to keep a cache of access paths, so that loads of common paths can be shared. More specifically, let us look at the function `i` (the innermost function inside `f`) in Figure 15: assuming that `i` uses the safely linked closure shown in Figure 9, then accessing each non-local variable (e.g., `w,x,y,z`) inside `i` requires traversing two links; but we can first load the 2nd field of the closure `I` into a register r, and then access `w`, `x`, `y`, and `z` directly from r via one load. These intermediate variables (e.g., register r) may use up all the available machine registers and cause unnecessary register spilling, but this can always be avoided by selectively keeping limited number of intermediate variables in the "lazy display" (registers).

4.4.6 Remarks

Graph-coloring global register allocation and targeting [Cha82, BCKT89], which have been implemented for SML/NJ by Lal George *et al* [GGR94], accomplishes most control transfers (function calls) (such as line 12 and 13 in Figure 19) without any register-register moves. This allows a more flexible boundary between callee-save and caller-save registers than is normal in most compilers.

Programs, in our scheme, tend to accumulate values in registers and only dump them into a closure at infrequent intervals. It may be useful to use more callee-save (and fewer caller-save) registers to optimize this (e.g., to reduce total heap allocation)

Our closure scheme handles tail calls very nicely, simply by re-arranging registers. Hanson [Han90] shows how complicated things become when it's necessary to re-arrange a stack frame.

A source-language function that calls several other functions in sequence would, in previous CPS compilers (including our own), allocate a continuation closure for each call. The callee-save registers and safely linked closures allow us to allocate a simple shared closure.

General deep recursions are handled very efficiently in our scheme. A conventional stack implementation tends to have a high space overhead per frame, because each frame contains all variables that are live in the current scope (this violates the SSC rule, see Section 4.2). Our safely linked closures—which only contains variables that are actual live—are quite concise. Thus, memory usage from deep recursion will be much less.

4.5 Case studies

A good environment allocation scheme must implement frequently used control structures very efficiently. Many compilers identify special control structures at compile time, and assign each of them a special closure allocation strategy. For example, in Kranz's Orbit compiler [Kra87], all tail recursions are assigned a so-called "*stack/loop*" strategy, and all general recursions are assigned a "*stack/recursion*" strategy. Our new closure conversion algorithm, on the other hand, uniformly decides the closure strategy (i.e., number of slots) and the closure representation for each function solely based on the lifetime information of its free variables and simple control flow information.

In Section 4.3, we described how our algorithm implements tail recursion very efficiently (i.e., function `iter`). In this section, we use several more examples to show how our new algorithm effectively deals with other common control structures such as a sequence of function applications, calling a known function, and general recursion.

4.5.1 Function calls in sequence

One common control structure in functional programs is making a sequence of function applications, as shown in the following example:

```
fun f(g,u,v,w) =
  let val x = g(u,v)
      val y = g(x,w)
      val z = g(y,x)
   in x+y+z+v+1
  end
```

Here the function `g` (a formal parameter of `f`) is called three times in a row inside the function `f`. Under the traditional stack scheme, when function `f` is called, an activation record for `f`—containing formal parameters (i.e., `g,u,v,w`) and local variables (i.e., `x,y,z`)—will be pushed onto the stack. Each time before `g` is called, certain local variables in registers must be saved onto the stack. For example, assuming all function arguments (i.e., `g,u,v,w`) and return results (i.e., `x,y,z`) are passed in registers, then before the first call to `g`, the registers holding `g` and `w` must be saved so that they can still be retrieved later after `g` returns.

If activation records are allocated on the heap, things get much worse. Every time registers need to be saved before a function call, a closure record has to be built on the heap. Because heap allocated closures are not contiguous in memory, an extra memory write (and later a memory read) of the frame pointer is necessary at each function call.

```
fun f(C,g,u,v,w) =
  let fun J(x) =
        let fun K(y) =
              let fun Q(z) =
                    let val r = x+y+z+v+1
                     in C(r)
                    end
               in g(Q,y,x)
              end
         in g(K,x,w)
        end
   in g(J,u,v)
  end
```

Figure 21: Function f in CPS

Table 11: Raw free variables and closure strategies for function f

Function	Stage Number	Raw Free Variables	Closure Strategy
f	1	$\emptyset$	1 slot
J	2	$\{(\mathtt{C},4,4),(\mathtt{v},4,4),(\mathtt{g},2,3),(\mathtt{w},2,2)\}$	3 slots
K	3	$\{(\mathtt{C},4,4),(\mathtt{v},4,4),(\mathtt{g},3,3),(\mathtt{x},3,4)\}$	3 slots
Q	4	$\{(\mathtt{C},4,4),(\mathtt{v},4,4),(\mathtt{y},4,4),(\mathtt{x},4,4)\}$	3 slots

With our new closure analysis technique to make good use of callee-save registers, heap-allocated activation records can be made almost as efficient as stack allocation (see Chapter 5). The idea is that we can allocate most parts of the current activation record in callee-save registers. With careful lifetime analysis, register save/restore around several function calls can often be eliminated or amalgamated, so that function calls in sequence need to allocate only one heap record.

Figure 21 and 22 list the CPS and CLO code for function `f`. Table 11 lists the stage numbers, raw free variables, and closure strategies for function f and continuations `J`, `K`, and `Q`. During the closure conversion of f (as shown in Figure 22), continuations are still represented as one code pointer plus three callee-save registers, all denoted by capital letters. As before, escaping function calls (i.e., calls to `g` on line 14,17,21) are implemented as first selecting the 0^{th} field, placing the closure itself in a special register (the first formal parameter), and then doing a "jump with arguments" (lines 13-14,16-17,20-21). Before the

```
01 fun f(C0,C1,C2,C3,g,u,v,w) =
02   let fun J0(J1,J2,J3,x) =
03         let fun K0(K1,K2,K3,y) =
04               let fun Q0(Q1,Q2,Q3,z) =
05                     let val v = select(4,Q1)
06                         val r = Q3+Q2+z+v+1
07                         val C0 = select(0,Q1)
08                         val C1 = select(1,Q1)
09                         val C2 = select(2,Q1)
10                         val C3 = select(3,Q1)
11                      in C0(C1,C2,C3,r)
12                     end
13                   val g0 = select(0,K2)
14                in g0(K2,Q0,K1,y,K3,y,K2)
15               end
16             val g0 = select(0,J2)
17          in g0(J2,K0,J1,J2,x,x,J3)
18         end
19       val CR = (C0,C1,C2,C3,v)
20       val g0 = select(0,g)
21    in g0(g,J0,CR,g,w,u,v)
22   end
```

Figure 22: Making a sequence of function calls

first call to `g` (line 21), we put variables that have smaller *lut* numbers (i.e., `g,w`) callee-save registers (i.e., `J2,J3`), and spill the rest (i.e., `C0-C3,v`) into a heap record `CR` (line 19). At the second and the third calls to `g` (line 17,14), no register save/restore are necessary. This is because the lifetime of `w` and `x` (also `g` and `y`) does not overlap, so they can just share one callee-save register (i.e., `J3` and `K3`, `K2` and `Q2`).

4.5.2 Lambda lifting on known functions

Lambda lifting [Joh85] is a well-known transformation that rewrites a program into an equivalent one in which no function has free variables. Lambda lifting on known functions essentially corresponds to the special closure allocation strategy that allocates as many free variables in registers as possible. But this special strategy does not always generate efficient code [Kra87]. For example, in the following program, assume that `f` is a known function, and `p,w,x,y`, and `z` are its free variables.

```
fun f u = (p u, u+w+x+y+z+1)

fun g(x,y) = (p x, f x, f y)
```

If the closure for `f` is allocated in registers, then before the call to `p` inside `g`, some of `f`'s free variables must be saved in the heap (assuming there are only three callee-save registers). When the call to `p` returns, these variables must be reloaded back into registers, and passed to function `f`; after entering `f`, some of them again have to be saved when `f` calls `p`, and so on. Clearly, allocating `f`'s environment in registers dramatically increases the need for more callee-save registers inside `g`. This leads to more memory traffic when there are only a limited number of callee-save registers.

The *closure strategy analysis* described in Section 4.4.3 uses an iterative algorithm to decide the number of registers assigned to each known function. The number of registers assigned to `f` will be restricted by those of its callers, that is, the return continuation for `p x` and the return continuation for the first call to `f`. As a result, `f` is only assigned one slot, and its closure will be allocated in the heap.

4.5.3 General recursion

The *closure strategy analysis* algorithm described in Section 4.4.3 conservatively represents all continuation functions using the same (fixed) number of callee-save registers. In some cases, this restriction can be relaxed: continuations that are passed to known functions can be represented in any number of callee-save registers. This special calling convention is especially desirable for general recursion, such as in the case of the following CPS translation of the `map` function:

```
fun map(C,f,l) =
  let fun m(J,z) =
        if (z=[]) then []
        else let val a = car z
                 val r = cdr z
                 fun K(b) =
                   let fun Q(s) =
                         let val y = b::s
                          in J(y)
                         end
                    in m(Q,r)
                   end
              in f(K,a)
             end
   in m(C,l)
  end
```

Notice that the recursive function `m` is called only at two places: one by function `map` with `C` as the return continuation, one inside `K` with `Q` as the return continuation. Because the second call to `m` is a recursive call, it will be executed much more often than the first. We can represent all normal continuation functions in three callee-save registers, but represent continuations `J` and `Q` in two callee-save registers. Figure 23 lists the code of function `map` after translation into CLO using the above special calling convention.

```
01 fun map(C0,C1,C2,C3,f,l) =
02   let fun R0(R1,R2,x) =
03         let val C0 = select(0,R1)
04             val C1 = select(1,R1)
05             val C3 = select(2,R1)
06          in C0(C1,R2,C3,x)
07         end
08
09       fun m(J0,J1,J2,z,f) =
10         if (z=[]) then J0(J1,J2,[])
11         else (let val a = car z
12                   val r = cdr z
13                   fun K0(K1,K2,K3,b) =
14                     let fun Q0(Q1,Q2,s) =
15                           let val y = Q2::s
16                               val J0 = select(0,Q1)
17                               val J1 = select(1,Q1)
18                               val J2 = select(2,Q1)
19                            in J0(J1,J2,y)
20                           end
21                      in m(Q0,K1,b,K2,K3)
22                     end
23                   val CR = (J0,J1,J2)
24                   val f0 = select(0,f)
25                in f0(f,K0,CR,r,f,a)
26               end)
27       val CC = (C0,C1,C3)
28    in m(R0,CC,C2,l,f)
29   end
```

Figure 23: Function `map` using special calling conventions

Here `m` is a known function, and the environment for `m` (i.e., the free variable `f`) is allocated in a register (i.e., `f` is treated as an extra argument of `m`, see line 9,21,28). Since continuation `C` still uses the normal calling convention, when it is passed to the function `m` (line 28), a new "coercion" continuation (i.e., `R0` on line 2-7) has to be built to adjust the normal convention (three callee-save registers `C0-C3`) into the special convention (two callee-save registers `R0-R2`). Because the return continuation `J` of `m` is represented in two callee-save registers (i.e., `J0-J2`), we can build a smaller heap closure (of size 3, on line 23) for continuation `K`.

If both `J` and `Q` are represented in three callee-save registers, the heap closure for `K` would at least be of size 4.

4.6 Measurements

We have implemented our "space-efficient" closure conversion algorithm in the Standard ML of New Jersey compiler version 1.03z. In order to find out how much performance gain we can get from our new closure conversion algorithm, we have measured the performance of six different compilers on ten SML benchmarks (see Table 2 in Section 2.4).

The six compilers we use are all simple variations of the SML/NJ compiler version 1.03z. All six compilers satisfy the "safe for space complexity" rule, and all use the type-directed compilation techniques described in Chapter 3 to allow arguments being passed in registers and to support more efficient data representations. The "lazy display" technique is implemented in all six compilers, but it is used more effectively in compilers that use the new closure conversion algorithm, because of their more extensive use of shared closures.

sml.occ This version uses the old closure conversion algorithm [App92, AS92]. More specifically, it uses the linked closure representation if it is space safe, otherwise it uses the flat closure representation. Continuation closures are represented using three callee-save registers.

sml.gp1,sml.gp2,sml.gp3,sml.gp4 These compilers all use the new closure conversion algorithm described in this chapter. They respectively use one, two, three, four (general-purpose) callee-save registers to represent continuation closures. The **sml.gp3** compiler is exactly same as the **sml.ffb** compiler used in Chapter 3.

sml.fp3 This compiler uses the new closure conversion algorithm described in this chapter. Continuation closures are represented using three general-purpose callee-save registers and three floating-point callee-save registers.

All measurements are done on a DEC5000/240 workstation with 128 mega-bytes of memory, using the methodology described in Section 2.4. In Figure 24 and Table 12, we illustrate and list the execution time of running the benchmarks using the above six compilers. Only the execution time (user time plus garbage collection time plus system time, in seconds) for the **sml.occ** compiler is shown; the performance for all other compilers

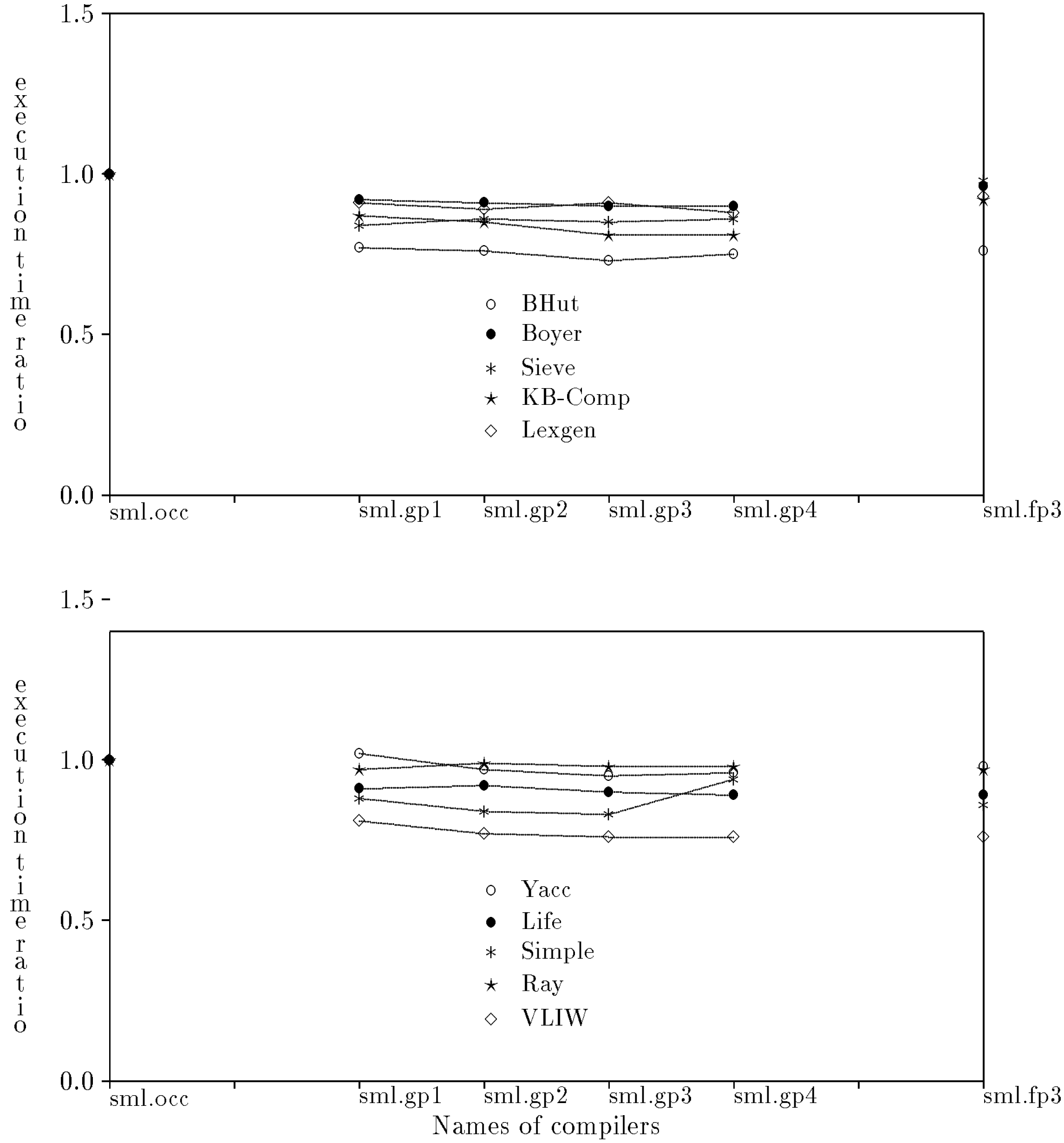


Figure 24: A comparison of execution time (illustration)

are denoted by a relative ratio to the **sml.occ** compiler. The garbage collection time (corresponding to the data in Table 12 is listed separately in Table 13. Similarly, 14, 15, and 16 respectively compare the heap allocation (in mega-bytes), the compilation time (in seconds), and the code size (in kilo-bytes).

Table 12: A comparison of execution time

Program	**Basis** (seconds)	**sml.occ** (ratio)	**sml.gp1** (ratio)	**sml.gp2** (ratio)	**sml.gp3** (ratio)	**sml.gp4** (ratio)	**sml.fp3** (ratio)
BHut	30.5	1.00	0.77	0.76	0.73	0.75	0.76
Boyer	2.5	1.00	0.92	0.91	0.90	0.90	0.96
Sieve	35.6	1.00	0.84	0.86	0.85	0.86	0.98
KB-Comp	7.5	1.00	0.87	0.85	0.81	0.81	0.92
Lexgen	11.5	1.00	0.91	0.89	0.91	0.88	0.93
Yacc	4.4	1.00	1.02	0.97	0.95	0.96	0.98
Life	1.3	1.00	0.91	0.92	0.90	0.89	0.89
Simple	22.2	1.00	0.88	0.84	0.83	0.94	0.86
Ray	20.3	1.00	0.97	0.99	0.98	0.98	0.97
VLIW	16.1	1.00	0.81	0.77	0.76	0.76	0.76
Average		1.00	0.89	0.88	0.86	0.87	0.90

Table 13: A comparison of garbage collection time

Program	**Basis** (seconds)	**sml.occ** (ratio)	**sml.gp1** (ratio)	**sml.gp2** (ratio)	**sml.gp3** (ratio)	**sml.gp4** (ratio)	**sml.fp3** (ratio)
BHut	1.61	1.00	0.81	0.85	0.78	0.80	0.83
Boyer	1.03	1.00	0.98	0.97	0.93	0.93	1.01
Sieve	17.9	1.00	0.70	0.72	0.72	0.73	0.85
KB-Comp	0.94	1.00	0.98	0.98	0.98	0.99	0.99
Lexgen	0.83	1.00	0.90	0.89	0.83	0.88	0.94
Yacc	1.05	1.00	1.12	1.02	1.04	1.04	1.07
Life	0.02	1.00	1.00	1.50	0.00	1.00	1.00
Simple	3.43	1.00	0.90	0.89	0.90	0.91	0.90
Ray	0.05	1.00	1.80	1.60	2.20	2.00	1.80
VLIW	0.53	1.00	0.94	0.83	0.96	0.94	1.06

In Table 17, we have also listed the number of memory fetches (in millions) for local/global variables and the allocation profile of various kinds of closures for the **sml.occ** and **sml.gp3** compilers. Here, "escape", "known", and "cont" are respectively the total size of closures (in mega-words) allocated for escaping user functions, known user functions, and continuation functions; heap allocation for other non-closures is not listed in Table 17 (but is included in Table 14).

We can draw the following conclusions from these comparisons:

- The **sml.gp3** compiler has the exactly same setup as the **sml.occ** compiler except that one uses the new closure conversion algorithm, and the other uses the old algorithm. On average, the **sml.gp3** compiler reduces heap allocation by 32% (closure allocation

Table 14: A comparison of total heap allocation

Program	**Basis** (Mbytes)	**sml.occ** (ratio)	**sml.gp1** (ratio)	**sml.gp2** (ratio)	**sml.gp3** (ratio)	**sml.gp4** (ratio)	**sml.fp3** (ratio)
BHut	619.0	1.00	0.55	0.50	0.44	0.44	0.45
Boyer	30.8	1.00	0.86	0.80	0.75	0.73	0.88
Sieve	253.9	1.00	0.72	0.72	0.70	0.70	0.83
KB-Comp	156.7	1.00	0.79	0.70	0.59	0.57	0.79
Lexgen	96.9	1.00	0.78	0.71	0.64	0.49	0.69
Yacc	57.5	1.00	0.89	0.81	0.77	0.75	0.78
Life	10.2	1.00	0.77	0.73	0.68	0.63	0.68
Simple	323.5	1.00	0.82	0.68	0.64	0.70	0.57
Ray	331.6	1.00	0.92	0.96	0.93	0.92	0.93
VLIW	160.8	1.00	0.80	0.75	0.70	0.71	0.71
Average		1.00	0.79	0.74	0.68	0.66	0.73

Table 15: A comparison of compilation time

Program	**Basis** (seconds)	**sml.occ** (ratio)	**sml.gp1** (ratio)	**sml.gp2** (ratio)	**sml.gp3** (ratio)	**sml.gp4** (ratio)	**sml.fp3** (ratio)
BHut	59.1	1.00	0.99	0.92	0.96	0.95	1.01
Boyer	26.4	1.00	0.99	0.96	0.96	0.98	1.01
Sieve	31.8	1.00	0.99	0.98	0.97	0.99	1.02
KB-Comp	19.4	1.00	1.03	0.97	0.98	1.01	1.03
Lexgen	37.8	1.00	0.91	0.92	0.86	0.89	0.99
Yacc	132.2	1.00	0.98	0.93	0.95	0.95	1.01
Life	4.8	1.00	1.04	1.01	0.98	1.02	1.04
Simple	64.6	1.00	0.85	0.83	0.82	0.78	0.86
Ray	15.5	1.00	1.01	0.98	0.97	0.98	1.06
VLIW	185.9	1.00	0.98	0.90	0.91	0.97	0.98
Average		1.00	0.98	0.94	0.94	0.95	1.00

by 40%) and memory fetches for local/global variables by 46%; and improves the already efficient code generated by the **sml.occ** compiler by 14%. The **sml.gp3** compiler also uniformly generates more compact code, achieving an average of 19% reduction in code size over the **sml.occ** compiler. **BHut** and **VLIW** achieve up to respectively 27% and 24% speedup in execution time, because they get significant benefits from using safely linked closures.

- Varying the number of callee-save registers under the new closure conversion algorithm has little effect on the execution time (within 6% range, 3% on average), but has a

Table 16: A comparison of code size

Program	**Basis** (Kbytes)	**sml.occ** (ratio)	**sml.gp1** (ratio)	**sml.gp2** (ratio)	**sml.gp3** (ratio)	**sml.gp4** (ratio)	**sml.fp3** (ratio)
BHut	103.1	1.00	0.83	0.81	0.81	0.76	0.83
Boyer	107.2	1.00	0.93	0.92	0.92	0.93	0.92
Sieve	74.1	1.00	0.85	0.83	0.83	0.84	0.87
KB-Comp	47.1	1.00	0.91	0.86	0.86	0.88	0.88
Lexgen	105.5	1.00	0.83	0.80	0.79	0.79	0.80
Yacc	418.2	1.00	0.81	0.76	0.75	0.75	0.75
Life	14.5	1.00	0.91	0.88	0.88	0.87	0.88
Simple	183.1	1.00	0.64	0.61	0.59	0.56	0.58
Ray	53.5	1.00	0.87	0.84	0.85	0.85	0.87
VLIW	426.9	1.00	0.82	0.78	0.78	0.78	0.78
Average		1.00	0.84	0.81	0.81	0.80	0.82

Table 17: Breakdown of closure access and allocation

	Closure Access (mem-reads in millions)			**Closure Allocation** (escape+known+cont in mega-words)		
Program	**sml.occ**	**sml.gp3**	**saving**	**sml.occ**	**sml.gp3**	**saving**
BHut	87.57	37.08	57.66%	0.22+0.18+76.0	0.12+1.59+36.6	49.91%
Boyer	4.58	2.58	43.64%	1.91+4.88+0.95	1.18+0.48+3.18	37.56%
Sieve	47.20	27.49	41.76%	7.28+11.4+26.5	7.28+8.29+9.84	43.71%
KB-Comp	16.00	13.01	18.71%	12.5+0.04+24.2	5.99+1.07+12.9	45.57%
Lexgen	15.89	7.57	52.38%	1.47+0.18+17.1	0.48+0.67+6.90	56.92%
Yacc	9.75	4.21	56.78%	0.10+2.71+8.03	0.09+2.07+5.33	30.89%
Life	1.57	0.66	57.97%	0.06+0.00+1.50	0.06+0.06+0.73	46.05%
Simple	76.47	45.78	40.14%	3.47+0.55+62.4	2.64+4.13+35.5	36.45%
Ray	20.20	13.64	32.49%	0.00+0.01+15.2	0.00+2.11+11.5	10.35%
VLIW	40.65	16.79	58.70%	7.38+2.60+33.5	6.78+3.07+15.8	40.89%
Average			46.02%			39.83%

large effect on the total heap allocation. The **sml.gp3** compiler is about 3-4% faster than the **sml.gp1** compiler, but its total heap allocation is more than 11% less.

- The effect of the new closure algorithm on the garbage collection time (g.c. time, see Table 13) varies dramatically depending on the benchmarks. Eight of the ten benchmarks we measured spend less time in garbage collection (because of less heap allocation), however, the g.c. time for **Ray** is almost doubled. This is not surprising since the new closure algorithm has complete different allocation behavior from the old algorithm. Having more callee-save registers (especially the **sml.fp3** compiler) generally increases the g.c. time, which sort of reflects the heap allocation data in Table 14.

- From the allocation profiling data in Table 14, we can see that most of the reduction in heap allocation is from the continuation closures; closure analysis has almost no effect on non-closures.

- The new closure conversion algorithm surprisingly improves the compilation time by nearly 6%. This is probably because that the old algorithm used in **sml.occ** compiler contains expensive ad-hoc heuristics, while the new algorithm is much more systematic. Another reason might be that the new algorithm generates less code, thus requires less instruction scheduling time.

- Using three floating-point callee-save registers (i.e., the **sml.fp3** compiler) does not achieve any better performance than using no floating-point callee-save registers. The slowdown mostly comes from benchmarks such as **Sieve** and **KB-Comp** that frequently use first-class continuations and exception handlers. First-class continuations and exceptions may be put into a record or stored into some reference cell, so they must be representable in just one word, not as k separate callee-save registers; when a continuation is captured, the k-register representation has to be packaged into a single word by making a record on the heap; when a continuation is triggered (by `throw`), the single-word representation must be unpackaged into k callee-save registers. This overhead is higher if more callee-save registers are used.

4.7 Related work

Guy Steele's Rabbit compiler [Ste78] is the first compiler that uses the continuation-passing style as the intermediate language; it is also the first one that represents the "stack frames" using continuation closures. Rozas's Lias compiler [Roz84] used closure analysis to choose specialized representations for different kinds of closures; Kranz's Orbit compiler [KKR*86, Kra87] uses six different closure allocation strategies for different kinds of functions; Appel and Jim investigated closure-sharing strategies [AJ88] and proposed many alternative closure representations. Unfortunately, all these closure analysis techniques violate the "safe for space complexity" rule due to unsafe closure sharing. The closure conversion algorithm described in this chapter combines all of these analyses (except stack allocation) and more, while still satisfying the "safe for space complexity" rule.

Hanson [Han90] showed the complexity of implementing tail calls correctly and efficiently on a conventional stack. In our heap-based scheme, the correctness is straightforward because dead frames are automatically reclaimed by the garbage collector; the efficiency is achieved by using the loop-header technique [App94b] to hoist the loop-invariant free variables out of the tail recursion, and using the callee-save registers [AS92] to simulate the top reusable stack frames.

Some compilers [Ste78, KKR*86, Car84a] perform closure conversion and closure analyses as part of their translation from lambda calculus or continuation-passing style into machine code. But it is useful to separate the closure introduction from machine code generation so that the compiler is more modular; this has been done in compilers based on ordinary λ-calculus (through lambda lifting) [CCM85, Joh85] and on continuation-passing style (using closure-passing style) [AJ89, KH89].

Many have tried to make call/cc efficient, but this is very hard to achieve in traditional stack-based schemes. With an ordinary contiguous stack implementation, the entire stack must be copied on each creation or invocation of a first-class continuation. Clinger [CHO88b] and Hieb [HDB90] presented several mixed stack/heap strategies intended to support call/cc efficiently in the presence of stacks. Their basic idea is to make a "stack chunk" that holds several stack frames; if this fills, it is linked to another chunk allocated from the heap. This turns to be complicated to implement. Danvy [Dan87] proposed to use a free list of re-usable frames (or "quasi-stack") to support fast call/cc; but his method may incur extra overhead at each function call (or return).

Both Chow [Cho88a] and Steele [SS80] observed that dataflow analysis can help decide whether to put variables in caller-save or callee-save registers. We are the first to show how to represent callee-save registers in continuation-passing style [AS92, App92] and how to use compile-time variable lifetime information to do a much better job of it.

Local variables of different functions with nonoverlapping live ranges can be allocated to the same register or global without any save/restore [GS91, Cho88a]. We achieve this by allocating part of the closures (for known functions and continuations) in both caller- and callee-save registers, and then using the graph-coloring global register allocation and targeting algorithms [Cha82, BCKT89, GGR94].

4.8 Summary

Our new closure conversion algorithm is a great success. The closure conversion algorithm itself is faster than our previous algorithm (see Table 15). It makes programs smaller (by an average of 19%) and faster (by an average of 14%). It decreases the rate of heap allocation by 32%, and by obeying the "safe for space complexity" rule and keeping closures small, it helps reduce the amount of live data preserved by garbage collection.

The closure analysis technique introduced in this chapter can also be applied to compilers that do not use CPS as their intermediate language. Both safely linked closures and good use of callee-save registers are essential in building compilers that satisfy the "safe for space complexity" rule.

Chapter 5

Heap vs. Stack

It has been proposed that allocating procedure activation records on a garbage collected heap is more efficient than stack allocation. But, previous comparisons of heap vs. stack allocation have been over-simplistic, neglecting, for example, frame pointers, or the better locality of reference of stacks.

In this chapter, we present a comprehensive analysis of all the components of creation, access, and disposal of heap-allocated and stack-allocated activation records. Among our results are:

- Although stack frames are known to have a better cache read-miss rate than heap frames, our simple analytical model (backed up by simulation results) shows that the difference is too trivial to matter.

- The cache write-miss rate of heap frames is very high; we show that a variety of miss-handling strategies (exemplified by specific modern machines) can give good performance, but not all can.

- The write-miss policy of the primary cache is much more important than the write-miss policy of the secondary cache.

- Stacks restrict the flexibility of closure representations (for higher-order functions) in important (and costly) ways.

- The extra load placed on the garbage collector by heap-allocated frames is very small.

- The demands of modern programming languages make stacks quite complicated to implement efficiently and correctly.

Table 18: Cost breakdown of different frame allocation strategies

Component	Heap	Stack	Stack Chunks (see §5.9)	Quasi-Stack (see §5.9)	see:
Creation	3.1	1.0	3.0	3.0	§5.2
Frame pointers	2.0	0.0	0.0	2.0	§5.3
Copying and sharing	0.0	3.4	3.4	3.4	§5.4
Cache write misses	0.0 or 5.3	0.0	0.0	0.0	§5.6.1
Cache read misses	1.0	0.0	0.0	0.0	§5.6.2
Disposal (pop)	1.4	1.0	1.0	4.0	§5.7
Total Cost	**7.5** or 12.8	**5.4**	**7.4**	**12.4**	
Call/cc	$O(1)$	$O(N)$	$O(X)$	$O(1)$	§5.9
Implementation	easy	hard	hard	hard	§5.10

Overall, the execution cost of stack-allocated and heap-allocated frames is similar; but heap frames are simpler to implement and allow very efficient first-class continuations (call/cc).

5.1 Garbage-collected frames

In a programming language implementation that uses garbage collection, all procedure activation records (frames) can be allocated on the heap. This is convenient for higher-order languages (Scheme, ML, etc.) whose "closures" can have indefinite extent, and it is even more convenient for languages with first-class continuations.

One might think that it would be expensive to allocate, at every procedure call, heap storage that becomes garbage on return. But not necessarily [App87]: modern generational garbage-collection algorithms [Ung86] can reclaim dead frames efficiently, as cheap as the one-instruction cost to pop the stack.

But there are other costs involved in creating, accessing, and destroying activation records—whether on a heap or a stack. In Table 18, the cost for each component of frame creation, access, and disposal for heap-allocated and stack-allocated frames is shown, measured in *instructions per frame* (tail recursions and leaf procedures do not make frames). The numbers for *cache write misses* depend critically on the design of the machine's *primary cache*; we show the cost of two alternatives. The last column has references to the section number (in this chapter) of the explanation of each component. In the last row, N is the stack depth; X is the size of one stack chunk. These costs are explained and analyzed in the remainder of this chapter.

The numbers in Table 18 depend on many assumptions. The most critical assumptions are these:

- The language in question has static scope, higher order functions, and garbage collection. The only question being investigated is whether there is an activation-record stack *in addition* to the garbage collection of other objects.
- The compiler and garbage collector are required to be "safe for space complexity;" that is, statically dead pointers (in the dataflow sense) do not keep objects live. (See Section 2.3.3.)
- There are few side effects in compiled programs, so that generational garbage collection is efficient.

These assumptions, and others, are explained in the rest of this chapter.

Table 18 clearly shows that there are three important criteria in the choice between a stack or heap representation:

1. The write-miss policy of the machine's primary cache (discussed in Section 5.6.1). On machines with *fetch-on-write* or *write-around* write-miss policies, heap-allocated frames are significantly more expensive.
2. Stacks are harder than heaps to implement without space leaks, as explained in Section 5.10.
3. If the programming language supports first class continuation (call/cc [Lan65])—a primitive often used to support multi-threading, exceptions, and so on—stacks have a much higher cost (see Section 5.9).

The (perhaps) startling result is that heap-allocated frames have almost the same cost as stack frames.

Finally, we point out that the absolute differences are small: two instructions per frame is less than 2% of total execution cost, as can be calculated from Figure 20.

We count *instructions* rather than *cycles*. In general, *load* and *store* instructions for frame management can usually be scheduled to avoid stalls (they are rarely in the critical path of a loop, for example). The *branch* instructions for heap-limit testing will be at least 99% predictable—because hundreds of frames are allocated (heap limit not exceeded) between garbage collections (heap limit exceeded); so branches for heap-limit tests will cause almost no stalls. Thus, instruction counts, plus a separate accounting of the cache misses, form a suitable cost model.

Table 19: Shared limit checks

Many frame allocations are in the same (extended) basic block as other non-frame allocations. In these cases the heap limit check would have to be done anyway, and should not be charged to the frame allocation. This table shows the proportion of frame allocations that are *not* in the same block as a non-frame allocation. The results shown are from measurements of ML benchmark programs (see Table 2 in Section 2.4 for details) as compiled by the *Standard ML of New Jersey* [AM91] compiler.

Program	Limit Checks per Frame
Boyer	.717
Knuth-B	.783
Lexgen	.864
Life	.456
YACC	.631
Simple	.665
VLIW	.695
Average	**.687**

5.2 Creation

To allocate a stack frame, the program must add a constant to the stack pointer. This takes one instruction. It is also necessary to check for stack overflow; but since overflow is so rare, this can usually be done at no cost using an inaccessible virtual memory page.

Allocating a heap frame is more complicated:

1. Heap overflow must be checked. As explained by Appel and Li [AL91], and contrary to the ideas of Appel [App89], this should not be done by a virtual memory fault: (1) operating-system fault handling is too expensive, (2) heap overflow is unrelated to locality of reference, and (3) the technique is almost impossible on machines without precise interrupts.

   Thus, a comparison and a conditional branch are required; by keeping the free-space pointer and the limit pointer in registers, this takes about two instructions. However, many of the frame allocations occur in the same extended basic block[1] as other (non-frame) allocations, which would require limit checks anyhow (see Table 19). The actual cost is therefore $2 \cdot 0.687 = 1.374$.

[1] An extended basic block has one entry point, followed by a tree of control flow with several exits.

2. The free-space pointer must be incremented. This costs one instruction. But when the frame allocation is in the same basic block as another allocation, the increment can be shared. So the cost is 0.687 instructions per frame, on the average.

3. A descriptor word must be written to the frame, so the garbage collector can understand it. However, the frame usually contains a return address; the garbage collector can have a mapping of return addresses to descriptors, so frame need not explicitly contain the descriptor.[2]

4. The free-space pointer must be copied to the frame pointer; this takes one *move* instruction.

The total cost is about 3.1 instructions, on the average.

5.3 Frame pointers

When a stack frame is popped, the frame pointer must be set back to the caller's frame. Some implementations of stack frames have put a copy of the (previous) frame pointer in each frame, and this is fetched back upon function return. But for contiguous stack frames of known size, this is clearly unnecessary; the stack pointer itself can be used as the frame pointer, and the pop can just be a subtraction from the stack pointer. This is the common modern practice.

But when frames are not contiguous (e.g., when they are heap-allocated), then each frame must contain a pointer to the caller's frame. One instruction will be necessary to store the (previous) frame pointer into a new frame; and one instruction will be necessary to fetch it back.

Thus, heap-allocated frames have a 2-instruction cost, per frame, for frame pointer manipulation; stack-allocated frames incur no such cost.

Other registers

Efficient heap allocation uses a *free-space pointer* and a *free-space limit* which should be kept in registers.[3] However, the cost of reserving these registers should not be charged to heap allocation of frames, because we are assuming that the implementation in question already

[2] Actually, SML/NJ does write an explicit descriptor to each frame, for simplicity.

[3] Some implementations use a BIBOP (BIg Bag Of Pages [Han80]) scheme that allocates each kind of object in a different contiguous space, so that only one g.c.-descriptor is required per space, instead of per object. This requires a free-space pointer and a limit pointer *per space*.

has garbage collection (presumably with efficient allocation) for other purposes (lists and closures, for example).

5.4 Copying and sharing

A language (such as Scheme, ML, Smalltalk) with higher-order functions needs *closures* to hold the free variables of functions that have been created but not yet called. If one function's free variables overlap with another's, then one closure might point to another (which saves the expense of copying the contents).

So there are two kinds of objects: activation records, whose lifetimes have last-in first-out behavior; and higher-order function closures, which have indefinite extent. The former can be stack allocated (or heap allocated), but the latter must be allocated on a garbage-collected heap. Furthermore, *stack frames may point at heap closures, but heap closures may never point at stack frames*, otherwise there will be dangling pointers.

This means that if the compiler wants to build a closure containing free variables (x, y, z) which are available in a stack frame, all three variables must be copied into the closure; the closure cannot just point to the stack frame.

But if all activation records are heap-allocated, then closures may point at them. This flexibility allows the closure analysis phase of a good compiler to choose much better (smaller, shallower) representations for closures, with more sharing and less copying (see Chapter 4).

The restriction that heaps cannot point to stacks must be counted as a "cost" of using stack-allocated frames. To quantify this cost, we measured two versions of the *Standard ML of New Jersey* compiler [AM91, App92] outfitted with the new closure conversion algorithm described in Chapter 4.

The version shown as *Ordinary Heap* in Table 20, allocates all frames and closures on the heap. The *"Stacklike Heap"* obeys the restriction that closures cannot point to frames (though frames can point to closures). "Frames" are those objects with LIFO lifetimes. But "Stacklike heap" proceeds to allocate frames and closures on the heap; it does *not* use a stack, and does not gain any advantages of using a stack.

The difference in execution time between the two versions is attributable *only* to the slightly more cumbersome representations that are imposed by the "closures cannot point to frames" restriction. The frames themselves are not much bigger, but the closures are: since they can't point to the frames, data from frames must be copied into the closures.

Table 20: Copying and sharing cost

The "Stacklike Heap" allocates all frames on the heap, but is careful to divide into two kinds: "stack" frames, which can point only to other "stack" frames; and "heap" frames, which can point to either kind. This lack of flexibility has a significant cost, as shown in the table. The first two columns show thousands of instructions executed; the third column shows thousands of frames created.
We count *frames* rather than *calls* because tail calls and leaf procedures do not make frames (stack or heap).

Program	Ordinary Heap $i/10^3$	"Stack-like" Heap $i/10^3$	Stack Frames $f/10^3$	Instrs per Frame	Extra Instrs per Frame
Boyer	61966	65770	662	94	5.75
Knuth-B	212702	222376	3465	61	2.79
Lexgen	310522	316813	1873	166	3.36
Life	48016	48437	201	239	2.09
YACC	114687	119065	1187	97	3.69
Simple	469543	492126	5516	85	4.09
VLIW	285370	292474	3274	87	2.17
Average				119	3.42

Table 21: Heap allocation data

This table shows the amount of frame allocation, the amount of non-frame allocation, and the proportion of allocation due to heap frames for the heap-based compiler. The last column shows the average frame size, calculated from the previous columns.
Even though the number of frames used by the heap-based compiler is slightly less than the number used by the stack-based compiler (because of improved copying/sharing) we use the stack-frame count for calculation, to make comparison between the two compilers more meaningful.

Program	Stack Frames $f/10^3$	Heap Frame Alloc. $words/10^6$	Other Alloc. $words/10^6$	$\frac{F}{F+O}$	Avg. Frame Size *words*
Boyer	662	3.17	2.61	0.55	4.8
Knuth-B	3465	12.92	11.55	0.53	3.7
Lexgen	1873	6.90	1.87	0.79	3.7
Life	201	0.63	0.99	0.39	3.1
YACC	1187	5.92	4.16	0.59	5.0
Simple	5516	25.23	13.09	0.66	4.6
VLIW	3274	15.72	15.90	0.50	4.8
Average				0.57	4.2

Some programs suffer more from this than others, but on the average the difference is quite significant: about 3.4 extra instructions are executed per every frame creation because of this restriction. Perhaps our lambda-lifting (closure analysis) algorithm is better tuned for heaps than it is for stacks, and this "copying vs. sharing" cost is overstated; it is difficult to tell.

5.5 Space safety

Assumption: The results of Table 20 are based on the assumption that the compiler must be "safe for space complexity" (see Section 5.5), which does put some restrictions on both the heap-allocated and stack-allocated frames.

It is possible to allow dead variables in frames and closures, *if the garbage collector knows they are dead.* This can be accomplished using special descriptors, which would reduce the "copying and sharing" penalty for stack frames.

For example, in the Chalmers Lazy ML compiler [Aug89] or the Gallium compiler [Ler92], associated with each return address is a descriptor telling which variables in the caller's frame are live *after the return.*[4] But this is not sufficient; heap closures still cannot point to stack frames. A fully flexible system must be able to let the stack frame point to a heap closure that contains several variables, some of which may die before the frame itself. The return-address descriptor would need to indicate not only which variables *in the frame* are dead, but which live variables point to records in which some of the fields are dead. This is complicated to implement, and we do not know of anyone who has done it.

5.6 Locality of reference

Stacks have excellent locality of reference: they are almost always moving up and down in a small region of memory, so access to the stack should almost always hit the cache, no matter how small that cache is.

But heap-allocated frames are scattered throughout memory, so creating and accessing them should cause more cache misses.

Since some machines these days have primary caches as small as 8k bytes, and secondary caches with miss penalties as long as 100 cycles, this is a serious concern.

[4]The bibliographic citations are merely *pro forma*; the author of neither paper has actually described this technique in print.

The analysis of cache behavior of garbage collected systems differs qualitatively depending on the size of the cache.

Large Caches For large (e.g., secondary) caches, a *generational* garbage collection algorithm [Ung86] can keep its youngest generation entirely within the cache [WLM92, Zor91]. Only the (rare) objects that survive a collection (or two) will be promoted into an older generation where they can cause cache misses. The collector itself helps to *improve* the locality of reference of the mutator. Thus, locality of reference in a large cache is basically a solved problem.

Furthermore, activation records die especially young. It will be extremely rare for an activation record to be promoted to a higher generation [SM94]. Since only the higher generations can cause cache misses[5], heap-allocated frames will (almost) never cause cache misses. Thus, while there may be secondary cache misses in a garbage-collected system, these will be on the non-frame objects (closures, records, etc.); the difference between stack-allocated and heap-allocated frames will be insignificant.

John Reppy has recently made empirical measurements of a multigeneration collector on a machine with a large (1MB) secondary cache. "The total CPU time reaches a minimum [significantly less than with the collector used this chapter] when the allocation arena is the same size as the secondary cache. This provides empirical evidence for the claim that sizing the allocation space to fit into cache can improve performance." [Rep93] Unfortunately, the measurements in this chapter were made using the older two-generation collector [App89].

Small Caches For small (e.g., primary) caches whose size is less than 100 kbytes, it is impractical to keep the youngest generation in the cache; doing so would cause garbage collections to be too frequent, and this would be expensive.

Let us consider locality in a small, primary cache. We assume that any cache of only 8 kbytes will have only a 10-cycle miss penalty—because there are many programs that cannot achieve a better than 90% hit rate in such a small cache, and machine designers will be forced to make a small miss penalty for "balanced" performance.

The essence of the locality argument against heap allocation is that stacks can exploit a small primary cache, and heap-allocated frames cannot. Stacks should have good locality even in a small cache. In a typical sequence of N procedure calls, the stack pointer is expected to go up and down over the same $\log(N)$ frames, re-using them over and over

[5] This is a slight oversimplification.

again. These frames should easily fit even in the smallest cache. Heap-allocated frames can have good locality in a large cache, but no one has analyzed locality in a small cache.

We will now demonstrate that heap-allocated frames have adequate locality of reference in a small cache, if the read miss penalty is not too large and the write miss penalty is zero.

5.6.1 Write misses

The Standard ML of New Jersey compiler [AM91] uses no stack; all frames are allocated on the garbage-collected heap. If any system should have poor cache locality, this is the one.

Diwan, Tarditi, and Moss [DTM94] simulated the memory-hierarchy performance of SML/NJ on a DECstation 5000, and found two things:

- SML/NJ program executions have an astoundingly high write-miss ratio.
- SML/NJ programs are not much delayed by cache misses.

The reason these two statements are not inconsistent, they discovered, is that the write-miss penalty on this machine is approximately zero—the write buffer can easily keep up with an enormous write miss rate.[6] Read misses stall the processor—which cannot continue computing until the data shows up—but write misses can be handled by the write buffer while the CPU continues its work. Many modern machines have a zero write-miss penalty, especially for their primary caches [Jou93]. Simulating machines with a high write-miss penalty, Diwan *et al.* found that SML/NJ performs badly, as might be expected.

Thus: on machines with a zero write-miss penalty, the average cost per frame of write misses is zero.

On machines with a nonzero write-miss penalty, the cost per frame is high. The average number of cache write misses caused by the creation of a frame is the ratio of frame size to cache line size (there is no fragmentation, because heap allocation is sequential and contiguous). Assuming a cache line size of 8 words (for example), and a frame size of 4.2 words (as in Table 21), the number of write misses per frame is about 0.53.

Thus, the cost of write misses shown in Table 18 is either 0 (for zero write penalty) or 5.3 (for 10-cycle write penalty). But see also Section 5.6.2.

They also found that *write-allocate* is important: on a write miss, the written data should be put in the cache. But a cache line is usually larger than a single word; on a write miss, "traditional" *(fetch-on-write)* caches read the rest of the line from memory; this can

[6] Reinhold [Rei94] makes similar observations about the interaction of garbage collection and caches, though not for a compiler with heap-allocated frames.

cause write misses to be slow, and also causes unnecessary traffic on the memory bus in the common case of sequential writes that will overwrite the just-read data. The simulations of Diwan *et al.*, and our analysis in Section 5.6.3, both show that this policy is costly.

Heap allocation (in a system with copying garbage collection) consists of sequential writes to a large contiguous free region. Under such a discipline, there are some equally good cache implementation strategies that will permit (or simulate) write-allocate with zero write-miss penalty.

Sub-block placement: With sub-block placement (also called *write-validate*), a write miss on one word will be written to the cache, and the rest of that cache line will be marked as allocated but invalid. Thus, a write miss does not require reading the rest of the written cache line from memory. Subsequent (sequential) writes will fill the rest of the line.

One-word cache line: The DECstation 5000 has a cache-line size of one word, but four lines are read on a miss [DTM94]. For some applications this is better than sub-block placement, but for sequential writes it is equally good. It is more expensive to implement, since it requires a full tag (not just a valid bit) for each word. Diwan *et al.* found excellent memory-subsystem performance for SML/NJ on this machine.

Cache-line zero instruction: On some machines (e.g., IBM R/S6000 [HHH*90] and PowerPC [AB93], Power2, HP PA) a cache line (64 bytes) can be allocated and zeroed with a special instruction. This avoids the write miss, with a 0.687-instruction cost per frame.[7]

Cache-control hint: On the HP PA7100, a store instruction can have a cache-control hint specifying that the block will be overwritten before being read; this avoids the read if the write misses [AAD*93]. But these machines have very large primary caches anyway, so locality can be handled by generational collection.

Smart write buffer: [8] Instead of sub-block placement (which complicates the cache), one might add a feature to the write buffer: write misses normally bypass the cache, but if the write buffer accumulates a full cache line, this line is put in the cache.

[7] In detail: the allocation pointer is made always to point exactly 64 bytes ahead of the next allocatable word. On each heap-limit check, a cache-line clear is performed. This does not clear the line currently being stored into (which might overwrite a frame recently allocated) but the line *soon to be entered*. Because the heap-limit check is often shared with a non-frame allocation (see Table 19), the average net cost *per frame* is only 0.687 instructions.

[8] This idea is discovered by Andrew Appel.

For sequential writes this is as good as sub-block placement. On a multiprocessor with cache coherence, this technique might be easier to implement than sub-block placement, because no cache line would ever be dirty (but partially full) in two different caches.

Garbage-prefetch: On a machine with a no-write-allocate (*write-around*) cache, write-allocate can be simulated (as long as read misses are nonblocking) by fetching the cache line (with an ordinary **read** instruction) in advance of the write [App94a]. This technique works (providing a modest performance enhancement) on the DEC Alpha 21064 [Dig92], for example.

On any of these machines, heap-allocated data should not incur a write-miss penalty.

Assumption: Any small cache will have write-allocate and no write-miss latency (or write-allocate can be emulated).

Indeed, this is not true of all machines: the VAX 11/780, VAX 8800, and Pentium do *write-around*, bypassing the primary cache on write misses (causing subsequent read misses); and most pre-1993 designs do *fetch-on-write,* stalling the processor on a write miss [Jou93]. In fact, the bad performance of garbage-collected systems on machines with a write-miss penalty is a good reason not to build such machines.

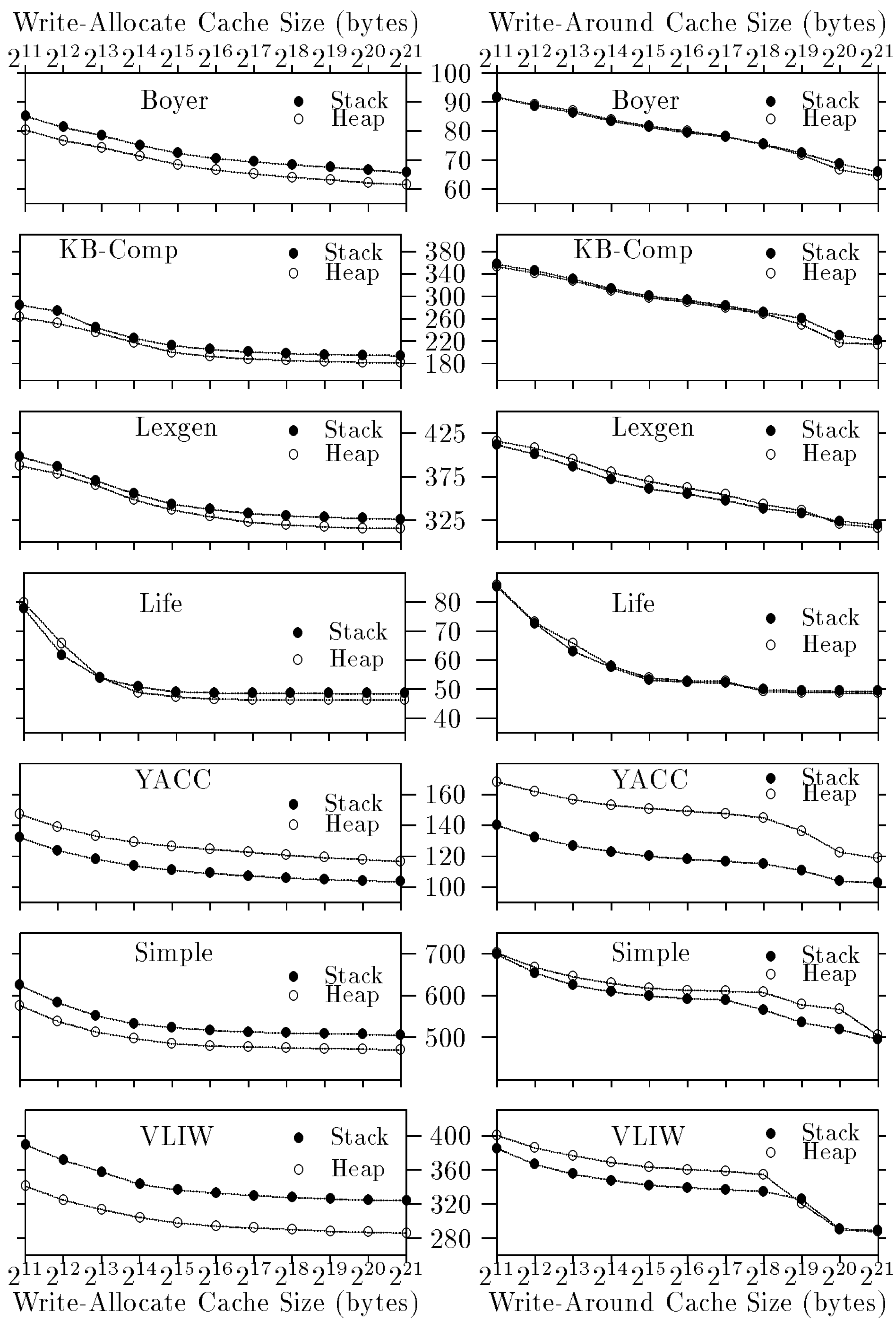
Finally, note that a write-miss penalty on large caches is not particularly problematic; as explained above, generational garbage collection solves that problem. The analysis in the rest of this section applies only to small caches.

5.6.2 Read misses: simulations

To see the effect of small caches on heap-allocated frames, we simulated several "standard" SML benchmarks (selected from Table 2) in two versions of the SML/NJ compiler: a *Heap* version with heap-allocated frames, and a *Stack* version with stack-allocated frames. The simulations was run in direct-mapped D-cache (of various sizes) and "infinite" I-cache. The simulations counted read misses, write misses, and total instruction count of SML programs compiled to the MIPS instruction set. The total instruction count also includes the instructions and cache misses of garbage collection.

Diwan *et al.* [DTM94] measured a heap-only ML system; Reinhold [Rei94] measured a stack-frame Scheme system. In order to make a more direct comparison, we measured stack frames vs. heap frames in the same ML system.

We simulated only the primary data cache. We simulated direct-mapped caches of sizes ranging from 2 kbytes to 2 Mbytes, with a 32-byte line size. Many modern machines



Simulations running in direct-mapped D-cache of various sizes and "infinite" I-cache. Vertical axis shows execution cycles in millions. Cycle count for stack programs is reduced by 6× number of frames, to discount the 7-instruction quasi-stack allocation/deallocation sequence.

Figure 25: Simulations: write-allocate vs. write-around cache

have direct-mapped caches especially at the first level of the memory hierarchy, so that tag comparison can be overlapped with further computations on the value fetched [Hil88].

Instead of a detailed cycle-level simulation, we use the approximation that each cache miss stalls the instruction-execution pipeline for p cycles, where $p = 10$ is the "miss penalty." Many modern machines do not stall non-memory instructions on a cache miss; for these machines our simulation will provide an upper bound on cache delays, which is sufficient for our analysis.

We did not simulate a conventional, contiguous stack. Instead, we implemented a free list of 8-word re-usable frames (a *quasi-stack*). Frames are popped by putting them back on a free list. This takes more instructions than conventional pushing and popping, but *should not cause more cache misses*: programs will still go "up and down" over the same tiny set of (noncontiguous) frames, and even a small cache should be able to hold these frames along with other frequently used data.

Measurements of SML/NJ show that most frames are smaller than eight words (see also Table 21); we do not load frames down with lots of useless overhead. When larger frames are needed, our "stack" simulation simply links together enough 8-word (32-byte) frames. Aggregate objects (arrays, records) are never kept in frames.

Free-list handling costs six instructions more than stack-pointer incrementing, so we subtract this cost when presenting results of the simulation (Figure 25).

Our garbage collector "marks" any frame that survives a collection; marked frames are not put back on the freelist upon procedure return. This enables our stacks to work well with generational garbage collection and with first-class continuations. At a youngest-generation collection, the freelist is set to *nil*; after the collection, new frames will be obtained from the heap (and, when freed, put back on the freelist).

Using a free list of frames, there is a considerable cost to allocate and deallocate a frame:

1. Test the freelist register.[9]
2. Set freelist register to the next free frame.

To deallocate,

3 Fetch the mark field.

4 Wait for fetch to finish.

[9] If the freelist is empty, one must heap-allocate instead of taking from the freelist (the heap-allocated frame will be deallocated back onto the freelist). But this case is so rare that we won't count it in the average cost.

5 If marked, stop here (do not put back on free list).

6 Store free list register into newly freed frame.

7 Set the free list register to point to this frame.

Thus, there is an overhead of seven "instructions" for stack allocation. But "ordinary, contiguous" stacks do not have this seven-instruction penalty—there's just a single "pop" instruction. Therefore, we adjust the execution time of the Stack version of the program by subtracting six cycles per frame.

Figure 25 shows the run times (after adjustment) of several benchmarks using *Heap* and *Stack* frames, running in simulated caches of different sizes. We simulated a write-allocate cache with partial fill (the left-hand-side of Figure 25), and also a write-around cache (the right-hand-side of Figure 25).

Jouppi [Jou93] simulated both kinds of cache for C programs without garbage collection; Diwan et al. [DTM94] simulated both caches for almost purely heap-allocating ML programs. By simulating both caches on stack *and* heap allocation for the same programs, we can compare more straightforwardly.

The results are not too surprising: write-allocate is better on all programs than write-around; and heap allocation is more sensitive to the cache policy than is stack allocation.

Though there are many differences between the Heap and Stack implementations that affect the run time, it is clear from the shapes of the curves that the *cache locality* behavior of heap and stack *in a write-allocate cache* is almost identical. (That is, if the two curves were translated vertically so that the large-cache points coincide, then the rest of the curves would be extremely close.)

The simulation measurements (Figure 25) show a cache-read-miss cost (for 16k write-allocate cache) of 1.0 cycles per frame. We calculate this by averaging

$$((D_{16k,heap} - D_{2M,heap}) - (D_{16k,stack} - D_{2M,stack})) * P/F$$

over all the benchmarks, where $D_{c,x}$ is the number of read misses in data cache size c with frame strategy x, $P = 10$ is the miss penalty, and F is the number of frames created (taken from Table 20).

A nonzero write miss penalty is usually found only on processors that *fetch on write*. Such processors will then allocate the line in the cache, so the average read miss cost will be low (as described in the previous paragraph). For a 10-cycle miss penalty, the write-miss cost per frame (as explained in Section 5.6.1) should be about 5.3 cycles. When the cost of read misses for a write-allocate cache is included, the total cost of read+write misses is 6.3 cycles per frame.

Write-around caches do not need to stall the processor on a write miss, at least for the well behaved sequential writes performed by a heap allocator. Thus, the write miss cost will be zero; but since (almost) every newly written cache line will soon be fetched [SM94, Rei94], we should expect the read-miss cost per frame for write-around caches to be similar to the cost per frame for fetch-on-write caches. For a 16k write-around cache, the cost of read misses (calculated from the simulations) is 5.1 cycles per frame—somewhat smaller than the 6.3-cycle read+write miss cost for fetch-on-write caches. We have not shown write-around caches in Table 18, but their total cost will be similar (though slightly smaller, for heap frames) to that of fetch-on-write caches.

Clearly, the small size of frames in SML/NJ is important in achieving good performance, especially for fetch-on-write or write-around caches.

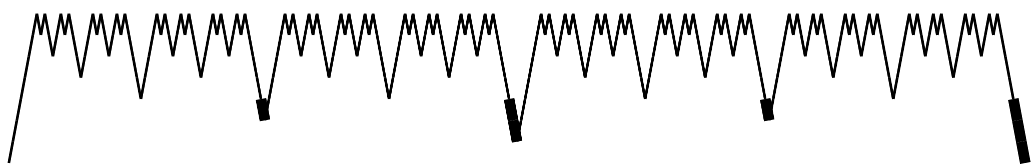


Figure 26: Execution of $T(7)$ in a 16-line cache. Every uptick (procedure call) is a write miss; only the bold downticks (procedure returns) are read misses.

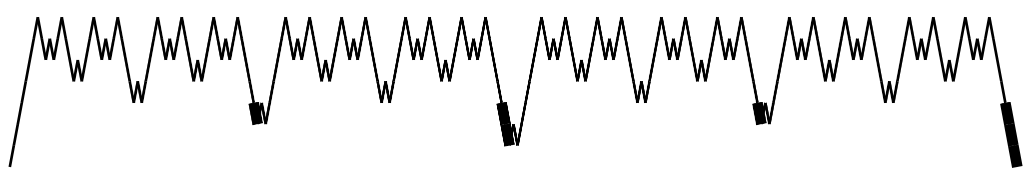


Figure 27: Execution of $T'(6)$ in a 16-line cache. Only the bold downticks (procedure returns) are read misses.

5.6.3 Read misses: analytically

Stacks continually re-use recently used frames for new purposes; heap-allocated frames "thrash" through the cache in sequential order. How could it possibly be the case that—in a write-validate cache—they both have equally good locality of reference?

Since the simulations give us no analytical understanding of what is really happening, we focus on three "typical" patterns of procedure call and return:

1. Tail recursion: each call re-uses the same frame, no allocation is performed.

2. Deep single recursion (as for the recursive factorial function): a deep sequence of calls followed by the returns from all of them.

3. "Towers of Hanoi:" lots of short up-and-down motion, but every Nth call briefly returns to depth $\log(N)$. This is shown graphically in figures 26 and 27.

We claim that these patterns represent "both extremes and the middle" of all procedure-call patterns. Furthermore, the Towers of Hanoi should be a *worst case* for heaps (as compared to stacks)—because the stack implementation has excellent locality—so its analysis will be instructive.

We assume that the cache is direct-mapped[10] and can hold C frames, with a line size equal to the size of a frame (different line sizes would affect our results by a constant factor). We assume that heap-allocated frames are allocated at sequential addresses. We assume that no heap allocations or memory accesses are performed, other than for activation records.

Then *tail recursion* is easy to analyze: No new frames are allocated by either the stack or heap methods. The miss rate for both is zero.

Deep single recursion is bad for both stacks and heaps: If the recursion is depth N, then there will be no read misses on any of the calls; the first C returns will hit, and the rest of the returns will have read misses. The miss ratio approaches 1 as N becomes much larger than C.

Towers of Hanoi should be ideal to demonstrate the better locality of reference of stacks in a small cache. Let us analyze its cache behavior carefully.

```
fun T(d) = if d>1 then { T(d-1); T(d-1) } else ...
```

Executing $T(d)$ requires $2^d - 1$ procedure calls. If $d \leq C$, all the stack frames fit in the cache, so the read-miss rate is zero (we will ignore write misses). If $d > C$, then there will be a miss on every return from a call to $T(e)$, where $e > C$. The number of such returns is $2^{d-C} - 1$. The miss rate is the number of misses divided by the number of calls, or approximately 2^{-C}. For an 8-kbyte cache, $C = 256$, so the miss rate for stacks is indeed negligible.

Now consider executing $T(d)$ with heap-allocated frames. The frames are allocated sequentially. If a clock counts one tick for each procedure call, then at time t the frame

[10] The results *for these three simple programs* would be the same for set-associative caches with LRU replacement in each set.

created at time $t-C$ will be removed from the cache by a newly allocated frame. Therefore, exactly those procedures that return more than C time units after they are entered will cause a cache miss upon return to their callers.

Executing $T(d)$ requires $2^d - 1$ calls, of which only $2^{d-\log_2 C}$ take more than C time units to execute. Thus, the amortized number of misses per call is about $2^{-\log_2 C}$, that is, one miss every C calls.

Figure 26 illustrates $T(7)$ running in a tiny cache ($C = 16$). The cache misses are clustered at intervals of $2C$ returns, but there are an average of 2 misses in each cluster.

Figure 27 shows a more traditional Towers of Hanoi, that calls `print("move disk d")` at each step.

In a 16k cache—which can hold 512 thirty-two-byte frames—the stack version will have one miss every 2^{512} calls, but the heap version will have one miss every 512 calls. Assuming that the primary cache miss (with secondary cache hit) costs 10 cycles, this is a cost of *one cycle every 50 calls.*

The analytical prediction (0.02 cycles/call) does not completely agree with the simulation (1.0). We believe this is because the simulation cannot directly measure the cache miss cost, because the "stack" and "heap" programs do not execute the same instructions, or even the same number of instructions. Instead, we measure the total cost of two different implementations (with different frame creation sequences and frame disposal sequences, different garbage collection times, garbage collectors trashing the cache at different frame layout, etc.) and attempt to subtract out the components not related to cache behavior. Small errors in the estimation of any of these components will be additive.

On the other hand, the simulation can capture the effect of "real" programs that cannot be analyzed in closed form like our three paradigmatic examples. Such programs will have interference effects from old objects and nonframe objects. Reinhold [Rei94] , however, finds that such interference does not much affect the cache behavior of recently allocated objects (such as, in our case, frames).

Therefore, we cannot say with confidence whether the simulation or the analytical prediction more accurately characterizes the cache behavior. To be conservative, we use the "worse" number (1.0) for **Cache read misses** in Table 18.

But now consider what happens in a write-no-allocate (e.g., write-around) cache. The vast majority of reads in the Towers of Hanoi example are to blocks that recently caused write misses. Even if the write misses themselves do not stall the processor, the read misses will. This costs about 5 cycles per frame (assuming the average frame size is roughly half the cache line size, and a 10-cycle miss penalty).

Future cache designs

What cache write-miss policies can we expect in the future? We must assume that hardware designs of "commodity" microprocessors will be driven by the SPEC benchmark suite, not by arguments about what's best for functional programs or garbage collection.

Jouppi's measurements of C and FORTRAN programs [Jou93] may perhaps be influential. He concludes that *write-validate* (that is, *write-allocate, no-fetch-on-write*) is the policy with best performance. This is exactly the policy that we and others [DTM94, Rei94, SM94] find best for garbage-collected strict functional programs.

On the other hand, as Jouppi points out, write-validate is difficult—though not impossible—to implement on a shared-memory multiprocessor with cache coherence. Such machines require each writable cache line to have a single owner. Since manufacturers will wish to make multiprocessor-compatible CPU chips, and won't wish to have two different primary-cache designs, this could mean that write-validate will not be common on multiprocessors *or* uniprocessors.

5.7 Disposal

To de-allocate a stack frame, one instruction is required to subtract a constant from the stack pointer. No explicit pop instruction is necessary to deallocate a heap frame. The (previous) frame pointer must be fetched, but we have counted this already under the heading "Frame pointers."

What is the garbage collection cost of heap-allocated frames? There are three components:

1. Live heap frames must be copied to an older generation and then scanned, whereas live stack frames need only be scanned.
2. Allocating frames causes more frequent garbage collection, leading to the premature promotion of non-frame objects that might have died if given just a little more time.
3. More frequent collections means more frequent executions of the garbage collector's entry-exit sequence.

We will analyze these costs separately.

Copying frames

We will analyze the cost, in instructions, of garbage collection for the three typical call-return patterns discussed in the previous section. We assume a generational collector with a youngest generation that holds G frames (for generation size of 128 kbytes, frame size of 32 bytes, $G = 4096$). On a youngest generation collection, all live frames are promoted to the next generation.

1. Tail recursion does not allocate, so the amortized cost per call is zero.

2. Towers of Hanoi will garbage-collect every G calls. At this collection, there will be at most $\log(d)$ live frames; but only $\log(G)$ of them will be "new" (not already promoted). The cost of collecting them will be $c\log(G)$, where c is the cost of copying one frame (perhaps 20 instructions).[11] The amortized cost per call is $c\log(G)/G$, or about 0.06 instruction per call.

3. A very deep recursion (deeper than 2000 calls) will promote almost every frame, at a cost of cG instructions per G calls, or c instructions per call. This is costly; but recursions this deep also begin to miss in the secondary cache! This is particularly so, since the size of the youngest generation should be less than the size of the secondary cache [Zor91, Rep93]. The secondary cache misses (which occur for both stacks and heaps) are probably just as important as the garbage-collection overhead.

Summary: 0.06 instructions per frame.

More frequent collections

With stack-allocated frames, N garbage collections of average cost $E + L$ will occur, where E is the entry-exit overhead of the collector and L is the cost of copying non-frame live data.

With heap-allocated frames, N' collections of average cost $E + L' + F$ will occur, where F is the the cost of copying frames. We have accounted for $N' \cdot F$ in the previous subsection. We will account for $N'L' - NL$ in the the next subsection.

Here we calculate $E(N' - N)$. $N' - N$ is simply the number of frames created divided by G, the number of frames the youngest generation can hold. Thus the cost *per frame* is just E/G; with $G = 4096$ and assuming $E = 800$ instructions, this is 0.2 instructions per frame.

[11] We do not include the cost of scanning the frame for pointers, because this has to be done for either the stack or the heap case.

Table 22: Garbage collection cost

Mutator time μ and garbage-collection time γ are shown (in seconds) for each benchmark. MIPS (ρ) and garbage-collection overhead per frame (Z_h and Z_q) are calculated as shown in the accompanying text.

	Heap		Q-Heap		Stack		MIPS	G.C.Instrs/Frame	
	μ_h	γ_h	μ_q	γ_q	μ_s	γ_s	ρ	Z_h	Z_q
Boyer	1.11	0.86	1.18	0.87	1.17	0.88	31	-0.95	-0.40
Knuth-B	5.96	0.86	6.31	0.93	6.56	0.88	31	-0.20	0.19
Lexgen	9.45	0.60	9.56	0.72	9.77	0.59	31	0.03	0.99
Life	1.24	0.04	1.29	0.06	1.28	0.03	38	2.62	2.20
Yacc	3.02	0.92	3.28	1.18	3.21	0.60	29	7.86	8.85
Simple	15.30	0.54	14.42	0.65	15.29	0.55	30	-0.03	0.33
VLIW	11.29	0.54	15.39	0.57	15.05	0.51	24	0.22	0.25
Average								1.36	1.77

Premature promotion of non-frames

In principle, the heap allocation of frames should cause more frequent collection (and undesirable promotion) of the non-frame data. We use lifetime statistics[12] as reported by Stefanovic and Moss [SM94] to find those objects that survive G calls but not $p \cdot G$. (The proportion $p = 0.57$ is the proportion of frames to total heap allocation (see Table 21).) Because of the shape of the object-survival curve, such objects are rare (1 object in 1000 allocations).

Consider a program that allocates one "ordinary" (i.e., non-frame, "random" lifetime) object per procedure call. One in 1000 of these objects will be promoted "to excess" in the heap-frame version, because the frame allocations cause more frequent collection. The cost of each such promotion is about 100 instructions. Thus the average cost per frame is about 0.1 instructions.

Sum of the collection costs

The three components (0.06, 0.2, 0.1) sum to 0.36 instructions per frame attributable to garbage collection.

Direct measurement of garbage collection

To support our analytical calculations of garbage-collection overhead, we measured the garbage collection time for stack vs. heap frames on benchmark programs. Table 22 shows

[12]The measurements done by Stefanovic and Moss [SM94] are based on an older version of the SML/NJ compiler which does not use the new closure conversion algorithm described in Chapter 4.

garbage collection costs for the execution of the benchmark programs on a DEC 5000/240 computer.

For each benchmark, three versions of the program were run: *Heap* (heap-allocated frames); *Stack* ("stack-allocated" frames, implemented as a free-list of re-usable frames, which have a different frame layout and choice of closures); and *Q-Heap* (heap-allocated frames that have exactly the same frame layout (and padding to 8 words) as the *Stack* frames).

The mutator time μ and garbage-collection time γ are shown for each benchmark. Times were calculated by executing each benchmark command five times consecutively (from within the same Unix execution) and dividing by five. Ten runs of each such test were made, and the fastest taken (as recommended, for example, by the SPEC benchmark consortium [Sta89]).

Time spent in the operating system is not shown, but was small in all cases (and did not much differ among the three versions of each program).

We calculated ρ, the effective MIPS (millions of instructions per seconds) for the DEC 5000/240 on each program, by dividing the instructions executed for the heap version of each program (taken from Table 20) by $\mu_h + \gamma_h$. The peak performance of this machine is 40 MIPS.

To calculate the extra garbage-collection cost attributable to heap-allocated frames, we compared stack g.c. time from heap g.c. time, converted from seconds to instructions, and divided by the number of frames F taken from Table 21:

$$Z_h = (\gamma_h - \gamma_s)\rho/F$$

In many cases this is negative! This indicates that any garbage-collection overhead of heap-allocated frames is less important than the benefit gained from the copying and sharing costs.

We then tried an alternate method of calculation. Since *Q-Heap* and *Stack* use exactly the same frame layout, the *only* difference is the failure of *Q-Heap* to free its frames. Thus, the garbage-collection overhead can be more consistently isolated. However, *Q-Heap* frames are all artificially padded to 8 words. This will overestimate the load on the collector; we expect any added load to be (roughly) proportional to the total size of all heap-allocated frames. Therefore, in our estimate of the overhead Z we multiply by the proportion of the frames that are not just padding:

$$Z_q = (U/8)(\gamma_q - \gamma_s)\rho/F$$

where U is the average frame size of each benchmark, taken from Table 21.

The Yacc benchmark is anomalous in showing a very high cost, in extra garbage collection, for heap-allocated frames. Closer examination of the Yacc execution showed that there were three major-generation collections with heap frames, but only two with stack frames.

Excluding Yacc, the average Z_h is 0.28 instructions/frame, close to the analytically predicted value of 0.36. Yacc must do something not foreseen by our analytical methods.

In Table 18 we show the measured value of $Z_h = 1.4$ instructions for disposal of heap-allocated frames.

5.8 Finding roots

In any garbage-collected system, local variables in activation records (e.g., stack frames) may point to the heap. At the beginning of each garbage collection, the collector must scan the frames to locate "roots" of the live data.

In a system with generational garbage collection, there is often very little live data in the youngest generation. Scanning a large stack would take more time than the rest of the collection! Therefore, the collector should scan only those stack frames *created since the last collection* and not yet popped.

It is trivial to treat heap-allocated frames this way. They are promoted (along with other live data) to older generations; older-generation data need not be scanned at a youngest-generation collection. Only the newly allocated (and not yet dead) frames will be scanned at a typical collection.

With stacks, a special trick is required. After a collection, the collector must mark the top stack frame. All frames underneath this are known to be "old." At the next collection, the stack must be scanned only from the top of the stack down to the "high-water mark;" for only these frames can contain pointers to the youngest generation.

But there is a complication. Between collections, if the "high-water" frame is popped, the mark must be moved down to the next-lower frame [Wil91]. The simplest way to do this would be to test for the mark on every return, but this would be expensive. Instead, the mark consists of a "special" return address, which replaces the real return address of a frame. When control returns to this point, the program at this special location executes, placing the mark (that is, the special return address) in the next-lower frame, and jumping to the real return address.[13]

The cost of this technique is quite low. The cost of placing and removing the high-water mark is between 10 and 100 instructions. Every frame that survives its first garbage collection will eventually hold the high-water mark. The cost of moving the high-water mark (in a stack-based system) is similar to the cost of promoting a live stack frame to the older generation (in a heap-based system); and it is exactly the same frames (new frames live at a collection) that need this service in either case.

The proportion of new stack frames live at a collection is usually extremely low, so the cost is negligible for both stacks and heaps. In rare cases (very deep one-way recursions) the cost will be higher, but the stack-based systems and heap-based systems will pay approximately the same price.

Doligez and Gonthier [DG94] have suggested that the collector put a one-bit mark in *every* live stack frame that it scans; this mark will be ignored by the collector but will be cleared in new frames. This is fine, if there is already some word in every frame that has a free bit.

Keeping track of the high-water mark in heap-based system has *no* implementation complexity: it is a natural consequence of garbage-collecting live frames. In contrast, in a stack-based system similar results can be achieved but it requires extra work.

Updating activation records

In order to guarantee that only "new" heap frames can be roots for garbage collection, it is necessary to prohibit any writes to frames after they have been allocated. Compilers using continuation-passing style (such as Rabbit [Ste78], Orbit [KKR*86], and SML/NJ [AJ89]) naturally initialize frames as soon as they are allocated, and then never write to them again. In effect, they save up any changes in registers, then dump everything out all at once. With good use of callee-save registers [AS92, App92] (also see Chapter 4), it is even easier to accumulate any changes in registers and write immutable frames in big chunks.

A stack-based compiler could update the topmost frame at any time, and the collector could always scan this frame for roots. But a heap-based compiler that wants to support efficient call/cc (see Section 5.9) should never update a frame after its initialization, because if a continuation is invoked more than once the two invocations will stomp on each others'

[13] This complicates the compiler and runtime system, particularly the implementation of exception handlers that must pop the stack.

data. In such a compiler, it is best to keep the top frame in callee-save registers and not in memory at all.

5.9 First-class continuations

The notion of "first class continuations" using the *call-with-current-continuation* (call/cc) primitive originated in the Scheme language [RC86] and has since been adopted in other systems as well [DHM91]. First class continuations are useful for implementing coroutines [Wan80] and concurrency libraries [Rep91].

But call/cc is much harder to implement efficiently if there is a stack. With an ordinary contiguous stack implementation, the entire stack must be copied on each creation or invocation of a first-class continuation. This is unacceptably slow if, for example, call/cc is the primitive used in implementing a concurrency library or exception-handling system.

With purely heap-allocated frames, which are not updated after their initialization, call/cc is no more expensive than an ordinary procedure call: the live registers must be written to a closure record, and that is all.

There have been mixed stack/heap implementations intended to support call/cc efficiently in the presence of stacks [CHO88b, HDB90]. The basic idea is to make a "stack chunk" that holds several stack frames; if this fills, it is linked to another chunk allocated from the heap. This turns out to be complicated to implement.

Stack chunks require a stack-overflow test on every frame,[14] so creation costs three instructions (add to SP, compare, branch).

Danvy [Dan87] made a free list of re-usable frames (we call this a "quasi-stack"); these reduce the load on the garbage collector and have good locality; but they are expensive to create and destroy, and require a frame pointer. The *"stack"* implementation that we have implemented and measured is actually a simplification of Danvy's method. For applications using first-class continuations (call/cc) our simplification would need an extra mechanism to copy part of the continuation, whereas Danvy's method does not.

Both methods suffer from the same "copying and sharing" penalty as ordinary stacks. Their performance is summarized in Table 18, and does not appear competitive, especially given the implementation complexity.

The simplicity and efficiency of call/cc in a pure heap discipline is a strong motivation for avoiding stacks.

[14] "Unfortunately, it has been our experience that memory exceptions are not a tenable means for detecting stack overflow...." [HDB90]

5.10 Implementation

One reason to avoid stacks is that they are complicated to implement, especially with all the tricks that are necessary to achieve good performance. Let us compare the implementation complexities of heaps vs. stacks, in a garbage-collected environment:

Implementation of Heap Frames

1. To achieve good performance with heap frames, it is necessary to have an sophisticated algorithm to choose closure representations. This algorithm must preserve space complexity, promote closure sharing, and use callee-save registers to minimize the number of distinct frames written. Chapter 4 has already described an implementation of such an algorithm, which is not particularly hairy.

2. To avoid having a descriptor in each frame, the runtime system can maintain a mapping of return addresses to frame layout descriptors. Kranz's ORBIT compiler used this technique [Kra87]. *Standard ML of New Jersey* does not bother, so it does indeed pay the price of a descriptor in each frame.

Implementation of Stacks

1. A good closure analysis algorithm must be used to preserve space complexity while still trying to avoid too much copying. It is not clear that such an algorithm will be much simpler than the one for pure heaps. In particular, most conventional stack implementations are not safe for space complexity.

2. To preserve space complexity and correctly implement tail recursion, certain activation records require a complicated scheme to determine when they must be popped [Han90]. (Or these frames could be heap allocated, even in a stack discipline; but they must be identified by static analysis.)

3. A high-water mark must be maintained to achieve efficiency in the generational collector.

4. If call/cc is to be supported, then stack copying or some more complicated technique must be implemented [HDB90].

5. To avoid having a descriptor in each frame, the runtime system must maintain a mapping of return addresses to frame layout descriptors.

6. In a system with multiple threads, each thread must have its own stack. A large contiguous region of virtual memory must be reserved.[15]

7. Stack-overflow detection must be implemented. In most cases this is handled automatically by the operating system using virtual-memory page faults.[16]

No stack implementation that we know of handles all of these necessary complexities. As a result, some are not safe for space complexity; some do not implement call/cc; and some scan too many frames on each collection. It is an open question whether all of these tricks can fit together in a real system.

5.11 Summary

Heap allocation of activation records is simple and competitively efficient. The fact that heap allocation is about as cheap as stack allocation, when all effects including cache locality are counted, certainly contravenes the conventional wisdom.

Heap frames are much easier to implement correctly: it is tricky to make stacks "safe for space complexity," or to support generation garbage collection efficiently, or first-class continuations (call/cc). In *Standard ML of New Jersey* compiler, which supports all of these features, heap allocation of activation records has proved to be a great success.

When *call-with-current-continuation* is needed, heap frames are *much* better than stack frames. Various hybrid systems (stack chunks, quasi-stacks) designed to support *call/cc* efficiently with a stack are less efficient than heaps for *both* normal call/return *and* call/cc.

On machines with a write-miss penalty, or where writes entirely bypass the cache, the results are different: heap-frame handling is about twice as expensive as stack-frame handling (about 7% penalty in overall performance), except for first-class continuations.

Finally, for languages without nested first-class functions with static scope, there is no "copying and sharing" cost. In this case stacks have a 6% overall performance advantage. Without closures, call/cc is not an issue, of course.

[15] In contrast, one heap-allocation region is necessary *per processor*, not per thread.

[16] Heap overflow detection must also be implemented, but this is true whether or not there is a stack.

Chapter 6

Unrolling Lists

Lists are ubiquitous in functional programs, thus supporting lists efficiently is a major concern to compiler writers for functional languages. Lists are normally represented as linked *cons* cells, with each *cons* cell containing a *car* (the data) and a *cdr* (the link); this is inefficient in the use of space, because 50% of the storage is used for links. Loops and recursions on lists are slow on modern machines because of the long chains of control dependences (in checking for *nil*) and data dependences (in fetching *cdr* fields).

In this chapter, we present a data structure for "unrolled lists," where each cell has several data items (*car* fields) and one link (*cdr*). This reduces the memory used for links, and it significantly shortens the length of control-dependence and data-dependence chains in operations on lists.

We further present an efficient compile-time analysis that transforms programs written for "ordinary" lists into programs on unrolled lists. The use of our new representation requires no change to existing programs.

We sketch the proof of soundness of our analysis—which is based on *refinement types*—and present some preliminary measurements of our technique.

6.1 Introduction

Efficient implementation of lists has always been a major concern to compiler writers for functional languages, because they occur so frequently in functional programs. Lists are normally represented as linked *cons* cells, with each *cons* cell represented by two contiguous memory locations, one for the *car* (the data) and another for the *cdr* (the link). This is inefficient in the use of space because half of the storage is used for links. Furthermore, traversing a list requires twice as many memory references as traversing a vector. And on any loop or recursion that traverses a list, there is a long chain of control dependences as each link is checked for *nil*; and a long chain of data dependences as each link fetch is dependent on the previous one. With modern superscalar hardware, these dependences are a serious bottleneck.

In order to save on storage for links, "cdr-coding" was proposed in the 1970's [Han69, Gre77, Cla76, CG77, BC79, Bob75]. Its main idea is to try to avoid some links by arranging for the second cons cell to directly follow the car of the first, and to encode that information in several bits contained in the car field of the first cell; thus the first cell does not need a cdr field at all. A depth-first (or breadth-first [Bak78]) copying garbage collector helps ensure that most lists are arranged sequentially in storage, so they can take advantage of this encoding. Cdr-coding solves the space-usage problem (and in the MIT version allows random access subscripting of lists [Gre77]), but makes the control-dependence problem even worse, as the cdr-coding tag of each *car* must be checked. Cdr-coding was popular on microcoded Lisp machines *circa* 1980 [WM81, Deu73], but it is not an attractive solution on modern machines.

Our new "compile-time cdr-coding" method works for statically typed languages such as ML. Our scheme allows a more compact runtime representation for lists, but *does not require any runtime encoding at all.* Furthermore, our encoding allows loops and recursions on lists to be unrolled much more efficiently than is possible with the conventional representation for lists.

Table 23: Standard vs. Unrolled List Representations

Length	**Old Standard Representation**	**Size**	**New Unrolled Representation**	**Size**
0	0	0	[E | 0]	0[1]
1	['a | 0]	2	[O | 'a | 0]	3
2	['a |] → ['a | 0]	4	[E |] → ['a | 'a | 0]	5
2n	['a |] → ['a |] → - - - → ['a | 0]	4n	[E |] → ['a | 'a |] → ['a | 'a |] → - - - → ['a | 'a | 0]	3n+2
2n+1	['a |] → - - - → ['a | 0]	4n+2	[O | 'a |] → ['a | 'a |] → - - - → ['a | 'a | 0]	3n+3

Our idea is simple: we put k items—but only one link—in each list cell. We use $k = 2$ to illustrate our idea. A list of *even* length is simply represented as a linked series of our bigger cons cells; a list of *odd* length is represented as a header cell that contains one data element

[1]The two-word record representing the NUR empty list ([E | 0]) can be shared among all uses of the empty list. This sharing can be introduced by the garbage collector to avoid complicating the compiled code, if necessary.

and one link to an even-length list. Table 23 gives a simple comparison of space usage between our *new unrolled representation* (NUR) and the *old standard representation* (OSR). In the table, "`'a`" represents the data element; "`O`" and "`E`" represent the tag word that is used to distinguish between odd-length and even-length lists at runtime. We represent the empty list by "0." Now we can easily see that for lists with length greater than 2, the new representation requires 25% less space than the usual representation. Furthermore, traversing a list in the new representation requires 25% fewer loads, and 50% fewer tests for *nil* on cdr pointers (because NUR has 50% fewer cdr links).

The new unrolled representation (NUR) promises to be extremely useful for superscalar or superpipelined machines. Suppose we use a representation with k items per link. Then we can unroll most loops on lists by a factor of k, and overlap (using standard software pipelining techniques) the executions of the (original) iterations. Such unrolling and software pipelining would be much less fruitful if performed on the standard list representation (OSR) for two reasons: the tests for nil introduce a chain of $k-1$ extra control dependencies, and the fetches of cdr introduce a chain of $k-1$ extra memory latencies. These chains are a serious obstacle to the software pipelining of anything at all! Note that the fetches of the k *car* fields (in an unrolled loop using NUR) can all be done in parallel; this is not possible in the standard representation.

Because programmers will still use the standard list notation (i.e., each list cell has a head and a tail), the compiler has to do the appropriate translation to utilize the new unrolled representations. This is possible in a statically typed language such as ML, because the type of each identifier is statically known at compile time, and computation on lists is expressed using pattern matching and recursive functions. For example, an integer list[2] in ML might be represented by the following concrete datatype, which matches the standard representation (OSR) in Table 23:

```
datatype list = nil | :: of int * list
```

where "`::`" is the infix *cons* constructor. The well-known function `map` might be written as follows using pattern matching:

```
fun map f =
  let fun m nil = nil
        | m (x::r) = (f x) :: (m r)
   in m
  end
```

[2]In Standard ML [MTH90], lists are declared as `datatype 'a list = nil | ::  of 'a * 'a list`. To simplify the presentation, we omit the type variable `'a` by considering only integer lists. All the results described in this chapter easily carry to the polymorphic case.

The new unrolled representation (NUR) in Table 23 can also be expressed by ML concrete datatypes:

```
datatype list2 = OLIST of int * tail2
               | ELIST of tail2

     and tail2 = TNIL
               | TAIL2 of int * int * tail2
```

Here, the data constructors `OLIST` and `ELIST` can be thought of as tags for lists of even length and odd length; they correspond to "`O`" and "`E`" in Table 23.

An efficient `map` function on NUR lists looks like:

```
fun map' f =
   let fun h (OLIST(i,r)) = OLIST(f i, m r)
         | h (ELIST(r)) = ELIST(m r)
       and m TNIL = TNIL
         | m (TAIL2(x,y,r)) = TAIL2(f x, f y, m r)
    in h
   end
```

The test for *nil* (in the pattern-matching for `m`) is done half as often. If `map'` and then `f` are in-line expanded, then the evaluations of `f x` and `f y` can be pipelined.

Simply unrolling the original `map` function, without changing the list representation, is not as attractive because of the extra control and data dependence:

```
fun map_unrolled f =
  let fun m nil = nil
        | m (x::r) =
            case r
             of nil => (f x) :: nil
              | y::s => (f x) :: (f y) :: (m s)
   in m
  end
```

Our static "list unrolling" transformation has the following advantages over the traditional representation, and over runtime "cdr-coding" techniques:

- Loops and recursions on lists are automatically unrolled.
- We avoid many *nil* tests and *cdr* fetches.
- Less memory is used for storing links.
- There is no extra runtime cost (as is incurred by cdr-coding) for handling of encoding bits (except parity testing on the list header).

- Unlike the "cdr-coding" technique that varies with the dynamic behavior of the program (i.e., cons cells have to be adjacent), our method guarantees a $(k-1)/2k$ savings of space usage for long list structures, using k-fold unrolling.

- The interface with garbage collectors is extremely simple, since we use ordinary record structures.

- Because our transformation only relies on the static type information that is usually available in module interfaces, it interacts very well with the module system and separate compilation.

6.2 Compiling with refinement types

In this section, we formally describe the compile-time analysis and present the translation algorithm that automatically transforms program written in OSR notations into one that uses NUR.

First we describe a simple syntactic transformation that gets us partway to our goal. A simple way to implement the NUR is to make the compiler interpret the normal "`::`" constructor *abstractly*, just as Aitken and Reppy deal with their *abstract value constructors* [AR92]. During the compilation, the *constructor function* of "`::`", which takes a data element and a list, and returns a list (the "cons" of the two), can be implemented as the following function `ucons`:

```
fun ucons(x, OLIST (i,r)) = ELIST (TAIL2 (x, i, r))
  | ucons(x, ELIST r) = OLIST (x, r)
```

The *deconstructor function* (also called *projection*) of "`::`", which takes a non-empty list, and returns the head and the tail of the list, can be implemented as the following function `uproj`:

```
fun uproj (OLIST (i,r)) = (i, ELIST r)
  | uproj (ELIST (TAIL2 (i,j,r))) = (i, OLIST (j, r))
```

This approach is extremely easy to implement in most compilers. But it can cause two kinds of runtime inefficiencies when traversing or building a list (such as the `map` function):

- Both `ucons` and `uproj` need to check the length parity of a list each time they are applied, while the old "`::`" requires no check.

- To build a list using `ucons`, one must alternately allocate an `OLIST` cell (e.g., `OLIST(j,r)`) on the heap, discard an `ELIST` cell, then take out `j` and `r`, build an

`ELIST` cons cell (e.g., `ELIST(TAIL2(i,j,r))`), and discard the `OLIST`. This is more expensive than the traditional *cons* operation, which just requires allocating a two element record.

Ideally, the NUR version should avoid the list length parity checks and the alternative allocations of `OLIST` and `ELIST` cells, and thus be more space and time efficient than the OSR version. The function `map'` shown in the previous section behaves this way: it first checks whether the argument is of even length or odd length, then the body `m` of the code "knows" the length parity of its argument.

Now we present a source-to-source program transformation that indeed translates the OSR version of `map` to this more efficient version `map'`. The basic idea is to rely on static analysis to distinguish between lists of even length and odd length at compile time, and to allow functions that take lists as arguments to have three entry points: one dispatch function for list whose length parity is unknown, and one specialized version each for list of even length and odd length. Because the specialized versions have the knowledge of the length parity information, the extra runtime costs of the `ucons` and `uproj` operations can be avoided.

In statically typed languages such as ML, we can keep track of length parity information for most program variables at compile time, because lists are accessed via data constructors and pattern matching only, and they are immune to side-effects[3].

We borrow the *refinement type* inference algorithm of Freeman and Pfenning [FP91, Fre92] by introducing a *refinement* of the `list` type: the type `olist` for odd-length lists and the type `elist` for even-length lists. For example, an empty list is an even-length list; "consing" an element onto an even-length list yields an odd-length list, and "consing" an element onto an odd-length list yields an even-length list. The `map` function always returns a list that has the same length parity as its argument list; "`append`-ing" two lists of same length parity results in an even-length list, and "`append`-ing" two lists of opposite length parity gives an odd-length list, etc.

In the following, we first define the source language (SRC) that uses the traditional OSR notation and the target language (TGT) that uses the NUR representations. Then we present a one-pass translation algorithm that infers the length parity information while at the same time compiling SRC expressions into TGT expressions. Finally we sketch the correctness proof and state the main theorems.

[3] Unlike some functional languages such as Lisp and Scheme, there is no "`setcdr`" operator in ML; list cells in ML are immutable.

$$
\begin{array}{lll}
e & ::= & c^{\tau} \mid x^{\tau} \mid \underline{\mathtt{fn}}\ m \mid e_1\ e_2 \\
 & \mid & \underline{\mathtt{fix}}\ d\ \underline{\mathtt{in}}\ e_1 \\
 & \mid & \mathtt{NIL} \mid \mathtt{CONS}(e_1,e_2) \\
 & & \\
d & ::= & d_1\ \underline{\mathtt{and}}\ d_2 \mid (x^{\tau} = e) \\
m & ::= & m_1 \,[\!]\, m_2 \mid (p \Rightarrow e) \\
p & ::= & x^{\tau} \mid \mathtt{NILP} \mid \mathtt{CONSP}(x,p_1)
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & c \mid x \mid \underline{\mathtt{fn}}\ m \mid e_1 e_2 \mid \underline{\mathtt{fn3}}\ (e_1, e_2, e_3) \\
 & \mid & \underline{\mathtt{fix}}\ d\ \underline{\mathtt{in}}\ e_1 \mid \mathtt{OLIST}(e_1) \mid \mathtt{ELIST}(e_1) \\
 & \mid & \mathtt{TNIL} \mid \mathtt{TAIL1}(e_1,e_2) \mid \mathtt{TAIL2}(e_1,e_2,e_3) \\
 & \mid & \mathtt{ucons}(e_1,e_2) \mid \mathtt{econs}(e_1,e_2) \mid \mathtt{ocons}(e_1,e_2) \\
 & \mid & \mathtt{ufetch}(e_1) \mid \mathtt{efetch}(e_1) \mid \mathtt{ofetch}(e_1) \\
 & & \\
d & ::= & d_1\ \underline{\mathtt{and}}\ d_2 \mid (x = e) \\
m & ::= & m_1 \,[\!]\, m_2 \mid (p \Rightarrow e) \\
p & ::= & x \mid \mathtt{OLISTP}(p_1) \mid \mathtt{ELISTP}(p_1) \\
 & \mid & \mathtt{TNILP} \mid \mathtt{TAIL1P}(x,p_1) \mid \mathtt{TAIL2P}(x,y,p_1)
\end{array}
$$

Figure 28: *left:* The Source Language SRC; *right:* The Target Language TGT

6.2.1 The source language SRC and the target language TGT

Figure 28 gives the syntax of expressions (ranged over by e), declarations (d), matches (m), and patterns (p) for the source language SRC and the target language TGT. We use c to denote constants, x for program variables, and keywords are underlined. The declarations inside a <u>`fix`</u> expression may be mutually recursive functions. For the source language, the data constructors `nil` and `::` under OSR are denoted by `NIL` and `CONS` in expressions, and by `NILP` and `CONSP` in patterns. For the target language, the data constructors under NUR are denoted by `OLIST`, `ELIST`, `TNIL`, `TAIL1` and `TAIL2` in expressions, and by `OLISTP`, `ELISTP`, `TNILP`, `TAIL1P` and `TAIL2P` in patterns. The underlying datatype definition for the NUR version of lists in TGT can be written in ML as follows:

```
datatype list = OLIST of olist
              | ELIST of elist

    and olist = TAIL1 of int * elist

    and elist = TNIL
              | TAIL2 of int * int * elist
```

This is essentially same as the `list2` and `tail2` type defined in Section 6.1. The constructor `TAIL1` and `TAIL1P` are introduced to avoid dealing with tuple expressions in our toy language.[4]

The source language SRC can be thought as a typed intermediate language typical of those used in many compilers. Variables and constants are annotated with types, as in x^τ and c^τ. We assume that the SRC programs are typed using the following very simple (monomorphic) types:

$$\tau ::= \iota \mid \texttt{list} \mid \tau_1 \rightarrow \tau_2$$

where ι denotes base types. This does not mean that our algorithm cannot be applied to polymorphic languages; polymorphic expressions can be easily translated into a monomorphically typed intermediate language by using *representation analysis*, a technique first proposed by Leroy [Ler92] and Peyton Jones [PL91]. But because of space limitations, we have also made this and several other simplifications to ease the presentation:

- We assume that the SRC programs are well-typed according to the standard static typing rules, and that all matches are complete and do not contain redundant patterns.
- Record patterns and expressions are omitted, but pose no problems for our technique.
- Multi-argument functions are also omitted, since their translations are similar to translating their curried versions.

The target language TGT has several other constructs and operators: the term $\underline{\texttt{fn3}}$ (e_1, e_2, e_3) is used to represent a function that takes a list as argument and has three entry points: one (e_1) for lists whose parity is unknown, and one each (e_2 and e_3) for lists of even length and odd length. The one for unknown parity is always a header function that checks the parity dynamically and immediately dispatches to one of the other two entry points. There are three special operators to extract the appropriate entry point from the term $\underline{\texttt{fn3}}$ (e_1, e_2, e_3): `ufetch` to get e_1, `efetch` for e_2, and `ofetch` for e_3. Finally, `ucons` denotes the basic *cons* operation that does not know the length parity of its arguments (the one described at the beginning of this section); `econs` denotes the special operator that *conses* an integer onto an even-length list, and `ocons` *conses* an integer onto an odd-length list. To understand why `econs` and `ocons` can be implemented more efficiently than `ucons`, notice that the expression `ocons`(e_1,`econs`(e_2,e_3)) can be transformed to `TAIL2`(e_1,e_2,e_3), which avoids the parity checking and the allocation of the intermediate odd-length list cell.

[4] In practice, `TAIL1` is a *transparent* data constructor, thus does not require any extra storage to represent [Car84a, App92].

6.2.2 An introduction to refinement types

In this section, we give a brief introduction to the *refinement type* system used in our translation algorithm (see Section 6.2.3). Most of the notation and concepts are directly borrowed from Freeman and Pfenning [FP91, Fre92], since our system is just a simplified version of theirs. Basically we *refine* the `list` type by introducing `elist` for even-length lists and `olist` for odd-length lists. Functions that take lists as arguments can have a more "refined" type, $\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle$, meaning that the result has type ρ_1 if applied to an even-length list, and ρ_2 if applied to an odd-length list. The refinement types (ranged over by ρ) are formally defined as follows:

$$\begin{array}{lcl} \rho & ::= & \tau \mid \bot_\tau \mid \texttt{olist} \mid \texttt{elist} \mid \tau \to \rho_1 \\ & \mid & \langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle \end{array}$$

For every SRC type τ, $\bot_\tau$ represents its bottom refinement type. In the following, we say that a refinement type ρ *refines* an SRC type τ, written $\rho \sqsubset \tau$, if it can be deduced by the rules (R1-R6) in Figure 29. Notice that we only refine the domain of a function type if it is a list type.

We say that a refinement type ρ_1 is a *subtype* of another refinement type ρ_2, written $\rho_1 \leq \rho_2$, if it can be deduced by the rules (S1-S7) in Figure 29. Similarly, Two refinement types ρ_1 and ρ_2 are *equal*, denoted by $\rho_1 \equiv \rho_2$, if $\rho_1 \leq \rho_2$ and $\rho_2 \leq \rho_1$. $\equiv$ is an *equivalence* relation on refinement types. In this chapter, when we talk about a refinement type ρ, we refer to its equivalence class under $\equiv$.

Every SRC type τ has a finite number of refinement types. Moreover, these refinement types form a lattice under the subtype relation "$\leq$", with τ as its top and $\bot_\tau$ as its bottom. For example, the lattice of refinement types for the SRC type `list` is:

$$\begin{tikzcd} & \texttt{list} \arrow[dl, dash] \arrow[dr, dash] & \\ \texttt{elist} \arrow[dr, dash] & & \texttt{olist} \arrow[dl, dash] \\ & \bot_{\text{list}} & \end{tikzcd}$$

Given a refinement type ρ, it is always possible to find out which SRC type it refines. This is denoted by the **top** operation, which is defined by rules (T1-T5) in Figure 29.

Given two refinement types ρ_1 and ρ_2, if $\mathbf{top}(\rho_1) = \mathbf{top}(\rho_2)$, then their *union* type, written as $\rho_1 \vee \rho_2$, is also a refinement type, and is inductively defined by rules (U1-U6). It is easy to see that given two refinement types ρ_1 and ρ_2, if $\rho = \rho_1 \vee \rho_2$, then $\rho_1 \leq \rho$ and $\rho_2 \leq \rho$ (proven by structural induction on ρ).

The operation **apprfty** of applying a refinement type ρ_1 to another refinement type ρ_2 is defined by rules (A1-A4) in Figure 29. This operation is used extensively by the meta operation **applyfun** during the translation (see Section 6.2.3).

Freeman and Pfenning [Fre92, FP91] give an refinement type inference algorithm for typed core-ML. The inference algorithm we used in Section 6.2.3 for the language SRC is just an adaption of their algorithm to the above refinement type system.

6.2.3 The source-to-target translation

The translation of a source language term into the target language is based on the SRC types and the *refinement types* inferred for the term and its subterms. Our translation proceeds by computing refinement types and the translated term simultaneously.

In Figure 30, 31, 32, 33, we present the translation functions for expressions (ExpComp), declarations (DecComp), matches (MatchComp), and patterns (PatComp) in the source language SRC. The function ExpComp takes a SRC expression e, a substitution S (from SRC program variables to TGT expressions), and a refinement type environment Γ as its arguments; and returns a TGT expression e' and the inferred refinement type ρ for e.

Translation of the application $(e_1 e_2)$ is a simple recursive call of ExpComp on e_1 and e_2. Proper coercions must be inserted depending on the inferred refinement types for e_1 and e_2; this is done by the meta-operation $\mathbf{applyfun}(e'_1, \rho_1, e'_2, \rho_2)$ defined in Section 6.2.4.

Translation of abstraction (i.e., $\underline{\texttt{fn}}\ m$) is divided into two cases. If the argument is not a list, this is a simple recursive call to ExpComp.[5] If the argument is a list, the corresponding matches are translated and specialized twice (via MatchComp); once assuming the argument as an even-length list (i.e., *par* is `elist`), and once assuming the argument as an odd-length list (i.e., *par* is `olist`). The two resulting TGT matches (f_e, f_o) correspond to two specialized entry points for lists of even length and odd length. The special entry point f_u for lists of unknown length is built by the **combine** meta-operation: $\mathbf{combine}(f_e, \rho_e, f_o, \rho_o)$ is a TGT function that checks the length parity of its argument first and then dispatch it to special versions f_e or f_o (see Section 6.2.4).

DecComp takes a SRC declaration d, a substitution S, and a refinement type environment Γ; and it returns the TGT declaration, and the resulting refinement type environment from d. The loop inside DecComp computes the fixed point of the refinement types; this is guaranteed to terminate because there are only finitely many refinement types below any given SRC type (a proof of this is given by Freeman [Fre94, Chapter 2]).

[5] Since there are no redundant matches, m must have the form $\underline{\mathrm{fn}}(x^\tau \Rightarrow e)$.

(R1) $\tau \sqsubset \tau$; (R2) $\bot_\tau \sqsubset \tau$;
(R3) $\texttt{elist} \sqsubset \texttt{list}$; (R4) $\texttt{olist} \sqsubset \texttt{list}$;
(R5) $(\tau \to \rho) \sqsubset (\tau \to \tau')$ if $\rho \sqsubset \tau'$;
(R6) $\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle \sqsubset (\texttt{list} \to \tau')$ if $\rho_1 \sqsubset \tau'$ and $\rho_2 \sqsubset \tau'$.

(S1) $\rho \leq \rho$;
(S2) $\rho_1 \leq \rho_3$ if $\rho_1 \leq \rho_2$ and $\rho_2 \leq \rho_3$;
(S3) $\rho \leq \tau$ and $\bot_\tau \leq \rho$ if $\rho \sqsubset \tau$;
(S4) $\tau \to \rho_1 \leq \tau \to \rho_2$ if $\rho_1 \leq \rho_2$;
(S5) $\texttt{list} \to \rho \leq \langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle$ if $\rho \leq \rho_1$ and $\rho \leq \rho_2$;
(S6) $\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle \leq \texttt{list} \to \rho$ if $\rho_1 \leq \rho$ and $\rho_2 \leq \rho$;
(S7) $\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle \leq \langle \texttt{elist} \to \rho_1', \texttt{olist} \to \rho_2' \rangle$ if $\rho_1 \leq \rho_1'$ and $\rho_2 \leq \rho_2'$.

(T1) $\mathbf{top}(\tau) = (\tau)$; (T2) $\mathbf{top}(\bot_\tau) = (\tau)$;
(T3) $\mathbf{top}(\texttt{elist}) = (\texttt{list})$; (T4) $\mathbf{top}(\texttt{olist}) = (\texttt{list})$;
(T5) $\mathbf{top}(\tau \to \rho) = \tau \to (\mathbf{top}(\rho))$;
(T6) $\mathbf{top}(\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle) = \texttt{list} \to (\mathbf{top}(\rho_1))$.

(U1) $\rho \vee \rho' = \rho'$ and $\rho' \vee \rho = \rho'$ if $(\rho \leq \rho')$;
(U2) $(\texttt{olist}) \vee (\texttt{elist}) = \texttt{list}$ and $(\texttt{elist}) \vee (\texttt{olist}) = \texttt{list}$;
(U3) $(\tau \to \rho_1) \vee (\tau \to \rho_2) = \tau \to (\rho_1 \vee \rho_2)$;
(U4) $\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle \vee (\texttt{list} \to \rho) = \langle \texttt{elist} \to \rho_1 \vee \rho, \texttt{olist} \to \rho_2 \vee \rho \rangle$;
(U5) $(\texttt{list} \to \rho) \vee \langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle = \langle \texttt{elist} \to \rho_1 \vee \rho, \texttt{olist} \to \rho_2 \vee \rho \rangle$;
(U6) $\langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_1' \rangle \vee \langle \texttt{elist} \to \rho_2, \texttt{olist} \to \rho_2' \rangle =$
$\langle \texttt{elist} \to \rho_1 \vee \rho_2, \texttt{olist} \to \rho_1' \vee \rho_2' \rangle$;

(A1) $\mathbf{apprfty}(\bot_{\tau_1 \to \tau_2}, \rho_1) = \bot_{\tau_2}$ if $\rho_1 \sqsubset \tau_1$;
(A2) $\mathbf{apprfty}(\tau \to \rho, \bot_\tau) = \bot_{\mathrm{top}(\rho)}$;
(A3) $\mathbf{apprfty}(\tau_1 \to \rho_2, \rho_1) = \rho_2$ if $\rho_1 \sqsubset \tau_1$;
(A4) $\mathbf{apprfty}(\rho, \bot_{\tau\ \mathrm{list}}) = \bot_{\mathrm{top}(\rho_1 \vee \rho_2)}$ and $\mathbf{apprfty}(\rho, \texttt{list}) = \rho_1 \vee \rho_2$ and
$\mathbf{apprfty}(\rho, \texttt{elist}) = \rho_1$ and $\mathbf{apprfty}(\rho, \texttt{olist}) = \rho_2$
where $\rho = \langle \texttt{elist} \to \rho_1, \texttt{olist} \to \rho_2 \rangle$.

Figure 29: Definitions of $\sqsubset$, $\leq$, $\vee$, **top**, and **apprfty** on refinement types

ExpComp $(c^\tau, S, \Gamma) = (c, \tau)$
ExpComp $(x^\tau, S, \Gamma) = (S(x), \Gamma(x))$
ExpComp $(\texttt{NIL}, S, \Gamma) = (\texttt{TNIL}, \texttt{elist})$
ExpComp $(\texttt{CONS}(e_1, e_2), S, \Gamma) =$
    let $(e_1', \texttt{int}) =$ ExpComp (e_1, S, Γ) and $(e_2', \rho) =$ ExpComp (e_2, S, Γ)
        if $\rho = \texttt{elist}$, *cons* and ρ' are respectively `econs` and and `olist`;
        if $\rho = \texttt{olist}$, *cons* and ρ' are respectively `ocons` and `elist`;
        otherwise, *cons* is `ucons` and $\rho' = \rho$;
    in $(cons(e_1', e_2'), \rho)$

ExpComp $(\underline{\texttt{fn}}\ m^{\text{list}\to\tau}, S, \Gamma) =$
    let $(m_o', \rho_o) =$ MatchComp $(m, S, \Gamma, \texttt{olist})$
        $(m_e', \rho_e) =$ MatchComp $(m, S, \Gamma, \texttt{elist})$
        f_u, f_e, f_o be new program variables and $e' = \underline{\texttt{fn3}}\ (f_u, f_e, f_o)$;
        $d' = (f_u = \mathbf{combine}(f_e, \rho_e, f_o, \rho_o))\ \underline{\texttt{and}}\ (f_e = \underline{\texttt{fn}}\ m_e')\ \underline{\texttt{and}}\ (f_o = \underline{\texttt{fn}}\ m_o')$
    in $(\underline{\texttt{fix}}\ d'\ \underline{\texttt{in}}\ e', \langle \texttt{elist} \to \rho_e, \texttt{olist} \to \rho_o \rangle)$

ExpComp $((\underline{\texttt{fn}}\ (x^\tau \Rightarrow e)), S, \Gamma) =$
    let $(e', \rho) =$ ExpComp $(e, S \pm \{x \mapsto x\}, \Gamma \pm \{x \mapsto \tau\})$
    in $(\underline{\texttt{fn}}\ (x \Rightarrow e'), \tau \to \rho)$

ExpComp $(e_1 e_2, S, \Gamma) =$
    let $(e_1', \rho_1) =$ ExpComp (e_1, S, Γ) and $(e_2', \rho_2) =$ ExpComp (e_2, S, Γ)
    in $\mathbf{applyfun}(e_1', \rho_1, e_2', \rho_2)$

ExpComp $(\underline{\texttt{fix}}\ d\ \underline{\texttt{in}}\ e, S, \Gamma) =$
    let $(d', \Gamma_1) =$ DecComp (d, S, Γ) and $S_1 = \{x \mapsto x \mid x \in \text{Dom}(\Gamma_1)\}$
        $(e', \rho) =$ ExpComp $(e, S \pm S_1, \Gamma \pm \Gamma_1)$
    in $(\underline{\texttt{fix}}\ d'\ \underline{\texttt{in}}\ e', \rho)$

Figure 30: Translation of Expressions

The argument *par* in the MatchComp function represents the length parity (either `elist` or `olist`) of the argument in the match m. A simple SRC rule $p \Rightarrow e$ is *compatible* with *par* if p is *compatible* with *par*. The *compatibility* between a SRC pattern and a parity is inductively defined as follows: variable pattern x is compatible with both `elist` and `olist`; `NILP` is only compatible with `elist`; $\texttt{CONSP}(x, p)$ is compatible with `elist` (or `olist`) if and only if p is compatible with `olist` (or `elist`).

During the translation of a SRC match, the resulting refinement types for different cases may be different. We use the **coerce** meta-operation defined in Section 6.2.4 to coerce all

DecComp $(d, S, \Gamma) =$

let assume d is $(x_1^{\tau_1} = e_1)$ <u>`and`</u> ... <u>`and`</u> $(x_k^{\tau_k} = e_k)$,
and $\Gamma_{\text{result}} = \{x_i \mapsto \bot_{\tau_i} \mid i = 1, ..., k\}$;
loop $\Gamma_{\text{start}} = \Gamma_{\text{result}}$;
$(e_i', \rho_i) =$ ExpComp $(e_i, S, \Gamma \pm \Gamma_{\text{start}})$ where $i = 1, ..., k$;
$\Gamma_{\text{result}} = \{x_i \mapsto \rho_i \mid i = 1, ..., k\}$
until $(\Gamma_{\text{start}} = \Gamma_{\text{result}})$
in $((x_1 = e_1')$ <u>`and`</u> ... <u>`and`</u> $(x_k = e_k')$, $\Gamma_{\text{result}})$

Figure 31: Translation of Declarations

MatchComp $(m, S, \Gamma, par) =$

let assume $\{p_i \Rightarrow e_i \mid i = 1, ..., k\}$ are rules in "m" that are *compatible* with par;
$(p_i', S_i, \Gamma_i) =$ PatComp (p_i, par) for $i = 1, ..., k$
$(e_i', \rho_i) =$ ExpComp $(e_i, S \pm S_i, \Gamma \pm \Gamma_i)$ for $i = 1, ..., k$
$\rho = \rho_1 \vee ... \vee \rho_k$ and $e_i'' = \mathbf{coerce}(e_i', \rho_i, \rho)$ for $i = 1, ..., k$
in $((p_1' \Rightarrow e_1'') [\!] ... [\!] (p_k' \Rightarrow e_k''), \rho)$

Figure 32: Translation of Matches

PatComp $(\texttt{NILP}, \texttt{elist}) = (\texttt{TNIL}, \emptyset, \emptyset)$;
PatComp $(x, \texttt{elist}) = (x, \emptyset, \{x \mapsto \texttt{elist}\})$;
PatComp $(x, \texttt{olist}) = (\texttt{TAIL1P}(y, z), \{x \mapsto \texttt{TAIL1}(y, z)\}, \{x \mapsto \texttt{olist}\})$
where y and z are new program variables;

PatComp $(\texttt{CONSP}(x, p), \texttt{olist}) = (\texttt{TAIL1P}(y, p'), S \pm \{x \mapsto y\}, \Gamma \pm \{x \mapsto \texttt{int}\})$
where $(p', S, \Gamma) =$ PatComp $(p, \texttt{elist})$ and y is a new program variable;

PatComp $(\texttt{CONSP}(x, p), \texttt{elist}) = (\texttt{TAIL2P}(y, z, p'), S \pm \{x \mapsto y\}, \Gamma \pm \{x \mapsto \texttt{int}\})$
where $(\texttt{TAIL1P}(z, p'), S, \Gamma) =$ PatComp $(p, \texttt{olist})$
and y is a new program variable.

Figure 33: Translation of Patterns

different representations (with refinement types ρ_1, ..., ρ_k) into a unified one (the *union* type of all ρ_i).

The PatComp function translates a SRC pattern p into the TGT pattern based on the parity assumption about p; it also derives a substitution and a refinement type environment for all variables in p.

SRC expression $\quad e = \underline{\texttt{fix}}\ \ (m{=}\underline{\texttt{fn}}\ (\texttt{NILP} \Rightarrow \texttt{NIL})$
$\qquad\qquad [\!]\,(\texttt{CONSP}(x, r) \Rightarrow \texttt{CONS}(x+1, m\ r)))\ \underline{\texttt{in}}\ m$

TGT expression $\quad e' = \underline{\texttt{fix}}\ \ m' = d'\ \underline{\texttt{in}}\ m'$ where
$d' = \underline{\texttt{fix}}\ \ ((f_u = e_u)\underline{\texttt{and}}(f_e = e_e)\underline{\texttt{and}}(f_o = e_o))\ \underline{\texttt{in}}\ (\underline{\texttt{fn3}}\ \ (f_u, f_e, f_o));$
$e_u = \underline{\texttt{fn}}\ \ (\texttt{OLISTP(x)} \Rightarrow \texttt{OLIST}(f_o\ x))[\!](\texttt{ELISTP(y)} \Rightarrow \texttt{ELIST}(f_e\ y));$
$e_o = \underline{\texttt{fn}}\ \ (\texttt{TAIL1P}(x, r) \Rightarrow \texttt{ocons}(x+1, ((\texttt{efetch}(m'))\ r)));$
$e_e = \underline{\texttt{fn}}\ \ (\texttt{TNILP} \Rightarrow \texttt{TNIL})[\!](\texttt{TAIL2P}(x, y, r) \Rightarrow e'_e))$
$e'_e = \texttt{econs}(x+1, ((\texttt{ofetch}(m'))\ (\texttt{TAIL1}(y, r))))$

Figure 34: Example on the `map` function

For example, Figure 34 shows the target expression from translating a simplified version of the `map` function (shown rather than as in Section 6.1). This function maps the "+1" function to a list. Our algorithm infers that m has the refinement type $\langle \texttt{elist} \rightarrow \texttt{elist}, \texttt{olist} \rightarrow \texttt{olist}\rangle$ and yield the target expression e'. Notice that in the real implementation, the expression $\texttt{efetch}(m')$ (inside e_o) and $\texttt{ofetch}(m')$ (inside e_e) will be contracted into f_e and f_o, and the application of f_o to $\texttt{TAIL1}(y, r)$ (inside e_e) will be inline-expanded, and then the consecutive application of `econs` and `ocons` in e_e will be contracted into $\texttt{TAIL2}(x+1, y+1, f_e\ r)$, which is exactly the form we desired in Section 6.1.

Notice that given an SRC expression e of type τ, the above translation algorithm returns a TGT expression e' and a refinement type ρ for e. This "ρ" happens to be the type of e' under $\vdash_{\mathrm{TGT}}$, where "$\vdash_{\mathrm{TGT}}$" is a set of (simple monomorphic) typing rules for TGT expressions by ρ defined above extended with "$\texttt{elist} \rightarrow \rho_1$" and "$\texttt{olist} \rightarrow \rho_1$". For example, the typing rule for the `econs` expression will be "if $\Gamma \vdash_{\mathrm{TGT}} e_1 : \texttt{int}$ and $\Gamma \vdash_{\mathrm{TGT}} e_2 : \texttt{elist}$; then $\Gamma \vdash_{\mathrm{TGT}} \texttt{econs}(e_1, e_2) : \texttt{olist}$;" the rule for $\underline{\texttt{fn3}}\ (e_1, e_2, e_3)$ will be "if $\Gamma \vdash_{\mathrm{TGT}} e_1 : \texttt{list} \rightarrow \rho$ and $\Gamma \vdash_{\mathrm{TGT}} e_2 : \texttt{elist} \rightarrow \rho_e$ and $\Gamma \vdash_{\mathrm{TGT}} e_3 : \texttt{olist} \rightarrow \rho_o$ and $\rho = \rho_e \vee \rho_o$; then $\Gamma \vdash_{\mathrm{TGT}} \underline{\texttt{fn3}}\ (e_1, e_2, e_3) : \langle \texttt{elist} \rightarrow \rho_e, \texttt{olist} \rightarrow \rho_o\rangle$. These typing rules are straightforward, and are thus omitted here.

$$\begin{array}{l}
\mathbf{combine}(e_1', \rho_1, e_2', \rho_2) = \\
\qquad \underline{\mathtt{fn}}\ (\mathtt{OLIST}\ x \Rightarrow \mathbf{coerce}(e_1' x, \rho_1, \rho)) [\!] (\mathtt{ELIST}\ y \Rightarrow \mathbf{coerce}(e_1' y, \rho_2, \rho)) \\
\qquad \text{where } \rho = (\rho_1 \vee \rho_2) \text{ and } x{,}y \text{ are new program variables}
\end{array}$$

$$\begin{array}{l}
\mathbf{coerce}(e', \rho, \rho) = e'; \\
\mathbf{coerce}(e', \bot_\tau, \rho) = e'; \\
\mathbf{coerce}(e', \mathtt{elist}, \mathtt{list}) = \mathtt{ELIST}(e'); \\
\mathbf{coerce}(e', \mathtt{olist}, \mathtt{list}) = \mathtt{OLIST}(e');
\end{array}$$

$$\mathbf{coerce}(e', \tau \to \rho_1, \tau \to \rho_2) = \underline{\mathtt{fn}}\ (x \Rightarrow \mathbf{coerce}((e'\ x), \rho_1, \rho_2));$$

$$\begin{array}{l}
\mathbf{coerce}(e', \langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle, \mathtt{list} \to \rho') = \\
\qquad \underline{\mathtt{fn}}\ (x \Rightarrow \mathbf{coerce}(((\mathtt{ufetch}(e'))\ x), \rho_1 \vee \rho_2, \rho'));
\end{array}$$

$$\begin{array}{l}
\mathbf{coerce}(e', \mathtt{list} \to \rho', \langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle) = \underline{\mathtt{fix}}\ d'\ \underline{\mathtt{in}}\ (\underline{\mathtt{fn3}}\ (f_u, f_e, f_o)) \\
\qquad \text{where } f_u, f_e, f_o \text{ are new program variables} \\
\qquad \text{and } d' = (f_u = m_u')\ \underline{\mathtt{and}}\ (f_e = m_e')\ \underline{\mathtt{and}}\ (f_o = m_o') \\
\qquad \text{and } m_u' = (\mathbf{combine}(f_e, \rho_1, f_o, \rho_2)) \\
\qquad \text{and } m_e' = (x \Rightarrow \mathbf{coerce}((e'\ x), \rho', \rho_1)) \\
\qquad \text{and } m_o' = (x \Rightarrow \mathbf{coerce}((e'\ x), \rho', \rho_2));
\end{array}$$

$$\begin{array}{l}
\mathbf{coerce}(e', \langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle, \langle \mathtt{elist} \to \rho_1', \mathtt{olist} \to \rho_2' \rangle) \\
\qquad\qquad\qquad\qquad = \underline{\mathtt{fix}}\ d'\ \underline{\mathtt{in}}\ (\underline{\mathtt{fn3}}\ (f_u, f_e, f_o)) \\
\qquad \text{where } f_u, f_e, f_o \text{ are new program variables} \\
\qquad \text{and } d' = (f_u = m_u')\ \underline{\mathtt{and}}\ (f_e = m_e')\ \underline{\mathtt{and}}\ (f_o = m_o') \\
\qquad \text{and } m_u' = (\mathbf{combine}(f_e, \rho_1', f_o, \rho_2')) \\
\qquad \text{and } m_e' = (x \Rightarrow \mathbf{coerce}(((\mathtt{efetch}(e'))\ x), \rho_1, \rho_1')) \\
\qquad \text{and } m_o' = (x \Rightarrow \mathbf{coerce}(((\mathtt{ofetch}(e'))\ x), \rho_2, \rho_2'))
\end{array}$$

$$\mathbf{applyfun}(e_1', \rho_1, e_2', \rho_2) = (e_1'\ (\mathbf{coerce}(e_2', \rho_2, \tau_2)), \mathbf{apprfty}(\rho_1, \rho_2)) \text{ if } \rho_1 = \tau_2 \to \rho;$$

$$\begin{array}{l}
\mathbf{applyfun}(e_1', \rho_1, e_2', \rho_2) = \\
\qquad ((\mathit{fetch}(e_1'))\ e_2', \mathbf{apprfty}(\rho_1, \rho_2)) \text{ if } \rho_1 = \langle \mathtt{elist} \to \rho_1', \mathtt{olist} \to \rho_2' \rangle, \\
\qquad \mathit{fetch} \text{ is respectively } \mathtt{efetch}, \mathtt{ofetch}, \text{ or } \mathtt{ufetch} \text{ if } \rho_2 \text{ is } \mathtt{elist}, \mathtt{olist}, \text{ or others;}
\end{array}$$

$$\mathbf{applyfun}(e_1', \rho_1, e_2', \rho_2) = (e_1' e_2', \mathbf{apprfty}(\rho_1, \rho_2)) \text{ for all other cases.}$$

Figure 35: Definitions of **combine**, **coerce**, and **applyfun**

6.2.4 The definition of several meta-operations

In this section, we formally define the three meta operations **coerce**, **combine** and **applyfun** used in the translation algorithm in Figure 30 through Figure 33.

Figure 35 gives the definition of **combine**. Given two target expressions e'_1 and e'_2, and two refinement types ρ_1 and ρ_2, the operation **combine**(e'_1,ρ_1,e'_2,ρ_2), constructs a dispatch TGT function that calls e'_1 or e'_2 respectively depending on the length parity of its argument.

It is occasionally necessary to introduce code to coerce the result of a term from one representation to another. Given a target expression e', two refinement types ρ_1 and ρ_2 such that $\rho_1 \leq \rho_2$, then the operation **coerce**(e', ρ_1, ρ_2), which returns a new target expression, is defined in Figure 35. Notice that **combine** and **coerce** are mutually recursive.

The **applyfun** operation inserts appropriate coercions for function applications. Given two TGT expressions e'_1 and e'_2, and two refinement types ρ_1 and ρ_2, the operation **applyfun**($e'_1, \rho_1, e'_2, \rho_2$), which returns a TGT expression and a refinement type, is also defined in Figure 35.

6.2.5 Correctness of the translation

The type and semantic correctness of our translation can be proven using a technique similar to that of Leroy [Ler92]. Here we only sketch the proof method and state the main theorem. We use $\vdash_{\mathrm{SRC}}$ to denote the type deduction rule for SRC, and $\vdash_{\mathrm{TGT}}$ to denote the refinement type deduction rule for TGT. More specifically, suppose TE is a SRC type environment (from variables to SRC types τ), and Γ is a refinement type environment, then $\mathrm{TE} \vdash_{\mathrm{SRC}} e : \tau$ means that e is well-typed in TE under $\vdash_{\mathrm{SRC}}$, and $\Gamma \vdash_{\mathrm{TGT}} e' : \rho$ means e' has the refinement type ρ in Γ under $\vdash_{\mathrm{TGT}}$. We also define the (straightforward) *call-by-value* operational semantics $\mathrm{VE} \vdash e \xrightarrow{s} v$ for the source language SRC, and $\mathrm{VE}' \vdash e' \xrightarrow{t} v'$ for the target language TGT, where VE and VE$'$ are *value environments* (from variables to values). A notion of *equivalence* between the typed SRC *values* (which corresponds to the OSR) and the typed TGT *values* (which corresponds to the NUR) is defined, written as $v : \tau \approx v' : \rho$. This $\approx$ relation is only defined for the pair of values when v has type τ, v' has type ρ, and $\rho \sqsubset \tau$. $\mathrm{VE} : \mathrm{TE} \approx \mathrm{VE}' : \Gamma$ is used to denote that for every $x \in \mathrm{Dom}(\mathrm{VE})$, such that $\mathrm{VE}(x) : \mathrm{TE}(x) \approx \mathrm{VE}'(x) : \Gamma(x)$. The type and semantic correctness of our translation algorithm now can be stated by the following proposition, which is proven by structural induction:

Proposition 6.2.1 *Given a* SRC *expression* e*, a* SRC *type environment* TE*, and a refinement type environment* Γ*, such that* $\mathrm{TE} \vdash_{\mathrm{SRC}} e : \tau$ *is valid, and* $\Gamma(x) \sqsubset \mathrm{TE}(x)$ *for every*

$x \in \mathrm{Dom}(\mathrm{TE})$, then $\mathrm{ExpComp}\ (e, ID, \Gamma) = (e', \rho)$ *will succeed; moreover, (1)* $\rho \sqsubset \tau$*; (2)* $\Gamma \vdash_{\mathrm{TGT}} e' : \rho$ *is valid; (3) Given a value environment* VE *under* $\xrightarrow{s}$ *and a value environment* VE' *under* $\xrightarrow{t}$*, if* $\mathrm{VE} : \mathrm{TE} \approx \mathrm{VE}' : \Gamma$*, and* $\mathrm{VE} \vdash e \xrightarrow{s} v$*, then, there exists a value* v' *such that* $\mathrm{VE}' \vdash e' \xrightarrow{t} v'$ *and* $v : \tau \approx v' : \rho$*.*

6.3 Compiling with multiple continuations

```
fun filter' (p, c) =
  let fun f_u(OLIST(x,r), ce, co) = f_o(x, r, ce, co)
        | f_u(ELIST r, ce, co) = f_e(r, ce, co)

      and f_o(x, r, ce, co) = if (p x)
            then let fun ke(z) = co(econs(x,z))
                     fun ko(z) = ce(ocons(x,z))
                  in f_e(r, ke, ko)
                 end
            else f_e(r, ce, co)

      and f_e(TNIL, ce, co) = ce(TNIL)
        | f_e(TAIL2(x, y, r), ce, co) = if (p x)
            then let fun ke(z) = co(econs(x,z))
                     fun ko(z) = ce(ocons(x,z))
                  in f_o(y, r, ke, ko)
                 end
            else f_o(y, r, ce, co)

   in c(f_u)
  end
```

Figure 36: Pseudo CPS code for `filter`

The algorithm presented in Section 6.2 successfully translates a program written in OSR notation into one in NUR. In most cases, it pleasantly eliminates the costs of extra length parity checks and alternating allocations of `OLIST` and `ELIST` cells incurred by `ucons` and `uproj`. To demonstrate this, we tried our algorithm on 15 frequently used list-processing library functions.[6] Among these 15 cases, our algorithm successfully eliminates all the extra costs of `ucons` and `uproj` for 14 of them. The only exception is the `filter` function, which selects only those elements of a list matching a given predicate. The problem with `filter`

[6] Here is a list of these functions: `hd`, `tl`, `length`, `append`, `rev`, `map`, `fold`, `revfold`, `app`, `revapp`, `nthtail`, `nth`, `exists`, `last`, and `filter`. They are mostly taken from the initial basis of the Standard ML of New Jersey compiler [AM91].

is that even if we know the length parity of its argument, we still do not know the length parity of its result.

Here is the `filter` function:

```
fun filter p =
  let fun f nil = nil
        | f (x::r) = if (p x) then x::(f r) else f r
   in f
  end
```

It turns out that this problem can be easily solved in the continuation-passing style (CPS) framework [Ste78, App92], because we can specialize the return continuation on the length parity of the result, and make it have multiple entry points also. The idea is as follows: when we are converting a SRC expression e into CPS, we use a method similar to that of Section 6.2 to infer the refinement type for e; whenever we are not sure about the length parity of a list expression, we duplicate its return continuation into one accepting an even-length list and another accepting an odd-length list. For example, the source-language `filter` function is CPS-converted into the `filter'` function in Figure 36 (written using pseudo-CPS notation in ML). Here, `c`, `ce`, `co`, `ke`, `ko` are the continuation variables. The length parity of the variable `z` (i.e., the return result of `f_e` and `f_o`) is statically unknown, but after duplicating the return continuation (`ke`, `ko`), `z` is then assumed as even-length list and odd-length list in each, thus no parity check is necessary, and the more efficient version (`econs` and `ocons`) of "`ucons`" can be used. The heap allocation of intermediate `OLIST` cons cell is still avoided because of *representation analysis* (see Chapter 3 and Leroy [Ler92]).

It is likely, however, that this transformation will only improve performance if the underlying compiler uses *representation analysis* (see Chapter 3 and Leroy [Ler92]), and is very sophisticated about closure construction and register usage (see Chapter 4). Otherwise, the extra cost of closure creations could outweigh the elimination of the `cons` operations.

Note, however, that though there are extra costs of testing for `ELIST/OLIST`, there are fewer tests for *nil*. The result (as shown in the next section) is that the NUR version of filter is about as fast as the OSR version, even without the specialized CPS version of our analysis.

6.4 Experiments

We have implemented the algorithm described in Section 6.2 in an experimental version of the Standard ML of New Jersey compiler (SML/NJ) [AM91, App92]. Because the compiler uses continuation-passing style as its intermediate language, the multiple-continuation approach described in Section 6.3 can be easily added (this has not been done yet). The SML/NJ compiler supports representation analysis (see Chapter 3 and [Ler92]), so intermediate odd-length lists are represented by unboxed records, which normally stay in registers; this makes the specialized versions (for even-length and odd-length lists) of the `ucons` and `uproj` operations require less memory allocation.

Table 24: Performance of the Benchmark Programs

	Lists Allocated (mega-words)			**CPU Time** (seconds)			**Code Size** (kilo-bytes)		
Benchmark	**OSR**	**NUR**	**Ratio**	**OSR**	**NUR**	**Ratio**	**OSR**	**NUR**	**Ratio**
Life	0.71	0.71	x1.00	1.19	0.91	x0.77	13.9	54.5	x3.9
Ray	13.89	13.89	x1.00	22.71	20.59	x0.91	44.0	77.5	x1.76
Quicksort	1.81	1.35	x0.75	0.98	0.87	x0.89	3.0	8.4	x2.8
Samsort	1.81	1.30	x0.71	1.03	0.88	x0.85	2.5	5.3	x2.1
Intset	0.54	0.36	x0.67	0.63	0.51	x0.81	4.1	12.3	x3.0
MMap	1.20	0.80	x0.67	1.73	1.29	x0.75	3.5	8.7	x2.5

6.4.1 Avoiding code explosion

Translating from OSR to NUR involves function specialization and recursion unrolling. If a function takes n list arguments with a k-way unrolled representation, we need k^n entry points.

Though most list-processing functions take only one list argument, for functions that take multiple list arguments (e.g., the `append` function which concatenates two lists), an exponential blowup is a serious concern.

To avoid the blowup, we use a system parameter called `unroll-level` to control the depth of specialization and unrolling. For example, suppose function `f` has five arguments that are of type `list`, and suppose `unroll-level` is 2, then the compiler will only specialize the first two arguments. The slight runtime cost for not specializing some arguments is not a problem in practice because most of the frequently used list functions take only one or two list arguments. For example, among the 15 functions in the initial "List" library in the SML/NJ compiler, 14 of them have only one list argument, and only the `append` function has two list arguments.

6.4.2 Measurements

Our technique guarantees a 25% savings in memory usage for 2-way unrolling (long) lists. But execution-time savings will be achieved only if most of the `ucons` and `uproj` operations can be removed.

To demonstrate the savings of execution time, we have compared the performance of several benchmarks under the standard representation (OSR) and the unrolled representation (NUR). Our benchmarks include: **Life**, the game of Life implemented using lists written by Reade [Rea89] (see Table 2 in Chapter 2); **Ray**, a simple ray tracer (see Table 2 in Chapter 2); **Quicksort**, sorting a list of 20000 real numbers using the quicksort algorithm (taken from Paulson [Pau91]); **Samsort**, sorting a list of 2000 real numbers using a variation of mergesort algorithm (taken from Paulson [Pau91]); **Intset**, "set" operations on sets of integers implemented with sorted lists; **MMap**, several runs of the `map` function on a long list.

Table 24 gives the total size of lists allocated during execution, the program execution time on a DECstation 5000/240, and the code size increase, with the `unroll-level` set as 2, and each *cons* cell 2-way unrolled. In all cases, the NUR version allocates less memory[7] and runs faster (11%-25%) than the OSR version. Notice that the **Life** benchmark frequently calls the `filter` function, and several functions that have more than two list arguments (thus some of them are not specialized). Because of this, the total size of lists allocated for NUR is about the same as OSR; but because NUR requires many fewer memory references and *nil* tests, it runs much faster than OSR (about 23%). Although the code size did not explode because of the `unroll-level` parameter, it does increase by a factor of 1.76 to 3.9. We are currently exploring ways of cutting down the code size for NUR, while still maintaining its performance gain. One problem of our current implementation of NUR is that our compiler does not have a good dead code detection algorithm, we believe that a more refined implementation can achieve more code sharing and produce much smaller code.

6.5 Related work

Cdr-coding techniques were first proposed in the early 70's by researchers at MIT and Xerox [Han69, Gre77, Cla76, CG77, BC79, Bob75]. While these schemes differ from each other on the encoding methods, they all rely on the hardware support from microcoded

[7] NUR allocates 33% less memory than OSR on certain benchmarks, because unlike in Table 23, each *cons* cell in our compiler contains an extra descriptor word.

Lisp machines [WM81, Deu73] to alleviate the high costs incurred by the runtime encoding bits. Since modern machines tend not to offer these kinds of special hardware support, the runtime cdr-coding technique quickly became obsolete in the 1980's. The "static cdr-coding" technique presented in this chapter is a simple compile-time method for doing list compaction. It is attractive for modern machines because it does not require any runtime encoding bits at all.

Li and Hudak [LH86] proposed a cdr-coding scheme for list compaction under parallel environments. When several lists are being constructed simultaneously from the same heap, the non-contiguous nature of the cells being allocated might eliminate the opportunity for compaction under traditional cdr-coding techniques. To overcome this, they also represent list as linked (fixed length) vectors, and do the "consing" by pre-allocating a vector first and consecutively filling in later elements. This technique still relies on runtime encoding bits to distinguish the state of each vector cell (i.e., filled or empty), and is thus quite expensive. Our static cdr-coding method, on the other hand, exploits compile-time analysis to eliminate most runtime checks; at the same time, it poses no more problem in parallel environments than does ordinary allocation.

On the side of statically typed languages, Hall [Hal94] has presented a list compaction technique for Haskell [HJ*et al*92]. In her scheme, lists can be represented as the old standard representation (OSR) at one place, and in an optimized representation at another place. The optimized representation in her scheme is adapted for lazy languages where the tail of a list may not yet be evaluated, and thus its length parity cannot be known. Therefore, she must put the "extra" elements at the end of the list, making the test on each unrolled iteration more complicated. In part because of this, her scheme requires more runtime checks than ours. Hall's analysis (based on Hindley-Milner type inference) determines where to insert coercions between different representations. But the representations themselves must already be used in the programs. In effect, this means that library functions must be explicitly programmed using several different representations, and programs will be improved only if they use the library functions.

The idea of using special and more efficient representations for frequently used data objects (through type-based analysis) is originally from Leroy [Ler92] and Peyton Jones [PL91]. Both propose a type-based program transformation scheme that allows objects with monomorphic ML types to use special unboxed representations. When an unboxed object interacts with a boxed polymorphic object, appropriate coercions are inserted. But as mentioned by Leroy [Ler92], their representation analysis techniques do not work well with ML's recursive data types, such as the list type. This is because the coercion between the unboxed and boxed representation for lists is often rather expensive (i.e., has costs proportional to the list length). Our translation scheme, on the other hand, allows commonly used list objects to uniformly use more efficient unrolled representations, whether they have a monomorphic type or not—though the *element values* must still use the standard (single-word) representation. Coercions among representations for even-length lists, odd-length lists, and lists whose length parity is unknown, are quite cheap.

The refinement type system used in Section 6.2 is a much simplified version of Freeman and Pfenning's refinement type system [FP91, Fre92]. While the underlying framework and type inference algorithm are quite similar, our motivation is rather different. In their system, the refinement type is declared by the programmer, and the refinement type information is used to detect program errors at compile time. The reason that we use the refinement types, on the other hand, is to do compile-time program transformations and optimizations. The refinement type declaration used in our scheme is embedded in the compiler, and is completely hidden from programmers.

As in Wadler's *views* mechanism [Wad87], the standard and unrolled representations of lists in our scheme can be linked together by a pair of **in** and **out** functions (e.g., the "`ucons`" and "`uproj`" function in Section 6.1). We introduce unrolled representation for lists mainly to improve the space and time efficiencies for programs using lists, while Wadler uses his *views* mechanism to hide the representations of concrete data types and reconcile pattern matching with data abstraction.

6.6 Summary

We have presented a "list unrolling" technique that allows a more compact and efficient list representation for statically typed languages. Our "list unrolling " technique generalizes to depth-d unrolling of k-ary trees with only k^{dm} entry points necessary for any function with m tree arguments. Reasoning about lists (and trees) in these languages are easier than about pointers in other languages because lists (and trees) are accessed only via data constructors and pattern matching. The higher-level of language abstraction permits the compiler to automatically transform a program into one that uses more efficient data representations, and that permits loop unrolling by eliminating certain control and data dependences.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The ultimate goal of software research is to find the right programming model in which one can write the "best quality" (reliable, efficient, portable, etc.) program with the least amount of time and effort.

Many language researchers believe that the most effective way to achieve this goal is to write programs in higher-level languages. Although higher-level languages do provide a simpler and cleaner programming model, they are very difficult to implement very efficiently.

This dissertation uses Standard ML (SML)—a statically typed functional language—as the test bed to explore new compilation techniques for modern higher-level languages. SML poses tough challenges to efficient implementations: frequent function calls, polymorphic types, recursive data structures, higher-order functions, and first-class continuations. We presented three new compilation techniques that meet these challenges by exploiting some of the higher-level language features in SML:

- *Type-directed compilation* exploits the use of compile-time type information to optimize data representations and function calling conventions. By inserting coercions at each type instantiation and abstraction site, data objects in SML can use the same unboxed representations as in C, even with the presence of polymorphic functions. Measurements show that a simple set of type-based optimizations improve the performance of the non-type-based compiler by about 19% on a DECstation 5000/240.

- *Space-efficient closure representations* utilizes the compile-time control and data flow information to optimize closure representations. By extensive closure sharing and allocating as many closures in registers as possible, the new *closure conversion* algorithm achieves very good asymptotic space usage, and improves the performance of the old compiler by about 14% on a DECstation 5000/240, *even without using a stack.* Further empirical and analytic studies show that the execution cost of stack-allocated and heap-allocated activation records is similar, but heap allocation is simpler to implement and allow very efficient first-class continuations.

- *Unrolling lists* takes advantage of the higher-level language abstraction in SML to support more efficient representations for lists. By representing each *cons* cell using multiple *car* fields and one *cdr* field, the *unrolled list* reduces the memory used for links and significantly shortens the length of control-dependence and data-dependence chains in operations on lists.

Table 25: Combined performance improvement (execution time)

Benchmark	Basis	nrp+occ	rep+occ	nrp+ncc	rep+ncc
BHut	31.9	1.00	0.96	0.88	0.69
Boyer	2.6	1.00	0.95	0.91	0.85
Sieve	35.6	1.00	1.00	0.86	0.85
KB-Comp	8.3	1.00	0.90	0.82	0.73
Lexgen	12.5	1.00	0.92	0.91	0.84
Yacc	5.3	1.00	0.84	0.91	0.80
Life	11.8	1.00	0.11	0.91	0.10
Simple	25.2	1.00	0.88	0.90	0.73
Ray	23.6	1.00	0.86	0.98	0.84
VLIW	16.7	1.00	0.96	0.84	0.73
Average		1.00	0.84	0.89	0.72

Table 25 summarizes the combined improvement of the type-directed compilation technique (**rep**) and the new space-efficient closure conversion algorithm (**ncc**). The code generated by the new compiler (**rep+ncc**) is on average about 28% faster (over 10 benchmarks) than that generated by the old SML/NJ compiler (**nrp+occ**), which neither supports type-directed compilation (i.e., **nrp**) nor uses the new closure algorithm (i.e., **occ**).

Although these techniques are developed in the context of SML, they also apply to other languages that share some of SML's properties. For example, the type-directed compilation technique (Chapter 3) should be useful for all statically typed languages, especially those using the Hindley-Milner polymorphic type system. The closure analysis technique in Chapter 4 is presented in the context of continuation-based compilers, but it can also be applied to compilers that do not use CPS as the intermediate language. Both safely linked closures and good use of callee-save registers are essential in building efficient compilers for languages with closures. The unrolling list technique (Chapter 6) may not work well for lazy functional languages, because under lazy evaluation, it is difficult to reason about the list length at compile time. But any strict languages that use lists frequently can benefit great from the new unrolled representation, especially on modern superscaler machines.

7.2 Future work

While the new techniques presented in this dissertation have significantly improved the performance of SML programs, more work remains to be done to actually make SML programs run as efficiently as C or C++ programs.

The type-directed compilation technique makes it possible to use many efficient data representations in SML, however, only a very small subset of them have been implemented and evaluated in this dissertation. It will be very valuable to support the flat unboxed representation with sophisticated descriptors (as shown in Figure 5). To avoid expensive coercions, our current implementation still uses the standard boxed representation for all recursive data structures. We are currently still looking for a scheme that supports unboxed representations (for recursive data structures) without incurring much runtime overhead.

The type-directed compilation technique presented in Chapter 3 implements polymorphic functions by inserting coercions at each type instantiation and abstraction site. Another way to implement polymorphic function efficiently is to use compile-time type specializations. To support general specialization of polymorphic functions (or even cross module boundaries), we may have to use a lambda language with explicit type abstractions and type applications. How to translate the SML module lambda language into this kind of typed lambda calculus remains to be a research problem.

There are many possible extensions that can be done along with the new closure analysis presented in Chapter 4. For example, Section 4.5.3 presented a way of using different numbers of callee-save registers to represent continuation closures. This technique has not been implemented and evaluated yet. Another extension would be to do register spilling analysis in the closure conversion phase. Unlike in the machine code generator, the layout for each closure is known in the closure conversion phase, so it is possible to avoid building the extra spilling record if we know most variables can be accessed from some other closure variables.

Type information (more specifically, the object size information) can also be useful in supporting better closure representations. To allow more closure sharing while still satisfying the safe for space complexity rule (see Section 2.3.3), the compiler can use the type information as a guide to check whether it is safe to share two closures or not. As long as the extra objects being held are of constant size, sharing two closures would still not cause asymptotic increase of the space usage.

The unrolling lists technique described in Chapter 6 is only implemented in an experimental version of the SML/NJ compiler, and evaluated mainly upon several toy benchmarks. Since the unrolled list significantly shortens the length of control- and data-dependence chains in operations on lists, it will be interesting to implement and evaluate this technique on modern superscalar machines. On the other hand, more work has to be done to control the code explosion.

Another new direction in efficient compilation of higher-level languages is to see whether languages such as SML support better instruction level parallelism than conventional languages such as C and C++. There are many reasons to believe they will, or at least SML will:

- SML is a value-oriented functional language; SML programs contain significantly fewer side-effects than C and C++ programs. This means that less pointer aliasing analysis is necessary to calculate the control- and data-dependence at compile time.

- SML can be compiled efficiently using a heap-based environment allocation scheme (see Chapter 4). In our heap-based scheme, once a closure is created, no later writes are made to it. This further eliminates the number of side effects occurring at run time. The stack-based scheme, on the other hand, often needs to side-effect each stack frame many times.

- SML is a statically typed language. With type-directed compilation, the compiler knows more static information (e.g., object size) about each variable; this static information may be useful for better instruction scheduling.

- SML uses concrete datatype declaration and pattern matching to define and access most data structures. This higher-level abstraction gives the compiler the freedom to choose more efficient data representations that fit well on superscalar machines (e.g., unrolling lists).

In summary, higher-level languages such as SML hold great promise for efficient implementation on modern machines. After all, languages that are easier to use should be easier to analyze at compile time; programs that are more abstract should provide more opportunities for advanced compiler optimizations. In the near future, programs written in certain higher-level languages may be able to get better performance (on machines of the future) than programs written in languages such as C and C++. This is certainly the hope and dream of many software researchers.

Bibliography

[AAD*93] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[AB93] Michael S. Allen and Michael C. Becker. Multiprocessing Aspects of the PowerPC 601. In *IEEE COMPCON Spring '93*, pages 117–126. IEEE Computer Society Press, February 1993.

[AJ88] Andrew W. Appel and Trevor Jim. Optimizing Closure Environment Representations. Technical Report 168, Dept. of Computer Science, Princeton University, Princeton, NJ, 1988.

[AJ89] Andrew W. Appel and Trevor Jim. Continuation-Passing, Closure-Passing Style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[AL91] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (SIGPLAN Notices v. 26, no. 4)*, pages 96–107. ACM Press, April 1991.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[AMT89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, December 1989.

[App87] Andrew W. Appel. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letter*, 25(4):275–279, 1987.

[App89] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software—Practice and Experience*, 19(2):171–183, 1989.

[App90] Andrew W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, 1990.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App94a] Andrew W. Appel. Emulating Write-Allocate on a No-Write-Allocate Cache. Technical Report CS-TR-459-94, Princeton University, June 1994.

[App94b] Andrew W. Appel. Loop Headers in λ-calculus or CPS. *Lisp and Symbolic Computation*, page (to appear), 1994. Also available as Princeton University Tech Report CS-TR-460-94.

[AR92] William E. Aitken and John H. Reppy. Abstract Value Constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11, June 1992. Longer version available as Cornell Univ. Tech. Report.

[AS92] Andrew W. Appel and Zhong Shao. Callee-save Registers in Continuation-Passing Style. *Lisp and Symbolic Computation*, 5(3):191–221, 1992.

[AS94] Andrew W. Appel and Zhong Shao. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. Technical Report CS-TR-450-94, Princeton University, Department of Computer Science, Princeton, NJ, March 1994. To appear in *Journal of Functional Programming*.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Aug89] Lennart Augustsson. Garbage collection in the $< \nu, G >$-machine. Technical Report PMG memo 73, Dept. of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, December 1989.

[Bak76] Henry G. Baker. The Buried Binding and Stale Binding Problems of LISP 1.5. unpublished, undistributed paper, June 1976.

[Bak78] Henry G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bar84] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[BC79] Daniel G. Bobrow and Douglas W. Clark. Compact Encodings of List Structure. *ACM Transactions on Programming Languages and Systems*, 1(2):267–286, October 1979.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. In *Proc. ACM SIGPLAN '89 Conf. on Prog. Lang. Design and Implementation*, pages 275–284, New York, July 1989. ACM Press.

[BH86] J.E. Barnes and P. Hut. A Hierarchical $O(NlogN)$ Force Calculation Algorithm. *Nature*, 324(4):446–449, December 1986.

[Bjo94] Nikolaj S. Bjorner. Minimial Typing Derivations. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, June 1994.

[BM72] R. S. Boyer and J Moore. The Sharing of Structure in Theorem-Proving Programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7.* Edinburgh University Press, 1972.

[Bob75] Daniel G. Bobrow. A note on hash linking. *Communications of the ACM*, 18(7):413–415, July 1975.

[Car84a] Luca Cardelli. Compiling a functional language. In *Proc. of the 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217, August 1984.

[Car84b] Luca Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types, International Symposium*, pages 51–68, Berlin, June 1984. Springer-Verlag.

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. The Categorical Abstract Machine. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture, LNCS Vol 201*, pages 50–64, New York, 1985. Springer-Verlag.

[CG77] Douglas W. Clark and C. Cordell Green. An Empirical Study of List Structure in Lisp. *Communications of the ACM*, 20(2):78–87, February 1977.

[Cha82] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *Symposium on Compiler Construction*, pages 98–105, New York, June 1982. ACM Sigplan.

[Cha88] David R. Chase. Safety considerations for storage allocation optimizations. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pages 1–9, New York, June 1988. ACM Press.

[Cho88a] Fred C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pages 85–94, New York, June 1988. ACM Press.

[CHO88b] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation Strategies for Continuations. In *1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, New York, June 1988. ACM Press.

[CHR78] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE Code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, February 1978.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[Cla76] Douglas W. Clark. *List structure: measurements, algorithms, and encodings*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, August 1976.

[Dan87] Olivier Danvy. Memory Allocation and Higher-Order Functions. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 241–252. ACM Press, June 1987.

[Deu73] L. P. Deutsch. A Lisp machine with very compact programs. In *Proc. 3rd IJACI*, pages 697–703, 1973.

[DG94] Damien Doligez and Georges Gonthier. Re: stack scanning for generational g.c. E-mail message `<9403041606.AA07877@lix.polytechnique.fr>`, March 1994.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing First-Class Continuations in ML. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 163–173, New York, Jan 1991. ACM Press.

[Dig92] Digital Equipment Corp., Maynard, MA. *DECchip(tm) 21064-AA Microprocessor Hardware Reference Manual*, first edition, October 1992.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symp. on Principles of Prog. Languages*, pages 207–212, New York, Jan 1982. ACM Press.

[DTM94] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14. ACM Press, 1994.

[EA87] 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.

[FP91] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 268–277, New York, July 1991. ACM Press.

[Fre92] Tim Freeman. Carnegie Mellon University, personal communication, 1992.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, March 1994. CMU-CS-94-110.

[FTL94] M. Feeley, M. Turcotte, and G. LaPalme. Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D strucure determination. *Lisp and Symbolic Computation*, page (to appear), 1994.

[Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Boston, MA, 1985.

[GGR94] Lal George, Florent Guillaume, and John Reppy. A portable and optimizing backend for the SML/NJ compiler. In *Proceedings of the 1994 International Conference on Compiler Construction*, pages 83–97. Springer-Verlag, April 1994.

[Gre77] R. Greenblatt. LISP Machine Progress Report memo 444. Technical report, A.I. Lab., M.I.T., Cambridge, MA, August 1977.

[GS91] Carsten K. Gomard and Peter Sestoft. Globalization and Live Variables. In *Proceedings of the 1991 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 166–177. ACM Press, June 1991.

[Hal94] Cordelia V. Hall. Using Hindley-Milner Type Inference to Optimize List Representation. In *1994 ACM Conference on Lisp and Functional Programming*, pages 162–172, New York, June 1994. ACM Press.

[Han69] Wilfred J. Hansen. Compact List Representation: Definition, Garbage Collection, and System Implementation. *Communications of the ACM*, 12(9):499–507, Sep 1969.

[Han80] David R. Hanson. A Portable Storage Management System for the Icon Programming Language. *Software—Practice and Experience*, 10:489–500, 1980.

[Han90] Chris Hanson. Efficient Stack Allocation for Tail-Recursive Languages. In *1990 ACM Conference on Lisp and Functional Programming*, pages 106–118, New York, June 1990. ACM Press.

[Har86] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Univ. of Edinburgh, Dept. of Computer Science, Edinburgh, EH9 3JZ, August 1986.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proc. ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, pages 66–77, New York, 1990. ACM Press.

[HHH*90] William R. Hardell, Dwain A. Hicks, Lawrence C. Howell, Warren E. Maule, Robert Montoye, and David P. Tuttle. Data Cache and Storage Control Units. In *IBM RISC System/6000 Technology*, pages 44–50. IBM, 1990.

[Hil88] Mark D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[Hin69] Roger Hindley. The principle type scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.

[HJ94] Fritz Henglein and Jesper Jorgensen. Formally Optimal Boxing. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 213–226. ACM Press, 1994.

[HJ*et al*92] Paul Hudak, Simon Peyton Jones, and Philip Wadler *et al.* Report on the Programming Language Haskell, A Non-strict, Purely Functional Language Version 1.2. *SIGPLAN Notices*, 21(5), May 1992.

[HL94] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 123–137, New York, Jan 1994. ACM Press.

[HLPR94] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental Recompilation for Standard ML of New Jersey. In *ACM SIGPLAN Workshop on ML and its Applications*, June 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling Polymorphism Using Intensional Type Analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, page (to appear), New York, Jan 1995. ACM Press.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Joh85] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *The Second International Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, September 1985. Springer-Verlag.

[Jon92] Mark P. Jones. A theory of qualified types. In *The 4th European Symposium on Programming*, pages 287–306, Berlin, February 1992. Spinger-Verlag.

[Jou93] Norman P. Jouppi. Cache Write Policies and Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201. ACM Press, May 1993.

[KH89] Richard Kelsey and Paul Hudak. Realistic Compilation by Program Transformation. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 281–292, New York, 1989. ACM Press.

[KKR*86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, 21(7):219–233, July 1986.

[Kra87] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, New Haven, CT, 1987.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Lan65] P. J. Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Commnuications of the ACM*, 8(2):89–101, 1965.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[Ler94] Xavier Leroy. Manifest Types, Modules, and Separate Compilation. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 109–122, New York, Jan 1994. ACM Press.

[LH83] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–29, 1983.

[LH86] Kai Li and Paul Hudak. A New List Compaction Method. *Software – Practice and Experience*, 16(2):145–163, February 1986.

[Mac84] David B. MacQueen. Modules for Standard ML. In *1984 ACM Conference on Lisp and Functional Programming*, pages 198–207, New York, August 1984. ACM Press.

[Mac88] David B. MacQueen. Weak types. Distributed with Standard ML of New Jersey, 1988.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, Apirl 1960.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, March 1978.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[Pet89] John Peterson. Untagged data in tagged environments: choosing optimal representations at compile time. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 89–99, New York, September 1989. ACM Press.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, New York, 1987.

[Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, August 1991. ACM Press.

[Plo75] Gordon D. Plotkin. Call-by-Name, Call-by-Value, and the λ-calculus. *Theoretical Computer Science*, 1:125–59, 1975.

[Pou93] Eigil Poulsen. Representation Analysis for Efficient Implementation of Polymorphism. Master's thesis, DIKU, University of Copenhagen, 1993.

[RC86] J. Rees and W. Clinger. Revised Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[Rea89] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, MA, 1989.

[Rei94] Mark B. Reinhold. Cache Performance of Garbage-Collected Programs. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, pages 206–217. ACM Press, June 1994.

[Rep91] John H. Reppy. CML: A Higher-order Concurrent Language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[Rep93] John H. Reppy. A High-Performance Garbage Collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1993.

[Roz84] Guillermo Juan Rozas. Liar, an Algol-like Compiler for Scheme. S.B. thesis, MIT Dept. of Computer Science and Electrical Engineering, June 1984.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–246, April 1993.

[SA92] Zhong Shao and Andrew W. Appel. Smartest Recompilation. Technical Report CS-TR-395-92, Princeton Univ. Dept. of Computer Science, Princeton, NJ, October 1992.

[SA93] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proc. Twentieth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 439–450. ACM Press, 1993.

[SA94] Zhong Shao and Andrew W. Appel. Space-Efficient Closure Representations. In *1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, New York, June 1994. ACM Press.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, May 1991. CMU-CS-91-145.

[SM94] Darko Stefanovic and J. Eliot B. Moss. Characterization of Object Behaviour in Standard ML of New Jersey. In *1994 ACM Conference on Lisp and Functional Programming*, pages 43–54, New York, June 1994. ACM Press.

[SRA94] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling Lists. In *1994 ACM Conference on Lisp and Functional Programming*, pages 185–195, New York, June 1994. ACM Press.

[SS80] Guy L. Steele and Gerald Jay Sussman. The Dream of a Lifetime: A Lazy Variable Extent Mechanism. In *Proceedings of the 1980 LISP Conference*, pages 163–172, Stanford, 1980.

[Sta89] Standards Performance Evaluation Corp. *SPEC Benchmark Suite Release 1.0*, October 1989.

[Ste78] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[Ste84] Guy L. Steele. *The Common LISP: The Language.* Digital Press, Digital Equipment Corporation, 1984.

[TA90] David R. Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[Tar74] Robert E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Science*, 9(3):355–365, December 1974.

[Tof92] Mads Tofte. Principal Signatures for High-order ML Functors. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press.

[Tur37] Alan M. Turing. Computability and λ-definability. *J. Symbolic Logic*, 2:153–163, 1937.

[Ull93] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[Ung86] David M. Ungar. *The Design and Evaluation of A High Performance Smalltalk System.* MIT Press, Cambridge,MA, 1986.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Fourteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 307–313, New York, Jan 1987. ACM Press.

[Wan80] Mitchell Wand. Continuation-Based Multiprocessing. In *Conf. Record of the 1980 Lisp Conf.*, pages 19–28, New York, August 1980. ACM Press.

[WB89] P. Wadler and S. Blott. How to make ad hoc polymorphism less ad hoc. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 60–76, New York, Jan 1989. ACM Press.

[Wil91] Paul R. Wilson. Some Issues and Strategies in Heap Management and Memory Hierarchies. *SIGPLAN Notices*, 26(3):45–52, March 1991.

[WLM92] 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, New York, June 1992. ACM Press.

[WM81] D. Weinreb and D. Moon. Lisp Machine Manual. Technical report, Symbolics Corp., Cambridge, Mass., 1981.

[Zor91] Benjamin Zorn. The Effect of Garbage Collection on Cache Performance. Technical Report CU-CS-528-91, University of Colorado, Boulder, CO, May 1991.