TIL: A Type-Directed Optimizing Compiler for ML

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

1 Introduction

We are investigating a new approach to compiling Standard ML (SML) based on four key technologies: *intensional polymorphism* [23], *nearly tag-free garbage collection* [12, 46, 34], *conventional functional language optimization*, and *loop optimization*. To explore the practicality of our approach, we have constructed a compiler for SML called TIL, and are thus far encouraged by the results: On DEC ALPHA workstations, programs compiled by TIL are roughly three times faster, do one-fifth the total heap allocation, and use one-half the physical memory of programs compiled by SML of New Jersey (SML/NJ). However, our results are still preliminary — we have not yet investigated how to improve compile time; TIL takes about eight times longer to compile programs than SML/NJ. Also, we have not yet implemented the full module system of SML, although we do provide support for structures and separate compilation. Finally, we expect the performance of programs compiled by TIL to improve significantly as we tune the compiler and implement more optimizations.

Two key issues in the compilation of advanced languages such as SML are the presence of *garbage collection* and *type variables*. Most compilers use a universal representation for values of unknown or variable type. In particular, values are forced to fit into a tagged machine word; values larger than a machine word are represented as pointers to tagged, heap-allocated objects. This approach supports fast garbage collection and efficient polymorphic functions, but can result in inefficient code when types are known at compile time. Even with recent advances in SML compilation, such as Leroy's representation analysis [28], values must be placed in a universal representation before being stored in updateable data structures (e.g., arrays) or recursive data structures (e.g., lists).

Intensional polymorphism and *tag-free garbage collection* eliminate the need to use a universal representation when compiling polymorphic languages. TIL uses these technologies to represent many data values "naturally". For example, TIL provides tag-free, unallocated, word-sized integers; aligned, unboxed floating-point arrays; and unallocated multi-argument functions. These natural representations and calling conventions not only improve the performance of SML programs, but also allow them to interoperate with legacy code written in languages such as C and Fortran. When types are unknown at compile time, TIL may produce machine code which is slower and bigger than conventional approaches. This is because types must be constructed and passed to polymorphic functions, and polymorphic functions must examine the types at run-time to determine appropriate execution paths. However, when types are known at compile time, no overhead is incurred to support polymorphism or garbage collection.

Because these technologies make polymorphic functions slower, it becomes important to eliminate as many polymorphic functions at compile time as is possible. Inlining and uncurrying are well-known techniques for eliminating polymorphic and higher-order functions. We have found that for the benchmarks used here, these techniques eliminate all polymorphic functions and all but a few higher-order functions when programs are compiled as a whole.

We have also found that applying traditional loop optimizations to recursive functions, such as common subexpression elimination and invariant removal, is important. In fact, these optimization reduce execution time by a median of 39%.

An important property of TIL is that all optimizations and the key transformations are performed on *typed intermediate languages* (hence the name TIL). Maintaining correct type information throughout optimization is necessary to support both intensional polymorphism and garbage collection, both of which require type information at run time. By using strongly-typed intermediate languages, we ensure that type information is maintained in a principled fashion, instead of relying upon *ad hoc* invariants. In fact, using the intermediate forms of TIL, an "untrusted" compiler can produce fully optimized intermediate code, and a client can automatically verify the type integrity of the code. We have found that this ability has a strong engineering benefit: type-checking the output of each optimization or transformation helps us identify and eliminate bugs in the compiler.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under Grant No. CCR-9502674, and in part by the Isaac Newton Institute for Mathematical Sciences, Cambridge, England. David Tarditi was also partly supported by an AT&T Bell Labs PhD Scholarship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency, the U.S. Government, the National Science Foundation or AT&T.

In the remainder of this paper, we describe the technologies used by TIL in detail, give an overview of the structure of TIL, present a detailed example showing how TIL compiles ML code, and give performance results of code produced by TIL.

2 Overview of the Technologies

This section contains a high-level overview of the technologies we use in TIL.

2.1 Intensional Polymorphism

Intensional polymorphism [23] eliminates restrictions on data representations due to polymorphism, separate compilation, abstract datatypes, and garbage collection. It also supports efficient calling conventions (multiple arguments passed in registers) and tag-free polymorphic, structural equality.

With intensional polymorphism, types are constructed and passed as values at run time to polymorphic functions, and these functions can branch based on the types. For example, when extracting a value from an array, TIL uses a `typecase` expression to determine the type of the array and to select the appropriate specialized subscript operation:

```
fun sub[α](x:α array, i: int) =
          typecase α of
            int => intsub(x, i)
          | float => floatsub(x, i)
          | ptr(τ) => ptrsub(x, i)
```

If the type of the array can be determined at compile-time, then an optimizer can eliminate the `typecase`:

```
sub[float](a, 5) ↪ floatsub(a, 5)
```

However, intensional polymorphism comes with two costs. First, we must construct and pass representations of types to polymorphic functions at run time. Furthermore, we must compile polymorphic functions to support any possible representation and insert `typecase` constructs to select the appropriate code paths. Hence, the code we generate for polymorphic functions is both bigger and slower, and minimizing polymorphism becomes quite important.

Second, in order to use type information at run time, for both intensional polymorphism and tag-free garbage collection, we must propagate types through each stage of compilation. To address this second problem, almost all compilation stages, including optimization and closure conversion, are expressed as type-directed, type-preserving translations to strongly-typed intermediate languages.

The key difficulty with using typed intermediate languages is formulating a type system that is expressive enough to statically type check terms that branch on types at run time, such as `sub`. The type system used in TIL is based on the approach suggested by Harper and Morrisett [23, 33]. Types themselves are represented as expressions in a simply-typed λ-calculus extended with an inductively generated base kind (the monotypes), and a corresponding induction elimination form. The induction elimination form is essentially a "Typecase" at the type level; this allows us to write type expressions that track the run-time control flow of term-level `typecase` expressions. Nevertheless, the type system used by TIL remains both *sound* and *decidable*. This implies that at any stage during optimization, we can automatically verify the type integrity of the code.

2.2 Conventional and Loop-Oriented Optimizations

Program optimization is crucial to reducing the cost of intensional polymorphism,improving loops and recursive functions, and eliminating higher-order and polymorphic functions. TIL employs optimizations found in conventional functional language compilers, including inlining, uncurrying, dead-code elimination, and constant-folding. In addition, TIL does a set of generalized "loop-oriented" optimizations to improve recursive functions. These optimizations include common-subexpression elimination, invariant removal, and array-bound check removal. In spite of the large number of different optimizations, each optimization produces type-correct code.

TIL applies optimizations across entire compilation units. This makes it more likely that inlining and uncurrying will eliminate higher-order functions, which are likely to interfere with the loop-oriented optimizations. Since the optimizations are applied to entire compilation units (which may be whole programs), we paid close attention to algorithmic efficiency of individual optimization passes. Most of the passes have an $O(N \log N)$ worst-case asymptotic complexity (excluding checking types for equality), where N is program size.

2.3 Nearly Tag-Free Garbage Collection

Nearly tag-free garbage collection uses type information to eliminate data representation restrictions due to garbage collection. The basic idea is to record enough representation information at compile time so that, at any point where a garbage collection can occur, it is possible to determine whether or not values are pointers and hence must be traced by the garbage collector. Recording the information at compile time makes it possible for code to use untagged representations. Unlike so-called conservative collectors (see for example [10, 14]), the information recorded by TIL is sufficient to collect all unreachable objects.

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

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

This approach is well-understood for monomorphic languages requiring garbage collection [12]. Following Tolmach [46], we extended it to a polymorphic language as follows: when a variable whose type is unknown is saved in a stack frame, the type of the variable is also saved in the stack frame. However, unlike Tolmach, we evaluate substitutions of ground types for type variables eagerly instead of lazily. This is due in part for technical reasons (see [33, Chapter 7]), and in part to avoid a class of space leaks that might result with lazy substitution.

3 Compilation Phases of TIL

Figure 1 shows the various compilation phases of TIL. The phases through and including closure conversion use a typed intermediate language. The phase after closure conversion use an untyped language where variables are annotated with garbage collection information. The low-level phases of the compiler use languages where registers are annotated with garbage collection information.

The following sections describe the phases of TIL and the intermediate languages they use in more detail.

3.1 Front-end

The first phase of TIL uses the front-end of the ML Kit compiler [8] to parse and elaborate (type check) SML source code. The Kit produces annotated abstract syntax for all of SML and then compiles a subset of this abstract syntax to an explicitly-typed core language called Lambda. The compilation to Lambda eliminates pattern matching and various derived forms.

We extended Lambda to support signatures, structures (modules), and separate compilation. Each source module is compiled to a Lambda module with an explicit list of imported modules and their signatures. Imported signatures may include transparent definitions of types defined in other modules; hence TIL supports a limited form of *translucent* [22] or *manifest* types [29]. Currently, the mapping to Lambda does not handle signatures, nested structures, or functors. In principle, however, all of these constructs are supported by TIL's intermediate languages.

3.2 Lmli and Type-Directed Optimizations

Lmli, which stands for λ_i^{ML} [23], is an intensionally polymorphic language that provides explicit support for constructing, passing, and analyzing types at run-time. We use these constructs in the translation of Lambda to Lmli to provide efficient data representations for user-defined datatypes, multi-argument functions, tag-free polymorphic equality, and specialized arrays.

After the conversion from Lambda to Lmli, TIL performs a series of type-directed optimizations. SML provides only single-argument functions; multiple arguments are passed in a record. The first optimization, argument flattening, translates each function which takes a record as an argument to a function which takes the components of the record as multiple arguments. These arguments are passed in registers, avoiding allocation to create the record and memory operations to access record components. If a function takes an argument of variable type α, then we use `typecase` to determine the proper calling convention, according to the instantiation of α at run time.

As with functions, datatype constructors in SML take a single argument. For example, the cons data constructor (`::`) for an α list takes a single record, consisting of an α value and an α list value. Naively, such a constructor is represented as a pair consisting of a tag (e.g., `cons`), and a pointer to the record containing the α value and the α list value. The tag is a small integer value used to distinguish among the constructors of a datatype (e.g., `nil` vs. `::`). Constructor flattening rewrites all constructors that take records as arguments so that the components of the records are flattened. In addition, constructor flattening eliminates tag components when they are unneeded. For example, cons applied to `(hd,tl)` is simply represented as a pointer to the pair `(hd,tl)`, since such a pointer can always be distinguished from `nil`. If the constructor takes an argument of unknown type, then we use `typecase` to determine the proper representation, according to the instantiation of α at run time.

Because lists are used often in SML, the SML/NJ compiler also flattens cons cells (and other constructors). However, in violation of the SML Definition [31], SML/NJ prevents programmers from abstracting the type of these constructors, in order to prevent representation mismatches between definitions of abstract datatypes and their uses [3]. In contrast, TIL supports fully abstract datatype components, but uses intensional polymorphism to determine representations of abstract datatypes, potentially at run time.

In addition to specializing calling conventions and datatypes, the conversion from Lambda to Lmli makes polymorphic equality explicit as a term in the language. Also, arrays are specialized into one of three cases: int arrays, float arrays, and pointer arrays. Intensional polymorphism is used to select the appropriate creation, subscript, and update operations for polymorphic arrays.

Finally, TIL boxes all floating point values, except for values stored in floating-point arrays. We chose to box floats to make record operations faster, since typical SML code manipulates many records but few floats. The issue is that floating-point values are 64 bits, while other scalars and pointers are 32 bits. If floats were unboxed, then record offset calculations could not always be done at compile time. Fortunately, the optimizer later eliminates unecessary box/unbox operations during the constant-folding phase, so straight-line floating point code still runs fast.

In all, the combination of type-directed optimizations reduce running times by roughly 40% and allocation by 50% [33, Chapter 8]. However, much of this improvement can be realized by other techniques; For example, SML/NJ uses Leroy's unboxing technique to achieve comparable improvements for calling conventions [42]. The advantage of our approach is that we use a single mechanism (intensional polymorphism) to specialize calling conventions, flatten constructors, unbox floating-point arrays, and eliminating tags for both polymorphic equality and garbage collection.

3.3 Optimizations

TIL employs an extensive set of optimizations. The optimizations include most of those typically done by compilers for functional languages. They also include loop-oriented optimizations, such as invariant removal, applied to recursive functions.

TIL first translates Lmli to a subset of Lmli called Bform. Bform, based on A-Normal-Form [18], is a more regular intermediate language than Lmli that facilitates optimization. The translation from Lmli names all intermediate computations and binds them to variables by a `let`-construct. It also names all potentially heap-allocated values, including strings, records and functions. Finally, it allows nested

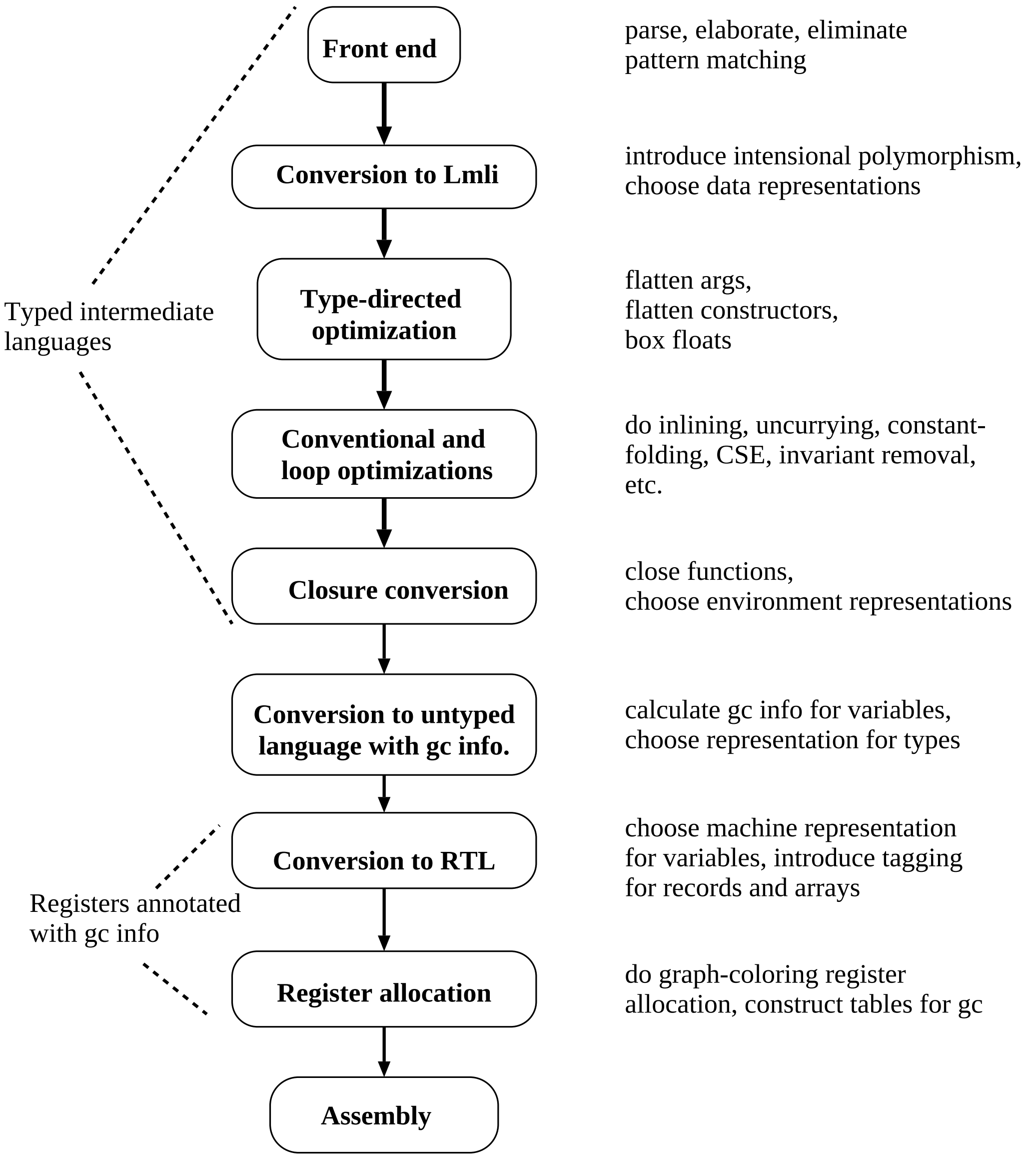

Figure 1: Phases of the TIL compiler

`let` expressions only within switches (branch expressions). Hence, the translation from Lmli to Bform linearizes and names nested computations and values.

After translation to Bform, TIL performs the following conventional transformations:

- **alpha-conversion:** Bound variables are uniquely renamed.
- **dead-code elimination:** unreferenced, pure expressions and functions are eliminated.
- **uncurrying:** Curried functions are transformed to multi-argument functions, when possible.
- **constant folding:** Arithmetic operations, switches, and typecases on constant values are reduced, as well as projections from known records.
- **sinking:** Pure expressions used in only one branch of a switch are pushed into that branch. However, such expressions are not pushed into function definitions.
- **inlining:** Non-escaping functions that are called only once are always inlined. Small, non-recursive functions are inlined in a bottom-up pass. Recursive functions are never (directly) inlined.
- **inlining switch continuations:** The continuation of a switch is inlined when all but one branch raises an exception. For example, the expression

  ```
  let x = if y then e2 else raise e3
  in e4
  end
  ```

  is transformed to

  ```
  if y then let x = e2 in e4 end else raise e3.
  ```

  This makes expressions in e_2 available within e_4 for optimizations like common sub-expression elimination.
- **minimizing fix:** Mutually-recursive functions are broken into sets of strongly connected components. This improves inlining and dead code elimination, by separating non-recursive and recursive functions.

In addition to these standard functional language transformations, TIL also applies loop-oriented optimizations to recursive functions:

- **common subexpression elimination (CSE):** Given an expression

  ```
  let x = e1
  in e2
  end
  ```

  if e_1 is pure or the only effect it may have is to raise an exception, then all occurrences of e_1 in e_2 are replaced with `x`. The only expressions that are excluded from CSE are side-effecting expressions and function calls.
- **eliminating redundant switches:** Given an expression

  ```
  let x = if z then
              let y = if z then e1 else e2
              in ...
  ```

  the nested `if` statement is replaced by e_1, since `z` is always `true` at that point.
- **invariant removal:** Using the call graph, we calculate the nesting depth of each function. (Nesting-depth is analogous to loop-nesting depth in languages like C.) TIL assigns a `let`-bound variable and the expression it binds a nesting depth equal to that of the nearest enclosing function. For every pure expression e, if all free variables of e have a nesting depth less than e, TIL moves the definition of e right after the definition of the free variable with the highest lexical nesting depth.
- **hoisting:** All constant expressions are hoisted to the top of the program. An expression is a constant expression if it uses only constants or variables bound to constant expressions.
- **eliminating redundant comparisons:** A set of simple arithmetic relations of the form $x < y$ is propagated top-down through the program. A "rule-of-signs" abstract interpretation is used to determine signs of variables. This information is used to eliminate array-bounds checks and other tests.

TIL applies the optimizations as follows: first, it performs a round of reduction optimizations, including dead-code elimination, constant folding, inlining functions called once, CSE, eliminating redundant switches, and invariant removal. These optimizations do not increase program size and should result in faster code. It iterates these optimizations until no further reductions occur. Then it performs switch-continuation inlining, sinking, uncurrying, comparison elimination, fix minimizing, and inlining. The entire process, starting with the reduction optimizations, is iterated two or more times.

3.4 Closure conversion

TIL uses a type-directed, abstract closure conversion in the style suggested by Minamide, Morrisett, and Harper [32] to convert Lmli-Bform programs to to Lmli-Closure programs. Lmli-Closure is an extension of Lmli-Bform that provides constructs for explicitly constructing closures and their environments.

For each escaping Bform function, TIL generates a closed piece of code, a type environment, and a value environment. The code takes the free type variables and free value variables of the original function as extra arguments. The types and values corresponding to these free variables are placed in records. These records are paired with the code to form an abstract closure. TIL uses a flat environment representation for type and value environments [5].

For known functions, TIL generates closed code but avoids creating environments or a closure. Following Kranz [27], we modify the call sites of known functions to pass free variables as additional arguments.

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

3.5 Conversion to an untyped language

To simplify the conversion to low-level assembly code, TIL translates Lmli-Closure programs to an untyped language called Ubform. Ubform is a much simpler language than Lmli, since similar type-level and term-level constructs are collapsed to the same term-level constructor. For example, in the translation from Lmli-Closure to Ubform, TIL replaces `typecase` with a conventional `switch` expression. This simplifies generation of low-level code, since there are many fewer cases.

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

3.6 Conversion to RTL

Next TIL converts Ubform programs to RTL,a register-transfer language similar to ALPHA or other RISC-style assembly language. RTL provides an infinite number of pseudo-registers each of which is annotated with representation information. Representation information is extended to include locatives, which are pointers into the middle of objects. Pseudo-registers containing locatives are never live across a point where garbage collection can occur. RTL also provides heavy-weight function call and return mechanisms, and a form of interprocedural goto for implementing exceptions.

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

3.7 Register allocation and assembly

Before doing register allocation, TIL converts RTL programs to ALPHA assembly language with extensions similar to those for RTL. Then TIL uses conventional graph-coloring register allocation to allocate physical registers for the pseudo-registers. It also generates tables describing layout and garbage collection information for each stack frame, as described in Section 2.3. Finally, TIL generates actual ALPHA assembly language and invokes the system assembler, which does instruction scheduling and creates a standard object file.

4 An example

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

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

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

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

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

Figures 2 through 7 show the actual intermediate code created as `dot` and `sub2` pass through the various stages of TIL. For readability, we have renamed variables, erased type information, and performed some minor optimizations, such as eliminating selections of fields from known records.

Figure 2 shows the functions after they have been converted to Lmli. The `sub2` function takes a type as an argument. A function parameterized by a type is written as Λ`t.`, while a function parameterized by a value is written as λ`i.` In the `dot` function, the `sub2` function is first applied to a *type* and then applied to its actual values. Each function takes only one argument, often a record, from which fields are selected. The quality of code at this level is quite poor: there are eight function applications, four record constructions, and numerous checks for array bounds.

Figure 3 shows the Lmli fragment after it has been converted to Lmli-Bform. Functions have been transformed to take multiple arguments instead of records and every intermediate compuation is named.

Figure 4 shows the Lmli-Bform fragment after it has been optimized. All the function applications in the body of the loop have been eliminated. `psub_ai(av,a)` is an application of the (unsafe) integer array subscript primitive. All of the comparisons for array bounds checking have been safely eliminated, and the body of the loop consists of 9 expressions. This loop could be improved even further; we have yet to implement any form of strength reduction and induction variable elimination.

Figure 5 shows the Lmli-Bform fragment after it has been converted to Ubform. Each variable is now annotated with *representation information*, to be used by the garbage collector. `INT` denotes integers and `TRACE` denotes pointers to tagged objects. The function is now *closed*, since it was closure converted before converting to Ubform.

Figure 6 shows the Ubform fragment after it has been converted to RTL. Every pseudo-register is now annotated with precise representation information for the collector. The representation information has been extended to include `LOCATIVE`, which denotes pointers into the middle of tagged objects. Locatives cannot be live across garbage-collection points. The `(*)` indicates points where the `psub_ai` primitive has been expanded to two RTL instructions. This indicates that induction-variable elimination would also be profitable at the RTL level. The `return` instruction's operand is a pseudo-register containing the return address.

Figure 7 shows the actual DEC ALPHA assembly language generated for the `dot` function. The code between `L1` and `L3` corresponds to the RTL code. The other code is epilogue and prologue code for entering and exiting the function. Note that no tagging operations occur anywhere in this function.

5 Performance

In this section, we compare the performance of programs compiled by TIL against programs compiled by the SML/NJ

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

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

Figure 2: After conversion to Lmli

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

Figure 3: Lmli-Bform before optimization

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

Figure 4: Lmli-Bform after optimization

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

Figure 5: After conversion to Ubform

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

Figure 6: After conversion to RTL

compiler. We measure execution time, heap allocation, physical memory requirements, executable size, and compile time. We also measure the effect of loop optimizations. Further performance analysis of TIL appears in Morrisett's [33] and Tarditi's theses [45].

5.1 Benchmarks

Table 1 describes the benchmark programs, which range in size from 62 lines to about 2000 lines of code. Some of these programs have been used previously for measuring ML performance [5, 16]. The benchmarks cover a range of application areas including scientific computing, list-processing, systems programming, and compilers.

We compiled the programs as single closed modules. For Lexgen and Simple, which are standard benchmarks [5], we eliminated functors by hand because TIL does not yet support the full SML module language. Because whole programs were given to the compiler, we found that the optimizer naturally eliminated all polymorphic functions. Consequently, for this benchmark suite, there was no run-time cost to support intensional polymorphism.

We extended the built-in ML types with safe 2-dimensional arrays. The 2-d array operations do bounds checking on each dimension and then use unsafe 1-d array operations. Arrays are stored in column-major order.

5.2 Comparison against SML/NJ

We compared the performance of TIL against SML/NJ in several dimensions: execution time, total heap allocation, physical memory footprint, the size of the executable, and compilation time.

For TIL, we compiled programs with all optimizations enabled. For SML/NJ, we compiled programs using the default optimization settings. We used a recent internal release of SML/NJ (a variant of version 1.08), since it produces code that is about 35% faster than the current standard release (0.93) of SML/NJ [41].

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

Figure 7: Actual DEC ALPHA assembly language

Program	lines	Description
Checksum	241	Checksum fragment from the Foxnet [7], doing 5000 checksums on a 4096-byte array.
FFT	246	Fast fourier transform, multiplying polynomials up to degree 65,536
Knuth-Bendix	618	An implementation of the Knuth-Bendix completion algorithm.
Lexgen	1123	A lexical-analyzer generator [6], processing the lexical description of Standard ML.
Life	146	The game of Life implemented using lists [39].
Matmult	62	Integer matrix multiply, on 200x200 integer arrays.
PIA	2065	The Perspective Inversion Algorithm [47] deciding the location of an object in a perspective video image.
Simple	870	A spherical fluid-dynamics program [17], run for 4 iterations with grid size of 100.

Table 1: Benchmark Programs

TIL always prefixes a set of operations on to each module that it compiles, in order to facilitate optimization. This "inline" prelude contains 2-d array operations, commonly-used list functions, and so forth. To avoid handicapping SML/NJ, we created separate copies of the benchmark programs for SML/NJ, and placed equivalent "prelude" code at the beginning of each program by hand.

Since TIL creates stand-alone executables, we used the `exportFn` facility of SML/NJ to create stand-alone programs. The `exportFn` function of SML/NJ dumps part of the heap to disk and throws away the interactive system.

We measured execution time on DEC ALPHA AXP 3000/-300LX workstations, running OSF/1, version 2.0, using the UNIX `getrusage` function. For SML/NJ, we started timing after the heap had been reloaded. For TIL, we measured the entire execution time of the process, including load time. We made 5 runs of each program on an unloaded workstation and chose the lowest execution time. Each workstation had 96MBytes of physical memory, so paging was not a factor in the measurements.

We measured total heap allocation by instrumenting the TIL run-time system to count the bytes allocated. We used existing instrumentation in the SML/NJ run-time system. We measured the maximum amount of physical memory during execution using `getrusage`. We used the `size` program to measure the size of executables for TIL. For SML/NJ, we used the `size` program to measure the size of the run-time system and then added the size of the heap created by `exportFn`. Finally, we measured end-to-end compilation time, including time to assemble files produced by TIL.

Figures 8 through 11 present the measurements. For each benchmark, measurements for TIL were normalized to those for SML/NJ and then graphed. SML/NJ performance is the 100% mark on all the graphs.

Figure 8 presents relative running times. On average, programs compiled by TIL run 3.3 times faster than programs compiled by SML/NJ. In fact, all programs except `Knuth-Bendix` and `Life` are substantially faster when compiled by TIL. We speculate that less of a speed-up is seen for `Knuth-Bendix` and `Life` because they make heavy use of list-processing, which SML/NJ does a good job of compiling.

Figure 9 compares the relative amounts of heap allocation. On average, the amount of data heap-allocated by the TIL program is about 17% of the amount allocated by the SML/NJ program. This is not surprising, because TIL uses a stack while SML/NJ allocates frames on the heap.

Figure 10 presents the relative maximum amounts of physical memory used. On average, TIL programs use half the memory used by SML/NJ programs. We see that floating-point programs use the least amount of memory relative to comparable SML/NJ programs. We speculate that this is due to TIL's ability to keep floating values unboxed when stored in arrays.

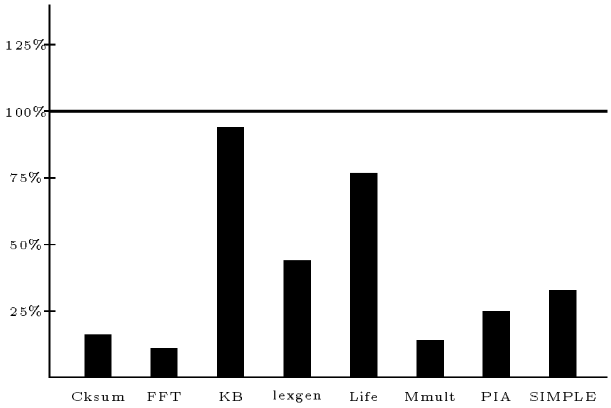

Figure 8: TIL Execution Time Relative to SML/NJ

TIL stand-alone programs are about half the size of stand-alone heaps and the runtime system of SML/NJ. The difference in size is mostly due to the different sizes of the runtime systems and standard libraries for the two compilers. (TIL's runtime system is about 100K, while SML/NJ's runtime is about 425K.) The program sizes for TIL confirm that generating tables for nearly tag-free garbage collection consumes a modest amount of space, and that the inlining strategy used by TIL produces code of reasonable size.

Figure 11 compares compilation times for TIL and SML/NJ. SML/NJ does much better than TIL when it comes to compilation time, compiling about eight times faster. However, we have yet to tune TIL for compilation speed.

5.3 Loop-Oriented Optimizations

We also investigated the effect of the loop-oriented optimizations (CSE, invariant removal, hoisting, comparison elimination, and redundant switch elimination). For each benchmark, we compared performance with the loop optimizations against performance without the loop optimizations.

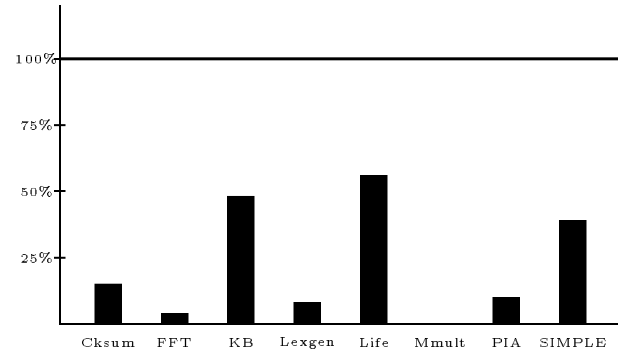

Figure 9: TIL Heap Allocation Relative to SML/NJ

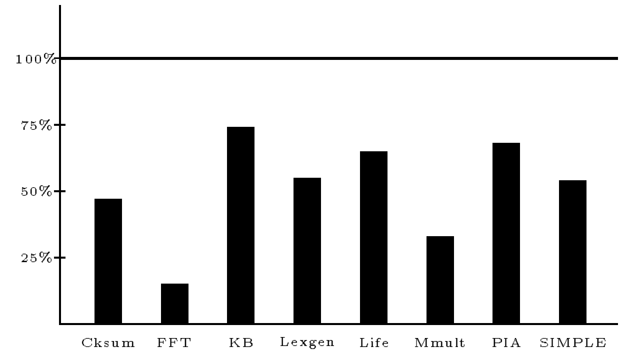

Figure 10: TIL Physical Memory Used Relative to SML/NJ

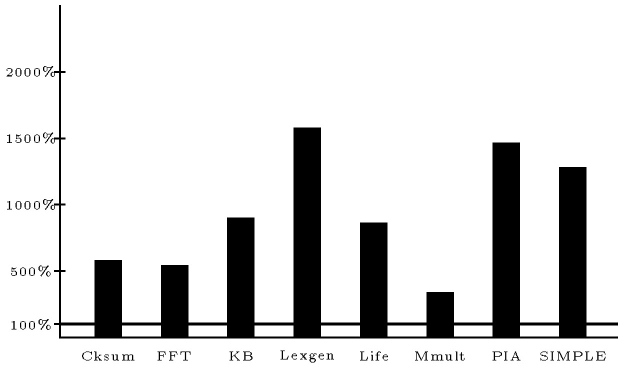

Figure 11: Til Compilation Time Relative to SML/NJ

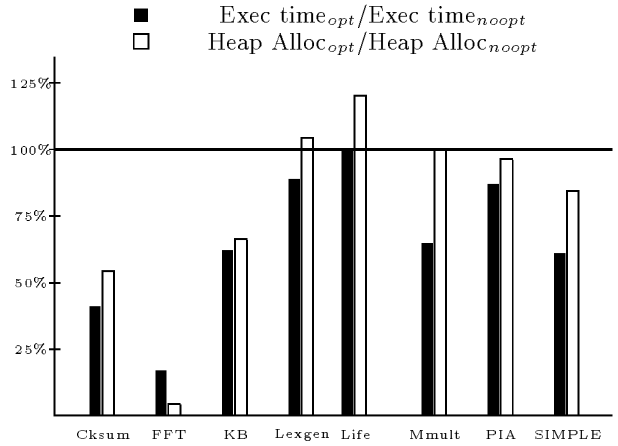

Figure 12: Effects of Loop Optimizations

Figure 12 presents the ratios of execution time with the loop optimizations to execution time without the loop optimizations, and similar ratios for total heap allocation. The loop optimizations reduce execution time by 0 to 83%, with a median reduction of 39%. The effect on heap allocation ranges from an increase of 20% to a decrease of 96.5%, with a median decrease of 10%.

For `matmult`, the matrix multiplication function is small enough that the optimizer inlines it, making the array dimensions known. If the array dimensions are held unknown, then the loop optimizations speed up `matmult` by a factor of 2.5.

6 Related Work

Morrison *et al.* used an "ad-hoc" approach to implement polymorphism in their implementation of Napier '88 [35]. In particular, they passed representations of types to polymorphic routines at run-time to determine behavior. However, to our knowledge, Napier '88 did not use types to implement tag-free garbage collection. Also, there is no description of the internals of the Napier '88 compiler, nor is there an account of the performance of code generated by the compiler.

Peyton Jones and Launchbury suggested that types could be used to unbox values in a polymorphic language [26]. However, they only supported a limited set of "unboxed types" (ints and floats) and restricted these types from instantiating type variables. Later, Leroy suggested a general approach for unboxing values based on the ML type system [28]. Leroy's approach has been extended and implemented elsewhere [38, 24, 42], including the SML/NJ compiler. It does not support unboxed array components nor flattened, recursive datatypes. Tolmach [46] combined Leroy's approach with tag-free garbage collection. However, he used an *ad hoc* approach to propagate type information to the collector.

Other researchers have suggested that polymorphism should be eliminated entirely at compile time [9, 25, 21], in the style of C++ templates [44]. This prevents separate compilation of a polymorphic definition from its uses. In contrast, intensional polymorphism, and in particular the intermediate forms of TIL, support separate compilation of polymorphic definitions, though we have yet to take advantage of this.

Tag-free garbage collection was originally proposed for monomorphic languages like Pascal, but has been used elsewhere [12, 11, 48, 15]. Britton suggested associating type information with return addresses on the stack [12]. Appel suggested extending this technique to ML by using unification [4]. Goldberg and Gloger improved Appel's algorithm [20, 19]. None of the unification-based algorithms were implemented due to the complexity of the algorithms and the overhead of performing unification during garbage collection.

Aditya, Flood, and Hicks used type-passing to support fully tag-free garbage collection for Id [1]. Independently, Tolmach [46] implemented a type-passing garbage collection algorithm for ML. Our approach differs from others by using "nearly" tag-free collection. In particular, records and arrays on the heap are tagged. Another difference is that we calculate type environments *eagerly*, while the other implementations construct type environments *lazily* during garbage collection.

Loop-oriented optimizations are well-known for imperative languages [2]. However, few results are reported for Lisp, Scheme, and ML. Appel [5] and Serrano [40] report common-subexpression elimination optimizations similar to ours. Appel found that CSE was not useful in the SML/NJ compiler. Serrano restricted CSE to pure expressions, while our CSE handles expressions which may raise exceptions.

7 Conclusions and future work

Our results show that for core-SML programs compiled as a whole, intensional polymorphism can remove restrictions on data representation, yet cost literally nothing due to the effectiveness of optimization. They also show that loop optimizations can improve program performance significantly.

These results suggest that ML can be compiled as well as conventional languages such as Pascal. TIL produces code that is similar in many important respects to code produced by Pascal and C compilers. For example, most function calls are known, since few higher-order functions are left, integers are untagged, and most code is monomorphic.

There are numerous areas that we would like to investigate further. We would like to explore the effect of separate compilation. With separate compilation, polymorphic functions may be compiled separately from their uses, leading to some cost for intensional polymorphism. We would like to measure this cost and explore what kinds of optimizations can reduce it.

Another direction we would like to investigate is how this approach performs for larger programs. We would like to add support for more of the ML module system, since large ML programs make extensive use of the module system. We would also like to improve TIL's compile times, so that large programs can also be compiled as a whole.

Finally, we would like to continue improving the performance of ML programs. We would like to extend our register allocation strategy along the lines of Chow [13] or Steenkiste [43]. We would also like to investigate more loop optimizations, such as strength-reduction, induction-variable elimination, and loop unrolling. On a more speculative note, we would like to explore stack allocation of data structures.

References

[1] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In LFP '94 [30], pages 12–23.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley Publishing Company, 1986.

[3] Andrew Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.

[4] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, (2):153–162, 1989.

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Andrew W. Appel, James S. Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[7] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In LFP '94 [30], pages 55–64.

[8] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit, Version 1. Technical Report 93/14, DIKU, 1993.

[9] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[10] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In PLDI '93 [36], pages 197–206.

[11] P. Branquart and J. Lewi. A scheme for storage allocation and garbage collection for Algol-68. In *Algol-68 Implementation*. North-Holland Publishing Company, Amsterdam, 1970.

[12] Dianne Ellen Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[13] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, Atlanta, Georgia, June 1988. ACM.

[14] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the 17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 1990. ACM.

[15] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, CA, June 1992. ACM.

[16] Amer Diwan, David Tarditi, and Eliot Moss. Memory-System Performance of Programs with Intensive Heap Allocation. *Transactions on Computer Systems*, August 1995.

[17] 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.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In PLDI '93 [36], pages 237–247.

[19] Benjamin Goldberg. Tag-free garbage collection in strongly typed programming languages. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, Toronto, Canada, June 1991. ACM.

[20] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 53–65, San Francisco, California, June 1992. ACM.

[21] Cordelia Hall, Simon L. Peyton Jones, and Patrick M. Sansom. Unboxing using specialisation. In D. Turner K. Hammond, P.M. Sandom, editor, *Functional Programming, 1994*. Springer-Verlag, 1995.

[22] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL '94 [37], pages 123–137.

[23] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995. ACM.

[24] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In POPL '94 [37], pages 213–226.

[25] M.P. Jones. Partial evaluation for dictionary-free overloading. Research Report YALEU/DCS/RR-959, Yale University, New Haven, Connecticut, USA, April 1993.

[26] Simon Peyton Jones and John Launchbury. Unboxed values as first-class citizens. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes on Computer Science*, pages 636–666. ACM, Springer-Verlag, 1991.

[27] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

[28] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, NM, January 1992. ACM.

[29] Xavier Leroy. Manifest types, modules, and separate compilation. In POPL '94 [37], pages 109–122.

[30] *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994. ACM.

[31] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[32] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996. ACM.

[33] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Published as Technical Report CMU-CS-95-226.

[34] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[35] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, July 1991.

[36] *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993. ACM.

[37] *Conference Record of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM.

[38] Eigil Rosager Poulsen. Representation analysis for efficient implementation of polymorphism. Technical report, Department of Computer Science (DIKU), University of Copenhagen, April 1993. Master Dissertation.

[39] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, Massachusetts, 1989.

[40] Manual Serrano and Pierre Weis. 1+1 = 1: an optimizing CAML compiler. Technical Report 2264, INRIA, June 1994.

[41] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton University, Princeton, New Jersey, November 1994.

[42] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, California, June 1994. ACM.

[43] Peter Steenkiste. Advanced register allocation. In Peter Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1990.

[44] Bjarne Stroustrup. *The C++ Programming Language, 2nd Edition*. Addison-Wesley, 1991.

[45] David R. Tarditi. *Optimizing ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[46] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In LFP '94 [30], pages 1–11.

[47] Kevin G. Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, August 1990.

[48] P.L. Wodon. Methods of garbage collection for Algol-68. In *Algol-68 Implementation*. North-Holland Publishing Company, Amsterdam, 1970.