Princeton University CS-TR-537-96

Lambda-Splitting: A Higher-Order Approach to Cross-Module Optimizations

Matthias Blume Andrew W. Appel
Princeton University, November 22, 1996

Abstract

We describe an algorithm for automatic inline expansion across module boundaries that works in the presence of higher-order functions and free variables; it rearranges bindings and scopes as necessary to move nonexpansive code from one module to another. We describe—and implement—the algorithm as transformations on λ-calculus. Our inliner interacts well with separate compilation and is efficient, robust, and practical enough for everyday use in the SML/NJ compiler. Inlining improves performance by 4–8% on existing code, and makes it possible to use much more data abstraction by consistently eliminating penalties for modularity.

1 Introduction

Abstraction and modular design of software promote clarity and provide clear lines along which large projects can be subdivided. But one often pays a large performance penalty for using abstraction. Cross-module inlining can bridge the gap between abstract design and high performance by transparently moving the border between compilation units.

We need not explain in detail the disadvantages of making the programmer inline-expand "by hand:" burdening the programmer, blurring modular design by exposing implementation details, and making it difficult to adjust the amount of inlining dynamically for development (recompiling) or "shrink-wrapping" mode.

But the automatic cross-module inlining schemes used to date have not treated free variables, nested scopes, higher-order functions, or link-time side effects from module-level initializers [DH88, CHT91, CMCH92]. They cannot move a function-body from module A to module B if the function has a free variable that is not exported from A and cannot be copied into B. This limits the generality of existing approaches, especially when applied to higher-order functional languages.

Our new technique, λ-splitting, is fully automatic. It exposes implementation details on a per-need basis only to the compiler of client code; abstraction and modularity are never compromised at the source level. Furthermore, by tuning a few compile-time parameters one can adjust the aggressiveness of cross-module inlining or turn it off completely.

In contrast to previous experiments [Sch77, CHT91, CMCH92] that did not explain how to preserve efficient separate compilation while inlining, our technique is fully integrated with SML/NJ's separate compilation system: it cleanly exports inlinable portions of one compilation unit through the binary object file into the importing module.

2 Example

Consider a simple procedure, which maintains the maximum of the values it has been presented with so far. We parameterize the implementation so it can be instantiated with different comparison predicates `lt`.

```
fun extremeFun (lt, x0) =
let val e = ref x0
    fun get () = !e
    fun check0 x =
        if lt (!e, x) then e := x
        else ()
    fun check [] = ()
      | check (h :: t) =
        (check0 h; check t)
in (get, check, lt) end
```

It is often desirable to inline function calls, especially if the functions are small and calls occur frequently. This is true regardless of whether the function in question is defined within the same or a different compilation unit. Let our example above be one compilation unit, while another unit instantiates `extremeFun` in order to be able to use the resulting procedures `check` and `get`:

```
val (get, check, lt) =
  extremeFun (op < , 10000);
...
```

Calls to `get` are really just accesses to reference cell `e`; we would like to inline-expand `get`, but `e` is not exported—presumably because we did not want it to be mutated other than via `check`. But after type-checking, the compiler should be allowed to inline the function.

In this paper we present a code transformation that achieves this effect even though `e` is originally not exported *and* everything is parameterized and therefore hidden within the body of `extremeFun`.

The transformation, called λ-splitting, takes one compilation unit and divides it into two parts. The first part contains those sections of the code that cannot or should not be inlined. Usually this happens with *expansive* code—expressions that actually compute something and are not just values—which must not be duplicated. We use the subscript e to label the expansive portion of a split. Code examples are marked with the letter `E`.

The other, inlinable part of a split—marked with subscript i or a letter `I`—contains no expansive code, only *values* (variables, constants, λ-abstractions) and other effect-free code (records, immutable vectors, datatype constructors). It will be compiled together with its client code.

In effect, λ-splitting redraws the lines between compilation units and lets an existing local optimizer look beyond the boundaries of the original compilation units. In our example the transformation will divide `extremeFun` into `extremeE` and `extremeI`:

```
fun extremeE (x0, check0Fun) =
let val e = ref x0
    val check0 = check0Fun e
    fun check [] = ()
      | check (h :: t) =
        (check0 h; check t)
in (e, check) end
```

The expansive portion `extremeE` is compiled to machine code; the inlineable portion `extremeI` is kept in symbolic form and will be copied verbatim into the client's code:

```
local
fun extremeI (lt, x0) =
let fun check0Fun e =
    let fun check0 x =
          if lt (!e, x) then e := x
          else ()
    in check0 end
    val (e, check) =
        extremeE (x0, check0Fun)
    fun get () = !e
in (get, check, lt) end
in
val (get, check, lt) =
    extremeI (op < , 10000)
...
end
```

Finally, a few straightforward reductions turn this into:

```
val (e, check) =
    extremeE (10000,
      fn e => fn (x: int) =>
          if !e < x then e := x
          else ())
fun get () = !e
val lt = (op <): int * int -> bool
...
```

Thus, λ-splitting achieves the following effects:

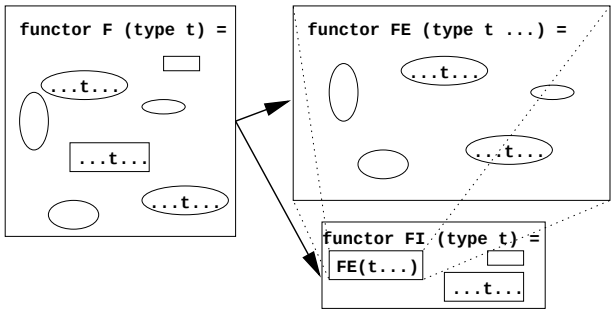


Figure 1: **Functor splitting.** We propagate type `t` through the functor into the client by copying `FI` into each client and β-reducing it. But `F` contains an expansive portion that cannot be copied; this is abstracted as `FE`. `F`, and `FE`, may be very large, but only the small inlinable portion `FI` gets copied into clients.

1. `extremeE` exports reference cell `e`.
2. The compiler can now rewrite invocations of `get` in the client as accesses to `e`.
3. Information about the fact that `lt` is the same as `op <` has not been lost on its way through the body of `extremeFun`.
4. The function `check0` is lifted out of `extremeE` and put into `extremeI` so that the actual comparison < can be inlined.

Our example shows what can be gained by a higher-order inlining algorithm. Previous approaches would not inline `get`, `check`, and `lt` because they do not appear at top level.

λ-splitting deals with parameterized modules in a general way. A functor can be taken apart almost arbitrarily (Figure 1) as long as the combined effect of its pieces taken together remains the same.

3 Separate compilation and linking

Standard ML has no circular dependencies among compilation units. Separate compilation and linking can be modeled using nonrecursive higher-order functions. This allows us to view cross-module[1] inlining as a problem of manipulating such functions expressed in λ-calculus.

Appel and MacQueen [AM94] describe how Standard ML of New Jersey implements separate compilation. Since this is the basis for our approach we will summarize it here:

A compilation unit consists of a number of definitions for types, values, signatures, structures, and functors. Types and signatures are relevant only for dealing with static semantics and can be ignored as far as linking, a dynamic concept, is concerned. The SML/NJ intermediate representation (λ-language [App92]) represents structures as records, and functors as functions, and does not distinguish between the core and module languages. Each compilation unit is turned into a λ-expression, where references to identifiers from other compilation units appear as free variables. By abstracting over those variables we get a closed expression, which can now be compiled to machine code without need to refer to any kind of context information.

Thus, a compilation unit becomes a function taking imported values to exported values. Exported values are collected in a record which is the result of executing the unit. Another unit, which imports those values, simply takes that record as a function argument.

For example, a unit consisting of definition

```
val a = 1 and b = 2
```

would be represented as `fun A () = (1, 2)`. The compiler remembers that `a` sits in the first position and `b` in the second. Similarly,

```
val c = 3 and d = 4
```

becomes `fun B () = (3, 4)`. Now, if we compile

```
val e = a * b + c - d
and f = a + c
```

the compiler will construct a λ-expression of the form

```
fun C [A, B] =
let val (a, b) = A
    val (c, d) = B
in (a * b + c - d, a + c) end
```

[1] In this paper we use the term *module* to always refer to compilation units, not to elements of the SML module language.

4 λ-splitting

For simplicity let us ignore the issue of importing from multiple sources, so we need not deal with vectors of imports. The code for a unit U is a closed function mapping imports to exports: $U : I \to E$. The objective of λ-splitting is to choose a triplet $(T, U_i : T \to E, U_e : I \to T)$, for which the composition $U_i \circ U_e$ yields U again. T is used to describe how U_e and U_i are interfaced to each other. We call the triplet a *B-decomposition*[2] or simply a *split*. For example, a trivial split for any $U : I \to E$ is (E, id_E, U), where $\mathrm{id}_E : E \to E$ is the identity function on E.

Now consider two units, where the second imports the exports of the first: $U^1 : I^1 \to E^1$ and $U^2 : I^2 \to E^2$, $E^1 = I^2$. The meaning of the program comprised of U^1 and U^2 can be understood in terms of their composition $U^2 \circ U^1$. For any split (T, U_i^1, U_e^1) this is the same as $U^2 \circ (U_i^1 \circ U_e^1)$, and by associativity $(U^2 \circ U_i^1) \circ U_e^1$. In other words, we can compile U_e^1 as one unit, $U^2 \circ U_i^1$ as the other, and still obtain the correct result. Cross-module inlining can therefore be achieved by cleverly placing the things to be inlined by U^2 into U_i^1.

In general, U^1 will be referred to by several other units, and we must be careful with what goes into U_i^1, because this code will be duplicated in each one. Therefore, our algorithm never places expansive code into the i-portion of a split.

4.1 The basic algorithm

Our approach to cross-module inlining should be applicable to a wide range of compilers and intermediate languages based on λ-calculi [KKR$^+$86, Pey87]. We have implemented a prototype in SML/NJ (version 109.19).

In SML/NJ a compilation unit translates into a sequence of nested variable bindings via `LET`- and `FIX`-expressions. A `LET`-expression binds one λ-variable to some value computed by an arbitrary expression, while `FIX` recursively binds one or more variables to function values. The innermost body of the innermost `LET` or `FIX` then builds a record of exported values. The entire construction is finally wrapped into a λ-abstraction together with some code for selecting values for the source code's free variables from the argument.

Eventually any piece of code will be of the following general form, where $\{w_1, \ldots, w_k\}$ is a subset of $\{v_1, \ldots, v_n\}$:

```
fn v0 =>
let val v1 = E1 in
  let val v2 = E2 in
    ...
      let val vn = En in
        (w1, w2, ..., wk)
      end
    ...
  end
end
```

As we will explain later, a simple λ-contract pass moves many bindings to top level, so they appear as v_i instead of being hidden inside the E_i.

The next step is to select a set of indices from $\{1, \ldots, n\}$, such that for every element j we are willing to move binding `val vj = Ej` into U_i. To avoid the duplication of side-effects we do not include any expansive `Ej`. Aside from this condition there is no further restriction on choosing from the remaining bindings.

Once a set of bindings for constructing U_i is chosen, we then need to fill in the remaining slots of the triplet $(T, U_i : T \to E, U_e : I \to T)$. Obviously, T needs to communicate values for all the free variables of U_i's body. These free variables will always be a subset of $\{v_0, v_1, \ldots, v_n\}$, so they are readily available at the time when the original code for U would have constructed its export record. Furthermore, since only non-expansive code has been placed into U_i it does no harm to duplicate it in U_e. Therefore, the task of constructing U_e becomes surprisingly simple: We can use U almost unchanged; only the export record has to be replaced with a record holding

[2] B is the composition combinator $\lambda f \lambda g \lambda x . f(gx)$.

said free variables. Of course, some of the bindings in U_e may become unnecessary. But the intramodular optimizer will delete unnecessary bindings.

4.2 A-normal form

To break large expressions into inlinable pieces we use a variant of A-normal form [FSDF93], in which every intermediate result is explicitly bound to a variable, as our calculus for λ-splitting. Continuation-passing style [Ste78, App92] would do as well; but the SML/NJ compiler transforms to CPS at a late stage after type information is discarded. We would rather preserve the ability to do monomorphic instantiation of polymorphic functions [SA95] *after* cross-module inlining has been done.

As an example of how A-normal form helps inlining, consider that the expression bound to `x` in

```
let val x = (1, ref 0)
    val y = 2
in (x, y) end
```

is expansive, because `ref 0` is expansive. But in A-normal form we have:

```
let val tmp = ref 0
    val x = (1, tmp)
    val y = 2
in (x, y) end
```

In this program `x` is bound to a non-expansive expression, which can very well be moved into the i-section of the split, thus exposing the 1 in the first field of the tuple. Thus, we can move things into U_i that would have been considered expansive.

4.3 Large functions

Unfortunately, abstractions are values, so large functions will be large even in A-normal form. Inlining small functions is part of the overall goal, but big functions pose a problem. While it is possible to move the entire code of any function into U_i without sacrificing correctness, we cannot do this indiscriminately because that would severely inflate the code. On the other hand, when it is not feasible to move the entire function, then we would like to make some of its parts available for inlining (see figure 1). After all, functors are encoded as functions, and highly functorized code could hardly benefit at all from cross-module inlining if we do not solve this problem. As we will explain, the λ-splitting algorithm can be applied recursively to function bodies.

4.4 λ-contract

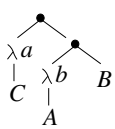
We found it useful to run a simple optimizer—λ-contract—over the code before attempting to split it. By un-nesting `LET`-bindings and performing other code rearrangements it brings the code into the general form expected by the splitting algorithm. It also performs value propagation, β-contractions, and some dead code elimination. After the code has been straightened out we can rely on simple syntactic cues when looking for things to place into U_i.

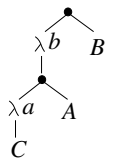
For example, if the right-hand side of a variable binding is another binding form itself, then the order of the two can be exchanged:

```
let val a =
  let val b = B
  in A end
in C end
```

⇒

```
let val b = B
in let val a = A
   in C end
end
```





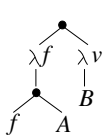
No α-conversion is necessary here because variable names are guaranteed to be distinct.

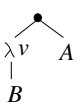
When abstraction and application are not adjacent, a β-reduction can make the `LET`-binding apparent:

```
let val f =
  fn v => B
in f A end
```

⇒

```
let val v = A
in B end
```



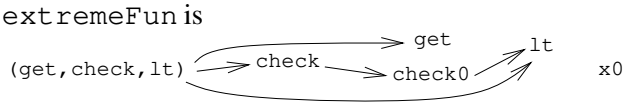


Such transformations move more bindings to "top

level", which improves the performance of λ-splitting.

4.5 B-decomposition

Some bindings never need to be moved to U_i because the variables they bind are not referenced. For a binding to be useful in U_i it must be reachable in the directed use-definition graph, which contains an edge going from every use of a variable to its definition, and the root node corresponds to the expression that constructs the export record. The use-def graph for `extremeFun` is



where there is no node for `e` because `ref x0` is expansive.

To avoid code blowup, we will put into U_i only a subset of the reachable portion of the graph. Our algorithm for constructing U_i proceeds as follows:

1. Calculate the initial set of bindings that are available for inlining: all non-expansive bindings to non-functions, all bindings to small functions, and bindings derived from recursively splitting large nonrecursive functions. "Small" is defined by a tunable parameter to the algorithm.

2. Construct the use-definition graph for this set.

3. Perform a breadth-first search on the graph starting at the root (the export record). Within each level of the graph, the search is biased to favor smaller expressions. It stops when either the entire graph has been traversed or the total size of nodes searched exceeds a pre-set threshold. This step automatically eliminates all dead definitions. It favors variables that are used by shorter use-def chains from the root node; we call this a *distance heuristic*. It also provides a cap for the total size of U_i, thereby preventing nonlinear increases in total code size.

4. The nodes selected by the breadth-first search are sorted into topological order using depth-first search. This is necessary for constructing a valid program, where uses of variables only occur within their scope.

5. The remaining free variables will not be inlined; they will be calculated inside U_e and their values will be passed at link-time (as function arguments) to the client module.

5 Functions

Functions are still a problem. Those that correspond to functors of the source language are particularly important, but the more general the approach the better. Programmers in other languages with less sophisticated module systems might approximate functors using regular functions.

Standard ML provides only first-order functors, but SML/NJ takes this a step further and offers a higher-order module system [Mac90, MT94]. Functors can be defined within functors, they can be passed as functor arguments, and they may be returned as part of functor instantiations. Therefore, in the intermediate representation there really is not much of a difference between ordinary functions and functors.

5.1 Recursive functions

Our algorithm does not attempt to take recursion apart; it either moves entire clusters of mutually recursive functions into U_i when the cluster as a whole is deemed small enough, or it leaves them completely alone. In order to be able to do this at as fine a granularity as possible we calculate strongly connected components of the use-definition graph for every `FIX`. Each component is then considered separately.

In most cases, a strongly connected component of functions represents a genuine loop or similar repetitive computation. This should not be spread across multiple compilation units, because the overhead of

cross-module function calls would be amplified considerably.

5.2 Nonrecursive functions and functors

B-decomposition works because compilation units are represented as functions. One could think of these compiler-constructed functions as implicit functors. From this observation we derive our idea for dealing with actual functors—we split them as well. The resulting e- and i-parts will be placed into U_e and U_i respectively.

B-decomposition again

As a first attempt we use B-decomposition recursively. In our introductory example, if we assume that `check0` and `check` are too big, then the "functor" `extremeFun` would be split like this:

```
fun extremeE (lt, x0) =
let val e = ref x0
    fun get () = !e (* dead *)
    fun check0 x =
        if lt (!e, x) then e := x
        else ()
    fun check [] = ()
      | check (h :: t) =
        (check0 h; check t)
in (e, check, lt) end

fun extremeI (e, check, lt) =
let fun get () = !e
in (get, check, lt) end

fun extremeFun args =
    extremeI (extremeE args)
```

Notice that `extremeFun` is the *B-composition* of `extremeI` and `extremeE`. Definitions for `extremeI` as well as the reconstructed `extremeFun` will be exported as part of U_i, allowing the client to inline calls to `get` and rewrite them as accesses to reference cell `e`.

S-decomposition

Functor splitting should provide three benefits:

1. Parts of the functor's body become part of U_i and can therefore be inlined into client code.
2. Existing simple connections between the functor's argument and result remain visible in U_i. Since the functor argument is supplied where the functor is instantiated, this will enable client code to inline parts of a functor's result if they correspond to inlinable parts of the argument.
3. Part or all of the functor's argument are inlined into the functor's body.

Unfortunately, only the first point is addressed by B-decomposition. This seems surprising at first, because it works so nicely at the level of compilation units. What is it that distinguishes explicit functors from the ones the compiler constructs implicitly? In SML, if a compilation unit U^2 refers to U^1, then for reasons of type checking U_1 must be compiled before U_2. This means that the code for constructing U^2's input is already available when U^2 is compiled. In other words, we really compile $U^2 \circ U_i^1$ and not U^2.

With functors the situation is different. Functors are compiled separately from the code that constructs their arguments. But we want to inline things from functor arguments into functor bodies; and from functor arguments into functor results (for clients to use). B-decomposition inlines things only from functor bodies into functor results.

A variation on B-decomposition—we call it S-decomposition—at least provides a partial solution. The functor argument is known at the time when `extremeI` is invoked, so instead of first funneling it all the way through `extremeE` we can simply pass it directly to `extremeI`. Therefore, the S-decomposition for $F : I \to E$ is

$$(T, F_i : I \to T \to E, F_e : I \to T)$$

such that $F = SF_iF_e$. S denotes the stronger version of composition, which distributes the argument to both functions: $Sfgx = fx(gx)$.

Under S-decomposition our example program turns into the following code, where F_i (here: `extremeI`) establishes an explicit link between the functor's argument and `lt`:

```
fun extremeE (lt, x0) =
let val e = ref x0
    fun get () = !e (* dead *)
    fun check0 x =
        if lt (!e, x) then e := x
        else ()
    fun check [] = ()
      | check (h :: t) =
        (check0 h; check t)
in (e, check) end

fun extremeI ((lt, x0), (e, check)) =
let fun get () = !e
in (get, check, lt) end

fun extremeFun args =
    extremeI (args, extremeE args)
```

Differences between our algorithms for B- and S-decomposition are minor. S-decomposition can use exactly the same methodology, in fact, share most of the implementation of B-decomposition as well. There is one small twist: At the point where the free variables of F_i are collected we explicitly exclude the original formal argument of F, because it will be passed to F_i directly.

$\underline{1}$-decomposition

Functor arguments can be inline-expanded only if they are used in the inlinable portion of the functor split. In our example even S-decomposition does not promote the definition of `check0` into `extremeI`. Therefore, the comparison predicate will not be inlined there. The only mention of `check0` is in `check`, which itself is not in `extremeI`.

Still, `check0` is bound to a value, so λ-splitting could factor it out and move it to `extremeI`, from which it will be passed *as an argument* to `extremeE`. The resulting code was shown in the introduction (section 1).

When splitting a functor F we can generally leave it up to F_i how F_e is invoked. F_e is compiled somewhere else and will be treated as a black box. Still, its interface can be chosen arbitrarily. This can be modeled by passing F_e itself, not its result, to F_i. Thus, a $\underline{1}$-decomposition of $F : I \to E$ is a triplet

$$(T, U, F_i : T \to I \to E, F_e : T)$$

such that $F = \underline{1} F_i F_e = F_i F_e$.[3] Normally T will itself be of the form $T_1 \to T_2$ for some T_1 and T_2. Both B- and S-decomposition can be viewed as special cases of this more general approach.

6 Implementation and experimental results

We implemented λ-splitting (B-decomposition for compilation units and S-decomposition for functions) in SML/NJ. For the general framework only very small modifications to existing code in the compiler were necessary. In particular, no changes had to be made to either the front-end including symbol table management, the optimizer, or the various architecture-specific back-ends. λ-splitting fits neatly between existing other compiler phases.

We changed the format of object files to accommodate symbolic λ-code and modified SML/NJ's compilation manager [Blu95]. Dependency analysis now has to take into account that compilation units are connected via static environments (i.e., symbol tables) and inlinable code. This does not change the shape of the dependency graph, but it has an impact on how modifications to one unit propagate to other, dependent units.

Many widely used SML benchmarks cannot be used to measure cross-module inlining because they contain only a single compilation unit. Therefore, we applied λ-splitting to SML/NJ's compiler itself and used several of its phases as our benchmarks. The compiler is a 100,000 line SML program comprised of more than 600 individual compilation units (source

[3] $\underline{1} = \lambda f.\lambda x.(f\,x)$ is the Church-numeral **1**.

files). With a successful test of this magnitude we are confident of the correctness and viability of the implementation.

6.1 Problems with λ-contract

In our prototype the introduction of λ-contract had one disturbing effect: In some cases execution slows down (Figure 2). The numbers indicate that whether or not λ-contract is advantageous on its own depends heavily on architecture and benchmark. In our case the overall result is dominated by `elaborate`, which improves on the Pentium but slows down on the Alpha.

To explain this we note that it is possible for β-contractions to increase the size of certain closures, and if such closures are constructed repeatedly—for example in a loop—then execution time may go up. We hope to improve the situation with a better closure-conversion algorithm.

In order to obtain a better idea of how much cross-module inlining itself affects overall performance, we will use SML/NJ with λ-contract enabled as the baseline for our comparisons. This compiler already does aggressive intramodular inlining [App92]; we measure only the increased performance from inlining across module boundaries and from pulling inlinable code out of higher-order functions.

6.2 Timing results from λ-splitting

Figure 3 shows timing results for running the benchmarks with λ-splitting enabled. The numbers reported correspond to compiler passes that took at least one second of user time. The net speedup of 8% for the Alpha is reduced to only 4% if we take the effects of λ-contract into account.

On the Pentium we find a somewhat different picture. λ-contract seems to have a positive net impact here, although some phases of the compiler also suffer a significant slowdown. Overall speedup due to cross-module inlining is at 6% with and 5% without counting effects from λ-contract.

6.3 SML/NJ without ad-hoc inlining

Cross-module inlining is so crucial for performance that SML/NJ's implementors decided to compromise abstraction and modularity in some places, because for a selection of the most important operations inline-expansion is absolutely necessary. Unfortunately, the mechanism only applies to a fixed set of pre-selected values, and it also is not robust with respect to simple source code modifications. For example, by only adding a few semantics-preserving signature constraints—something the unsuspecting user might do by accident—it can easily be disabled.

Nevertheless, the numbers reported so far show *additional* speedup, because both λ-splitting *and* ad-hoc inlining were in effect at the same time. Clearly, there is large overlap in what these two methods can achieve.

As an experiment we changed the compiler sources to eliminate old-style ad-hoc inlining. The results (Figure 4) indicate that a fully automatic technique like λ-splitting can provide benefits close to those of previous ad-hoc approaches, while at the same time being more robust and less intrusive.

Comparing raw numbers shows that λ-splitting still fares a little worse than the special-case solution (about 5%). However, the contest is not entirely fair. The ad-hoc approach implemented in SML/NJ not only propagates primitive operators from compilation unit to compilation unit, it also forces the *intra*modular optimizer to inline-expand every single use. In effect, it overrides the decisions the optimizer would have made on its own. λ-splitting does not do that. Instead, it relies entirely on the existing optimizer's heuristics. In our experiment all the operators handled by the old approach were also propagated by the new λ-splitting. The discrepancy in performance can therefore be explained by shortcomings in intramodular optimization technology.

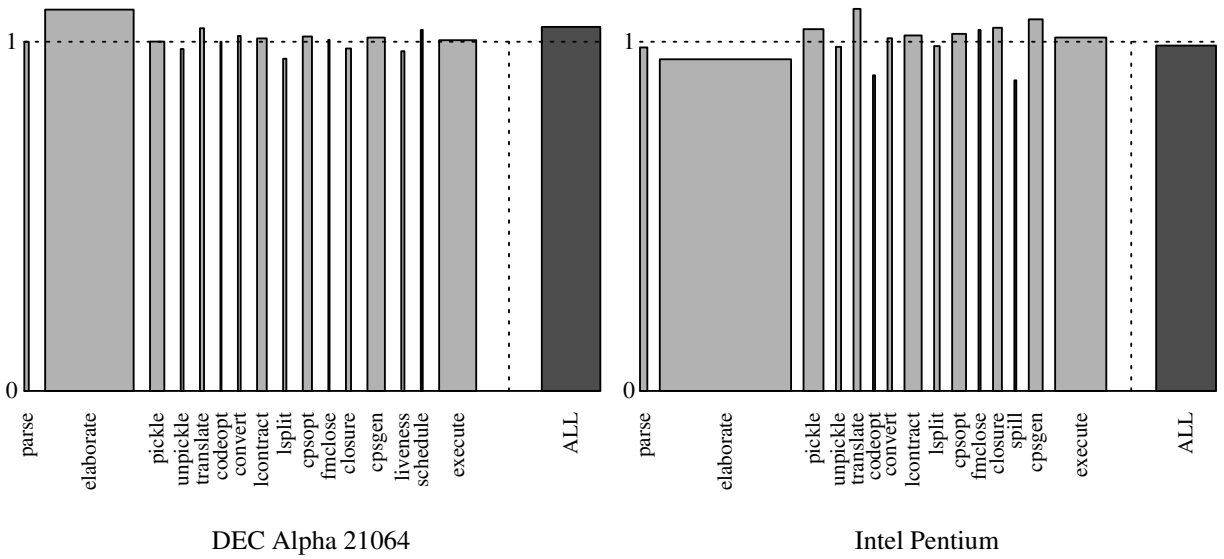


Figure 2: **λ-contract on Alpha and Pentium.** Individual compiler passes of SML/NJ—light bars in the graphs—serve as benchmarks for our study. With the exception of the dark bar on the right, which shows overall performance of the whole compiler, width was chosen proportional to the time spent in each phase. The heights of the bars indicate speedup or slowdown caused by λ-contract (shorter is better). λ-contract is not the focus of this work; it is an auxiliary transformation that allows our splitting algorithm to find many more opportunities for cross-module inlining. The numbers here show that depending on architecture and benchmark λ-contract has sometimes positive, sometimes negative effects on running time.

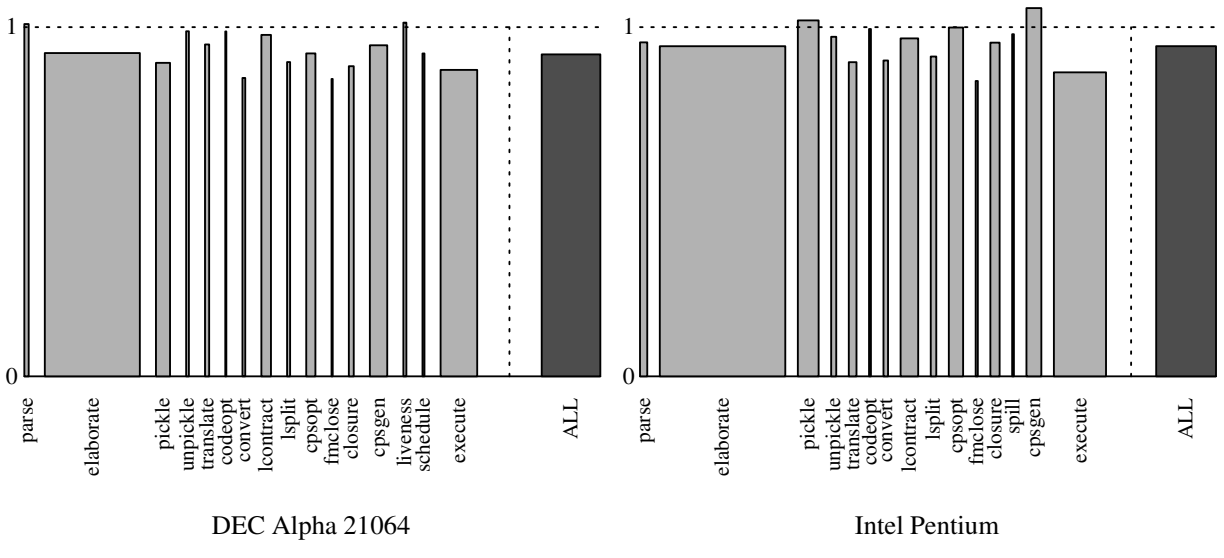
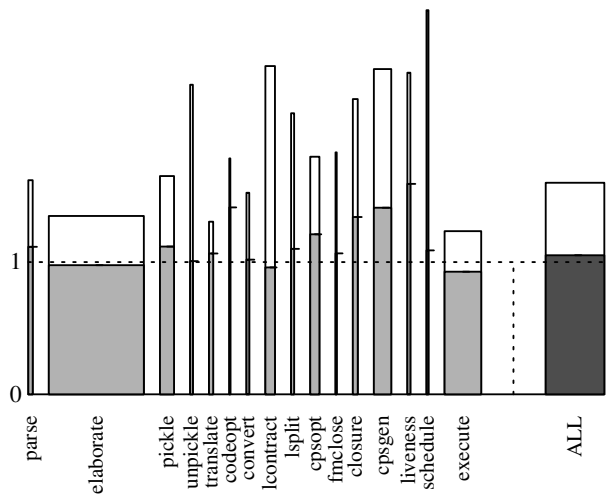


Figure 3: **Improvement due to cross-module inlining.** λ-splitting yields improvement on almost all benchmarks, and improves total execution time by 8% on the Alpha, 5% on the Pentium.



DEC Alpha 21064

Figure 4: **Inlining applied to a cleanly modularized program.** We reinserted signature constraints and abstractions that had been removed by the efficiency-conscious software engineers. The white bars show how much this would have cost without cross-module inlining. The dark bars show that λ-splitting recovers essentially all of the performance cost of modular abstraction.

6.4 Compile-time cost of inlining

In our preliminary implementation, the λ-contract phase takes 4.5% of total execution time, and λ-split takes 1.5% (this can be seen in figure 3). In addition, the intramodule CPS-optimization phase slows down because it has more (useful!) work to do. Compile time increases by 7% overall.

Our implementation of λ-contract could undoubtedly be improved; but even so, the cost of our cross-module inliner seems entirely reasonable.

7 Conclusions

λ-calculus is a powerful language for expressing programs and program transformation. It is no surprise that cross-module inlining, which is such a transformation, will benefit from being cast in this framework.

Compilation units can be viewed as functions. The process of linking applies them to imports, thus obtaining exports. By splitting the functions into expansive and inlinable parts we are able to regroup the code in such a way that cross-module inlining turns into the task of performing ordinary intramodular optimizations. The functions that represent compilation units are split in such a way that algebraic function composition (the B combinator) can be used to recombine the parts.

Higher-order constructions such as functions or SML's functors can be split recursively, thus allowing them to be inlined partially. This approach gives us the benefits of inlining in the presence of parameterized modules and still prevents excessive code growth.

For functors B-decomposition alone is relatively weak. It allows client code to inline-expand only parts of the functor's body. S-decomposition is a simple variation, which enables client code also to inline-expand parts of the functor's argument in the case that the functor re-exports them. Here the original functor can be reconstructed by applying the S combinator to its parts.

An even stronger version, $\underline{1}$-decomposition, is a generalization of the other two techniques. It can be used to inline the actual functor argument into parts of the functor's body even if those parts are not exported themselves. We plan to complete an implementation for this in our prototype soon.

Our experiments show modest speedup for several large benchmarks that had already been hand-optimized. If we start with cleanly structured code, then gains are substantial and competetive with previous ad-hoc approaches that resorted to sacrificing abstraction and modularity.

8 Future work

Now that the penalties for using abstract data types across module boundaries have been eliminated, we can clean up all the ad-hoc inlining of arithmetic primitives that pervade SML/NJ.

But we can do even more. Standard ML has al-

ways had the problem that efficient implementations of datatypes break down at functor-parameter boundaries [App93]. By representing *concrete* data types as *abstract* data types, so that constructors and pattern-matching become function calls, we can solve this problem. Without inlining this would have been totally impractical, but we expect inlining will eliminate any penalty (except in rare cases at functor-parameter boundaries, where a previously *wrong* case now becomes correct but inefficient).

References

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, pages 13–23. ACM Press, June 1994.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[App93] Andrew W. Appel. A critique of Standard ML. *J. Functional Programming*, 3(4):391–430, 1993.

[Blu95] Matthias Blume. Standard ML of New Jersey compilation manager. manual accompanying SML/NJ software, 1995.

[CHT91] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[DH88] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *1993 Conference on Programming Language Design and Implementation.*, pages 21–25, June 1993.

[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):219–33, July 1986.

[Mac90] David B. MacQueen. A higher-order type system for functional programming. In *Research Topics in Functional Programming*, pages 353–68, Reading, MA, 1990. Addison-Wesley.

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Proc. European Symposium on Programming (ESOP'94)*, pages 409–423, April 1994.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York, 1987.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc 1995 ACM Conf. on Programming Language Design and Implementation*, pages 116–129. ACM Press, 1995.

[Sch77] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

[Ste78] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.