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

Matthias Blume　　Andrew W. Appel
Princeton University,　April 2, 1997

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.

One might think of inlining functions after closure-conversion, when functions have no free variables, but this does not solve the problem because the free variables have become function arguments, and the callers in other scopes have no access to the corresponding actual-parameter values.

Our new technique, λ-splitting, is fully automatic. It exposes implementation details on an as-needed 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] 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.

Our approach should be applicable to a wide range of languages and compilers with intermediate languages based on λ-calculi [KKR+86, Pey87]. λ-calculus has the advantage over competing intermediate representations of being a well-studied logical system. It also proved to be very convenient for expressing the necessary transformations in our prototype. Although the emphasis on λ-calculus favors functional programming languages as a target for our technique, the ideas could be adapted to other languages and implementations. See appendix A for a description of the intermediate language used.

Our compiler makes cross-moduling inlining decisions automatically. This frees the programmer from the burden of having to worry about many "micro-optimizations." The relationship between functions to be inlined and source-level constructs is not always obvious to the programmer, especially in implementations that pass type information at runtime, and which therefore may encode polymorphism as abstraction and type specialization as function application [Oho92, Tol94, HM95].

Resource-conscious programmers often like to say explicitly that they believe certain procedure calls should be inline-expanded. Performance hints from a profiling feedback system [CMCH92] can play a similarly important role. Our framework can easily be adapted to take account of external hints.

λ-splitting moves function bodies from one module to another, but ensuring that these functions are then inlined *within* the importing module is up to the existing intramodule inliner [App92], which does a good but not perfect job. Since our intermodule inlining algorithm preserves separate compilation, it must make decisions without seeing the client modules, so we rely on simple syntactic cues instead of powerful dataflow analyses [JW96] that could aid inlining decisions.

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.

The transformation presented here achieves this effect even though `e` is originally not exported *and* everything is parameterized and therefore hidden within the body of `extremeFun`. λ-splitting takes one compilation unit and rewrites it as two parts. The first part contains those sections of the code that cannot or should not be inlined, while the other will be made available for being inline-expanded into client modules.

It is common for a compilation unit to have more than one client, which means that the inlinable portion of that unit will be duplicated. For correctness it is necessary to avoid the duplication of side effects, and to prevent excessive growth in total code size we must keep the inlinable portion small. Appendix C shows the measure that we use to estimate code size.

Of course, at compile time the presence of side effects cannot be decided precisely, so we rely on a safe approximation that can easily be characterized syntactically. It is very similar to the notion of *expansiveness*. Accordingly, we call the first part of a split the *expansive portion*—even though in general it will also contain a lot of non-expansive code—and use the subscript e as a label; code examples are marked with the letter `E`.

The second, 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. A formal definition of what we consider "permitted" to be inlined is given in appendix B. 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, assuming the size of `extremeFun` to exceed the limit that we impose on inlinable code, the transformation will divide it 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
```

The intramodule inliner reduces this client 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:

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.

Functions such as `extremeFun` in our example really play the role of parameterized modules. In ML terminology these are called *functors*, and there is special syntax (which we didn't use here to avoid complicating the presentation) together with ramifications for the static semantics of the language.

Standard ML provides only first-order functors, but this has been taken a step further in SML/NJ by offering 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

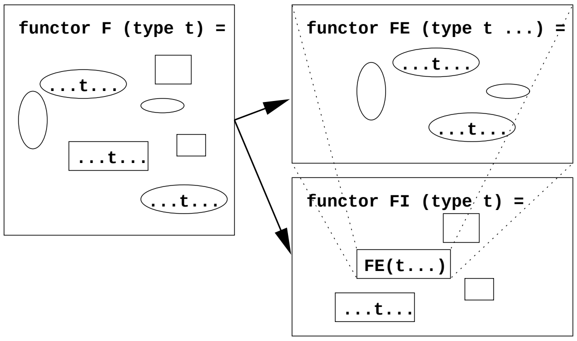


Figure 1: **Functor splitting.** We propagate type `t` through the functor into the client by copying `FI` into each client and β-reducing it. But `F` contains an expansive portion (ovals) that cannot be copied; this is abstracted as `FE`. `F`, and `FE`, may be very large, but only the inlinable portion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

instantiations. Therefore, in the intermediate representation there really is not much of a difference between ordinary functions and functors, because they are represented the same.

Functions are non-expansive themselves, so one could—without sacrificing correctness—place them into the inlinable portion as a whole, thereby effectively delaying the compilation of parameterized modules until they are instantiated. This would maximize the compiler's chances of generating optimized code taylored to each individual instantiation, but it also duplicates all of the functor's body every time it is used—potentially leading to intolerable increases in size. But large functors sometimes have a small "core" that should be delayed and inlined for efficiency, while the remainder can be compiled where it is declared. λ-splitting is able to accomplish that (almost—see the end of section 5.2).

3 Separate compilation and linking

The meaning of any compilation unit can be understood as a function that maps meanings for its imports to meanings for its exports. This allows us to view cross-module[1] inlining as a problem of manipulating such functions.

Imports correspond to the free identifiers in the source program; new definitions that other compilation units can refer to are considered exports. We can assign meaning to a system of several compilation units by solving the corresponding system of equations, which describe the import-export relations. In languages that permit circular dependencies among modules this will in general involve calculating a fixed point; without circularities it becomes much simpler.

Standard ML has no circular dependencies among compilation units. Separate compilation and linking can be modeled using nonrecursive higher-order functions. Type checking already imposes an ordering on compilation steps, guaranteeing that whenenver a module B refers to something exported from module A, then A must be compiled before B, thereby ensuring that inlining information from A will be available at the time when B is compiled.

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:

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

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 further 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, a unit consisting of the definitions

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

becomes `fun B () = (3, 4)`. Now, if we compile a third unit containing

```
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
```

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 $(T,\ U_i : T \to E,\ U_e : I \to T)$, such that the composition $U_i \circ U_e$ yields U again. Here T denotes the domain of possible values that are passed from U_e to U_i and corresponds to a type if the intermediate language is typed. 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

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

this code will be duplicated in each one. Therefore, our algorithm never places expansive code or code that is deemed too big into the i-portion of a split.

The idea of splitting functions into pieces can also be used to facilitate intramoduler optimizations like "call forwarding" [BDGK94] or partial inlining [Gou94].

4.1 The basic algorithm

In SML/NJ a compilation unit translates into a sequence of nested variable bindings (`let`-bindings), some of which can be recursive. The rightmost body of the rightmost `let`-expression 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 the code corresponding to any compilation unit U will be of the following general form, where the set $\{w_1, \ldots, w_k\}$ is a subset of $\{v_1, \ldots, v_n\}$:

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

To construct the code for U_e and U_i we first select all j such that the bindings `val vj = Ej` are guaranteed to be effect-free. This will ensure that no side effects will be duplicated by our algorithm. A formal definition of what we consider "guaranteed to be effect-free" is given in appendix B. In most cases the set of bindings chosen in this step will be rather large, so we employ a combination of several heuristics to further reduce it.

The remaining bindings are then used to build U_i's body, and finally we need to find T and U_e to complete the split $(T,\ U_i : T \to E,\ U_e : I \to T)$. The elements of T must 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 there will be no harm in duplicating some or even all of 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 said free variables. Of course, some of the bindings in U_e may become unnecessary. But the intramodular optimizer will delete dead bindings.

4.2 A-normal form

To break large expressions into inlinable pieces we use a variant of A-normal form [FSDF93] as our calculus for λ-splitting. The property we are interested in is that every intermediate result will be explicitly bound to a variable, because this increases the number of bindings while on average reducing the size of the expressions being bound. Thus, splitting decisions can be made with finer granularity. The structure of the intermediate language described in appendix A automatically enforces this. 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 more code into U_i.

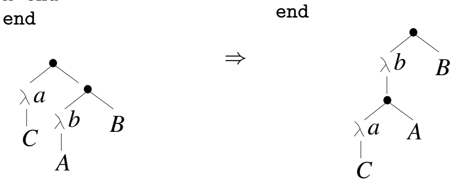
4.3 λ-contract

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 variable bindings to be exported in symbolic form as part of the inlinable portion of the split.

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. This technique is knows as *let-floating* [JPS96]:

```
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
```

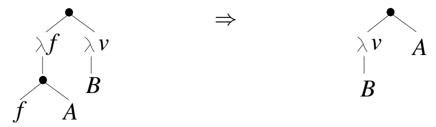


No α-conversion is necessary here because variable names are guaranteed to be distinct.

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.

It seems unfortunate that λ-contract duplicates some optimizations that are also performed by SML/NJ's CPS optimizer. We run the λ-splitter early to be able to take advantage of type information that is not present in later stages of the current SML/NJ compiler. Perhaps, in compilers where the intermediate language is typed at all times [TMC+96] λ-splitting can be moved to such a later stage, and the need for λ-contract and its associated redundancy can be eliminated.

4.4 Call counting

One major soure of inefficiencies related to separate compilation is the need for a generic function call protocol, which has to be used whenever the compiler is unable to consistently identify all call sites of a given function or all functions callable at a given call site. In particular, this is always the case when the function is defined in one compilation unit and used in another.

Cross-module inlining as proposed here is an attempt to improve the situation by moving function definitions into the modules where they are used or vice versa. But unless we abandon the idea of separate compilation completely there will always be some functions that are not defined where they are called. Therefore, there is a danger that cross-module inlining will make things worse than they were before! To illustrate this, let us consider the following scenario: Compilation unit B contains a call to a function `f`; `f` is defined in module A calls `g` as well as `h`, which are also defined in A.

```
(* module A *)
fun g () = ...
fun h () = ...
fun f () = (... g () ... h () ...)

(* module B *)
.... f () ...
```

If "for efficiency" we move only `f`'s definition into module B, then for every call to the original `f` that had to use the generic, non-optimized protocol we now have two such calls to `g` and `h`:

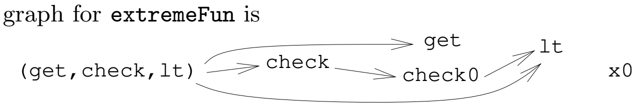
```
(* module A *)
fun g () = ...
fun h () = ...

(* module B *)
.... (... g () ... h () ...) ...
```

To avoid this effect we would like to keep track of the number of runtime function calls that use the non-optimized protocol for the reason that caller and callee are separated by module boundaries. This quantity cannot be calculated precisely, but we can use a static estimator $\mathcal{N}(\ldots)$, which is able to identify many of the bad cases [Blu97]. Moreover, inherent imprecisions do not have any bearing on correctness of the overall algorithm.

4.5 B-decomposition algorithm

Some bindings need not b appear in 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 where 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:

B-decompose($w = \lambda(v_1, \ldots, v_n)$. let *bindings* in $\vec{r}$) =
  $(E, A) \leftarrow$ Available(*bindings*)
  $I \leftarrow$ ToBeInlined$(A, \vec{r})$
  $I_{\text{ord}} \leftarrow$ the ordered sublist of E whose elements are $\in I$
  $(x_1, \ldots, x_k) \leftarrow \mathcal{F}$(let I_{ord} in $\vec{r}$)
  return ($w_e = \lambda(v_1, \ldots, v_n)$.let E in $(x_1, \ldots, x_k)$,
    $w_i = \lambda(x_1, \ldots, x_k)$.let I_{ord} in $\vec{r}$,
    w=$\lambda(v_1', \ldots, v_n')$. $w_i(w_e(v_1', \ldots, v_n'))$)

S-decompose($w = \lambda(v_1, \ldots, v_n)$. let *bindings* in $\vec{r}$) =
  $(E, A) \leftarrow$ Available(*bindings*)
  $I \leftarrow$ ToBeInlined$(A, \vec{r})$
  $I_{\text{ord}} \leftarrow$ the ordered sublist of E whose elements are $\in I$
  $(x_1, \ldots, x_k) \leftarrow \mathcal{F}$(let I_{ord} in $\vec{r}$) $\setminus \{v_1, \ldots, v_n\}$
  return ($w_e = \lambda(v_1, \ldots, v_n)$.let E in $(x_1, \ldots, x_k)$,
    $w_i = \lambda(v_1, \ldots, v_n).\lambda(x_1, \ldots, x_k)$.let I_{ord} in $\vec{r}$,
    $w = \lambda(v_1', \ldots, v_n')$.
      $w_i(v_1', \ldots, v_n')(w_e(v_1', \ldots, v_n'))$)

Available$(b_1, \ldots, b_m)$ =
  $E \leftarrow b_1, \ldots, b_m$
  for each $b \in E$
  if $\mathcal{S}(b) < K$
    $A \leftarrow A \cup \{b\}$
  else $(b_e, b_i, b') \leftarrow$ S-decompose(b)
    In the list E, replace b by b_e, b_i, b'
    $A \leftarrow A \cup \{b_i, b'\}$
  else if $\mathcal{P}(b)$
    $A \leftarrow A \cup \{b\}$
  return (E, A)

UseDef(b) =
  The set of bindings $v = M$ such that $v \in \mathcal{F}(b)$

ToBeInlined$(A, \vec{r})$ =
  $Q \leftarrow \{\vec{r}\}$
  $Q' \leftarrow \{\ \}$
  while $Q \neq \{\ \} \ \wedge \ \sum_{b \in B} \mathcal{S}(b) \ < \ K'$
    let b be the element of Q with smallest $\mathcal{S}(b)$
    $A \leftarrow A \setminus \{b\}$
    $Q \leftarrow Q \setminus \{b\}$
    $B \leftarrow B \cup \{b\}$
    $Q' \leftarrow Q' \cup$ (UseDef(b) $\cup A$)
    if $\mathcal{N}(B) < \mathcal{N}(I)$
      $I \leftarrow B$
    if $Q = \{\ \}$
      $Q \leftarrow Q'$
      $Q' \leftarrow \{\ \}$
  return I

Figure 2: λ-splitting algorithms

$\vec{r}$ is the expression that constructs the export record and serves as the starting point for the breadth-first search in function *ToBeInlined*.

E is the ordered list of bindings ot be kept in U_e; the intramodular optimizer may later delete dead bindings, but this is not part of the splitting algorithm.

A is the set of bindings available for inlining.

B is the current candidate for the inlinable part U_i.

I is the best candidate seen so far.

Q, Q' are the two parts of the breadth-first search queue. The separation enables us to implement a bias in favor of smaller bindings within each level of the breadth-first search tree.

K is the maximum size of any function body that can be inlined without being recursively split.

K' is the maximum size of the inlineable code U_i.

$\mathcal{F}$ computes the free variables of an expression (see appendix app:lang).

$\mathcal{P}$ tells whether an expression is nonexpansive (see appendix B).

$\mathcal{S}$ tells the size of an expression (see appendix C).

$\mathcal{N}$ approximates how many out-of-module calls a set of bindings makes.

Figure 3: Legend to the algorithm in figure 2

1. Calculate the initial set A 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. Code size is estimated using the measure $\mathcal{S}(\ldots)$ defined in appendix C; non-expansiveness is judged using the predicate $\mathcal{P}(\ldots)$ from appendix B. This part of the algorithm is the *Available* function of figure 2.

2. Construct the use-definition graph for this set (as in *UseDef* of figure 2).

3. Perform a breadth-first search on the graph starting at the root (the export record $\vec{r}$). Within each level of the graph, 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. The greedy approach 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. (See the *ToBeInlined* function of figure 2.)

4. From the space traversed by the greedy breadth-first search we pick the set of bindings for U_i that has the smallest call count $\mathcal{N}(\ldots)$.

5. The bindings in the set picked for constructing the body of U_i are used in the same order that they appeared in within U's code. This ensures correct scoping for bound variables.

6. Any 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 (see *B-decompose* in figure 2).

5 Functions

Abstractions are syntactic 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.

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 recursive `let`. 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 Functors and nonrecursive functions

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 and other large functions—we split them as well. The resulting expansive and inlinable parts will be placed into the corresponding expansive and inlinable portions of the surrounding large function or compilation unit.

For correctness we had to be careful that our splitting algorithm will never place expansive code into the inlinable portion of a compilation unit. When splitting functions this restriction doesn't exist, because side effects can only occur at the time the function is called. Therefore, side effects will be duplicated only if the function in question is explicitly invoked multiple times, which then makes this the correct behavior as mandated by the language definition.

Nevertheless, it is convenient to maintain the invariant that the inlinable portion of *any* split is nonexpansive. It frees us from having to pay attention to the ordering of effects within a function. Also, it guarantees that invocations of any inlinable portions themselves will be free of effects. Thus, it becomes possible to relax our notion of non-expansiveness (normally, function invocation is always considered expansive), so we can move such invocations into the inlinable portion of the enclosing function (appendix B).

Recursive decomposition

As a first attempt, the same technique—B-decomposition—that is used to split the functions representing entire compilation units can also be utilized when splitting functions too large to be placed into the inlinable portion. This idea recursively extends to large functions encountered within large functions and so forth.

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 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 propagate actual functor arguments to functor bodies, so they can be inline-expanded there; and functor arguments into functor results (for clients to use). B-decomposition propagates code only from functor bodies to functor results.

A variation on B-decomposition—we call it S-decomposition—addresses points 1 and 2 above. 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 (see the *S-decompose* function of figure 2).

$\underline{1}$-decomposition

Actual functor arguments can be inline-expanded in the functor body only if they are used in the inlinable portion of the functor split, because only then will both definition and use of the argument be available to the compiler at the same time. 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 syntactic 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 section 2.

When splitting a functor F we can generally leave it up to F_i how F_e is invoked. Even though F_e is compiled somewhere else and will be treated as a black box, its interface can still 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

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

such that $F = \underline{1}F_iF_e = F_iF_e$.[3] Normally T will itself be of the form $T_1 \to T_2$ for some T_1 and T_2.

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

The equality $F = F_i F_e$ is used to recover F from its pieces: any occurence of F in the client can be replaced with $F_i F_e$. But the code of F_i will then be available, so we can immediately β-reduce $F_i F_e$ and work with the result instead of F_i itself. We already did this when we presented the introductory example, which explains why `extremeE` is not being passed to `extremeI` as an argument but rather gets invoked directly.

Both B- and S-decompositions can be viewed as special cases of the more general approach of $\underline{1}$-decomposition. In particular, for any X-decomposition (T, F_i, F_e) of $F : I \to E$ (where X is either B or S) the triplet $(I \to T, XF_i, F_e)$ forms such a corresponding $\underline{1}$-decomposition.

We do not show an algorithm for $\underline{1}$-decomposition, because we have not found a good set of heuristics to separate the "core" of a function (or functor) from the non-inlineable body (as illustrated in figure 1). Therefore, our implementation cannot quite transform `extremeFun` as shown in section 2, but can transform it via B- and S-decomposition as shown in section 5.2.

6 Implementation and results

We implemented λ-splitting (B-decomposition for compilation units and recursive 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 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 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 4). The numbers indicate that whether or not λ-contract is advantageous on its own depends on architecture and benchmark. In our case the overall result is dominated by the `elaborate` benchmark, which improves on the Pentium but slows down on the Alpha.

To explain this we note that β-contractions can increase the size of certain closures, and if such closures are constructed repeatedly—for example in a loop—then execution time may go up:

```
let val f =
   fn x => g (a,b,c,d,e,x)
in let fun loop () =
      (...
        (fn y => f (z,y))
       ...loop())
   in
      loop ()
   end
end
```

In this example, let us replace the variable `f` in `(f (b, y))` with its definition:

```
let fun loop () =
   (...
     (fn y => g (a,b,c,d,e, (z, y)))
    ...loop())
in
   loop ()
end
```

The anonymous function inside `loop` now has seven free variables (`g`,`a`,`b`,`c`,`d`,`e`,`z`), while before it had only two (`f` and `z`). We hope to improve the situation with a better closure-conversion algorithm, much later in the compiler after λ-contract, λ-splitting, and intramodular optimization.

Even without β-reductions and with let-floating [JPS96] alone we sometimes found performance to be slightly diminished. Reordering `let`-bindings in itself does not change the size of any closure, it only moves bound variables closer to the expressions they are bound to. However, this may enable later phases of the compiler's CPS optimizer to recognize and reduce more β-redexes, which again may cause some closures to become bigger.

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 5 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.

We believe our results are not due to "random" variations in cache-conflict performance. Functional programs with properly tuned garbage collectors are not subject to many data-cache conflicts [Rei94, GA95], and informal inspection of the results did not reveal the kind of instruction-cache conflict variability [App92, p. 194] to which SML/NJ was subject on previous-generation architectures.

6.3 SML/NJ without ad-hoc inlining

Cross-module inlining is so crucial for performance that the implementors of SML/NJ decided to compromise abstraction and modularity in some places, because for a selection of the most important operations inline-expansion is absolutely necessary.

To achieve a limited form of cross-module inlining the compiler treated certain variable bindings specially: If the

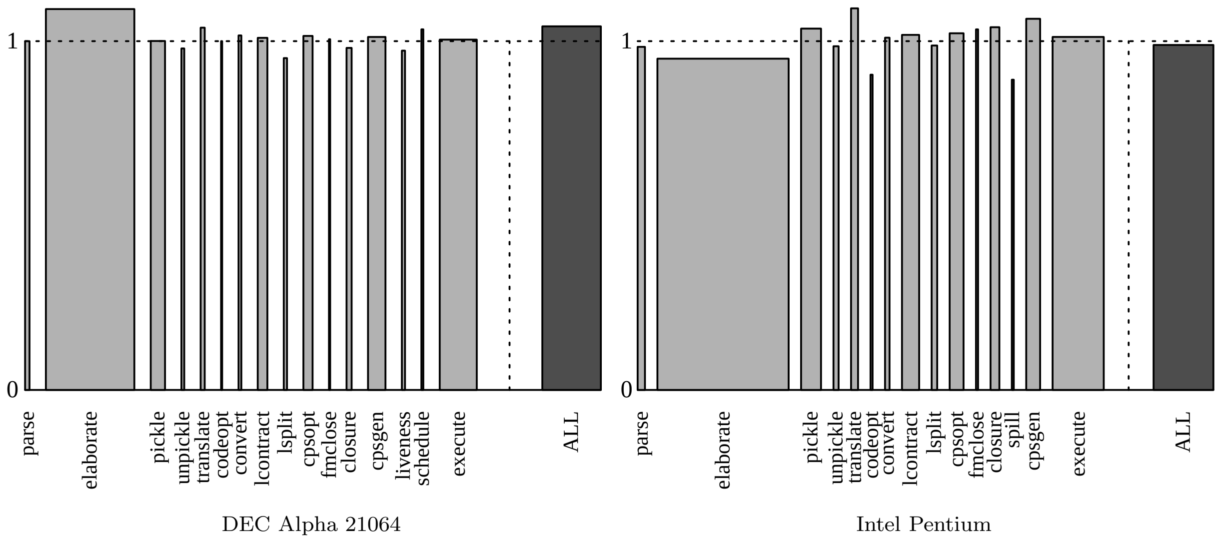


Figure 4: **λ-contract on Alpha and Pentium.** Individual compiler passes of SML/NJ—light bars in the graphs—serve as benchmarks for our study. With the exception of the dark bar on the right, which shows overall performance of the whole compiler, width was chosen proportional to the time spent in each 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.
The *lcontract* and *lsplit* bars graphically illustrate that the algorithms presented in this paper do not make up a large portion of compilation time.

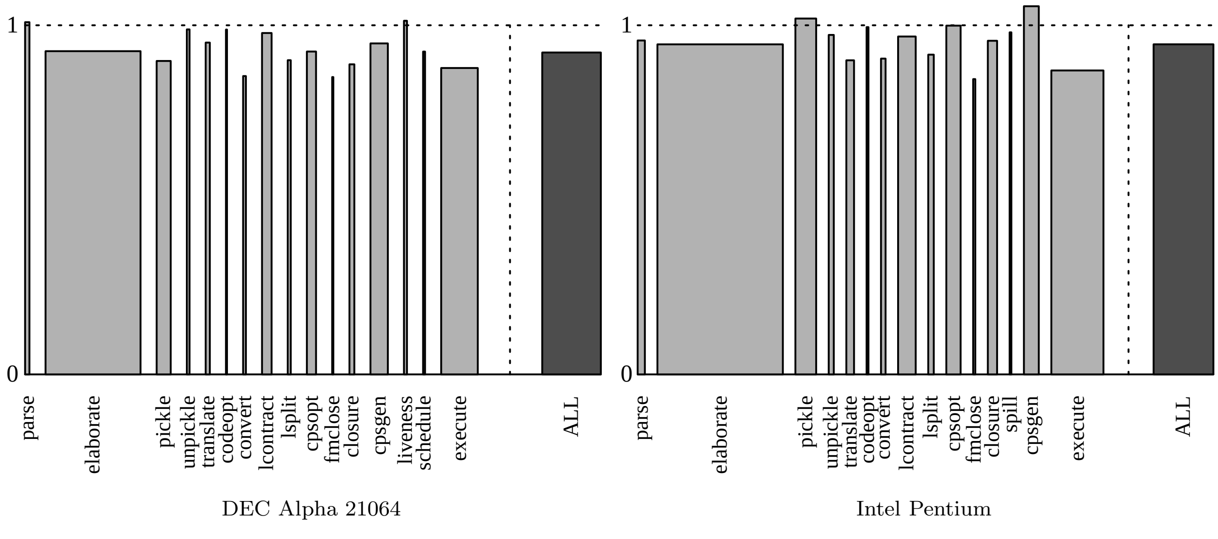


Figure 5: **Improvement due to cross-module inlining.** λ-splitting yields improvement on almost all benchmarks, and improves total execution time by 8% on the Alpha, 5% on the Pentium. The baseline (dotted line at 1.0) shows a compiler with λ-contract but without cross-module inlining; the grey bars show runtime (smaller is better) with both λ-contract and inlining.

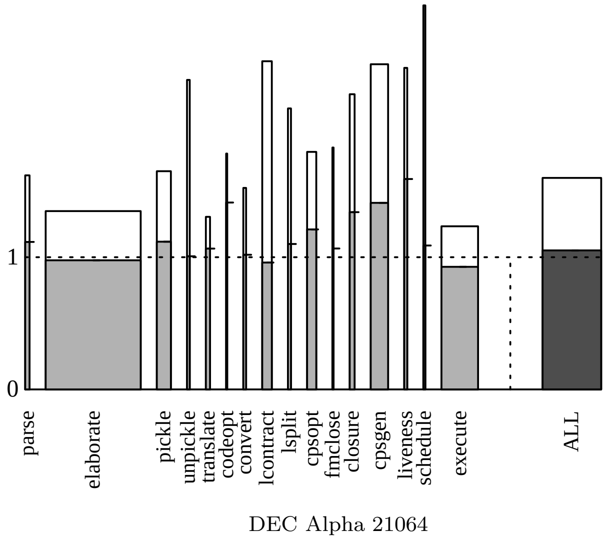


Figure 6: **Inlining applied to a cleanly modularized program.** We reinserted signature constraints and abstractions that had been removed by the efficiency-conscious software engineers. The white bars show how much this would have cost without cross-module inlining. The dark bars show that λ-splitting recovers essentially all of the performance cost of modular abstraction.

right-hand side was known to be in a pre-selected set of primitive operators, then this fact was recorded as part of the static information available for the variable bound on the left-hand side. In effect, inlining information was treated in a fashion similar to types.

Unfortunately, the mechanism only applied to a fixed set of pre-selected values, and it also was not robust with respect to simple source code modifications. For example, it was disabled by simply adding a few semantics-preserving signature constraints—something the unsuspecting user might do by accident.

Even so, figure 5 shows that λ-splitting plus ad-hoc inlining is better than ad-hoc inlining alone.

As an experiment we changed the compiler sources to eliminate old-style ad-hoc inlining. The results (Figure 6) 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.

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%. In figure 5 these phases are labeled `lcontract` and `lsplit`, respectively. In addition, the intramodule CPS-optimization phase slows down because it has more (useful!) work to do. Compile time increases by 7% overall, code size goes up by about 5%.

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 competitive with previous ad-hoc approaches that resorted to sacrificing abstraction and modularity.

8 Future work

We intend to find appropriate heuristics for $\underline{1}$-decomposition.

The closure-conversion algorithm, though already quite sophisticated [SA94], must be improved to undo harmful effects of λ-contract (β-reductions and let-floating).

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 always 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 represented using functions and function calls, we can solve this problem. Without inlining this would have been totally impractical, but we expect inlining will eliminate any penalty in almost all cases.

A The intermediate language

We use the following calculus as our intermediate representation. It is reminiscent of A-normal form, with the only difference being that we need not distinguish between ordinary function application and tail calls. Variable bindings are established by **fn**, **fix**, and **let**. Each variable is bound at most once; variables occuring in closed functions are bound exactly once. We distinguish between *pure* and *impure* primops; the former are guaranteed not to incur effects upon application. Even "read" effects such as accesses to reference variables are considered impure here.

var	$\rightarrow$ **v0** | **v1** | ...
const	$\rightarrow$ *int* | *real* | ...
primop	$\rightarrow$ **pure** *pure*
	$\rightarrow$ **impure** *impure*
val	$\rightarrow$ **var** *var*
	$\rightarrow$ **const** *const*
	$\rightarrow$ *primop*
exp	$\rightarrow$ **val** *val*
	$\rightarrow$ **fn** (*var*, *exp*)
	$\rightarrow$ **fix** ((*var*, *var*, *exp*) ..., *exp*)
	$\rightarrow$ **let** (*var*, *exp*, *exp*)
	$\rightarrow$ **app** (*val*, *val*)
	$\rightarrow$ **record** (*val* ...)
	$\rightarrow$ **select** (*int*, *val*)
	$\rightarrow$ **if** (*val*, *exp*, *exp*)

Expressions only consisting of a variable access, a constant, or a primitive operator use the **val** clause; **fn** introduces a λ-abstraction. Recursive function bindings are established using **fix**; an expression of the form

fix ((f1,v1,E1),(f2,v2,E2),...,B)

corresponds to the SML expression

```
let fun f1 v1 = E1
    fun f2 v2 = E2
in B end
```

A **let**-expression locally binds a variable; **app** denotes the application of one value to another. Records are constructed using **record** and selected from using **select**. The **if**-clause specifies a conditional expression.
$\mathcal{F}_V(\ldots)$ and $\mathcal{F}(\ldots)$ calculate the sets of free variables for *exp* and *val*, respectively:

$$\begin{aligned}
\mathcal{F}_V(\mathbf{var}\,v) &= \{v\}\\
\mathcal{F}_V(\mathbf{const}\,c) &= \emptyset\\
\mathcal{F}_V(\mathbf{pure}\,p) &= \emptyset\\
\mathcal{F}_V(\mathbf{impure}\,p) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}(\mathbf{val}\,x) &= \mathcal{F}_V(x)\\
\mathcal{F}(\mathbf{fn}\,(v,e)) &= \mathcal{F}(e)\setminus\{v\}\\
\mathcal{F}(\mathbf{fix}\,(l,b)) &= (\mathcal{F}_f(l)\cup\mathcal{F}(b))\setminus\mathcal{B}_f(l)\\
\mathcal{F}(\mathbf{let}\,(v,e,b)) &= \mathcal{F}(e)\cup(\mathcal{F}(b)\setminus\{v\})\\
\mathcal{F}(\mathbf{app}\,(x_1,x_2)) &= \mathcal{F}_V(x_1)\cup\mathcal{F}_V(x_2)\\
\mathcal{F}(\mathbf{record}\,(x_1,\ldots)) &= \bigcup_{i=1,\ldots}\mathcal{F}_V(x_i)\\
\mathcal{F}(\mathbf{select}\,(i,x)) &= \mathcal{F}_V(x)\\
\mathcal{F}(\mathbf{if}\,(x,t,e)) &= \mathcal{F}_V(x)\cup\mathcal{F}(t)\cup\mathcal{F}(e)
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_f((f_1,v_1,e_1),\ldots) &= \bigcup_{i=1,\ldots}\{\mathcal{F}(e_i)\setminus\{v_i\}\}\\
\mathcal{B}_f((f_1,v_1,e_1),\ldots) &= \{f_1,\ldots\}
\end{aligned}$$

B Expansiveness

Expressions can be moved into the inlinable portion of a split, if the predicate $\mathcal{P}(\ldots)$ ("permitted") holds for them:

$$\begin{aligned}
\mathcal{P}(\mathbf{val}\,v) &= T\\
\mathcal{P}(\mathbf{fn}\,(v,e)) &= T\\
\mathcal{P}(\mathbf{fix}\,(l,e)) &= \mathcal{P}(e)\\
\mathcal{P}(\mathbf{let}\,(v,e,b)) &= \mathcal{P}(e)\wedge\mathcal{P}(b)\\
\mathcal{P}(\mathbf{app}\,(\mathbf{pure}\,p,x)) &= T\\
\mathcal{P}(\mathbf{app}\,(\mathbf{impure}\,p,x) &= F\\
\mathcal{P}(\mathbf{app}\,(\mathbf{var}\,v,x)) &= \texttt{purefun}\,(v)\\
\mathcal{P}(\mathbf{record}\,l) &= T\\
\mathcal{P}(\mathbf{select}\,(i,x)) &= T\\
\mathcal{P}(\mathbf{if}\,(x,t,e)) &= F
\end{aligned}$$

The use of the auxiliary function `purefun` deserves some explanation: Normally all function applications are deemed expansive and, thus, are not allowed to occur within the inlinable part. However, we sometimes can be sure that certain functions have no side effects, in which case there is no harm in moving a corresponding application. In particular, this happens when our algorithm splits functions recursively, because it always maintains the invariant that the inlinable portion of a split is effect-free.

To illustrate this point consider a compilation unit that defines a functor and then also instantiates it:

```
fun F x = ...
val m = F a
```

S-decomposition on `F` transforms this into:

```
fun Fe x = ...
fun Fi (fe, x) = ...
fun F x = Fi (Fe x, x)
val m = F a
```

It is now possible to export the definitions for `Fi` and `F`, but we still cannot also export `m`'s definition, because its right-hand side is an application. This is unfortunate, because `m` is constructed by applying `Fe` as well as `Fi`, and `Fi` itself was constructed in such a way that its applications should become inlinable. In other compilation units that instantiate `F` this will not pose a problem, because λ-contract will turn such applications into explicit **let**-bindings. But within the same compilation unit λ-contract has already finished its work before `F` was ever split into `Fe` and `Fi`. We therefore explicitly keep track of such cases and β-reduce the call to `F` on the fly:

```
fun Fe x = ...
fun Fi (fe, x) = ...
fun F x = Fi (Fe x, x)
val tmp = Fe a
val m = Fi (tmp, a)
```

Since `Fi` refers to the inlinable portion of a split, we assert: `purefun` (`Fi`) $= T$, and `m`'s definition can also be exported and inlined into other compilation units.

Currently, purefun(...) returns *false* in all other cases. It would be possible to further relax this by checking the bodies of other functions as well to see whether or not they are expansive. We have not done this, because in many cases such function applications are already β-reduced during the λ-contract phase.

C Size estimates

We use a simple, syntax-driven size estimate $\mathcal{S}(\ldots)$ to limit the amount of code that will be exported symbolically to be inlined into other compilation units. This estimate is necessarily imprecise, because subsequent optimization can radically alter the code.

The values $A, B, C, D, E, F, G, H, I, K, L$ as well as the function $\mathcal{S}_{\mathrm{p}}(\ldots)$ are adjustable parameters to the algorithm and are chosen to approximately reflect the relative cost of implementing the respective language feature.

$$\begin{aligned}
\mathcal{S}_{\mathrm{V}}(\mathbf{var}\, v) &= A \\
\mathcal{S}_{\mathrm{V}}(\mathbf{const}\, c) &= B \\
\mathcal{S}_{\mathrm{V}}(\mathbf{pure}\, p) &= C \\
\mathcal{S}_{\mathrm{V}}(\mathbf{impure}\, p) &= C
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(\mathbf{val}\, x) &= \mathcal{S}_{\mathrm{V}}(x) \\
\mathcal{S}(\mathbf{fn}\,(v, e)) &= \mathcal{S}(e) + D|\mathcal{F}(e) \setminus \{v\}| + E \\
\mathcal{S}(\mathbf{fix}\,(l, b)) &= \mathcal{S}_{\mathrm{f}}(l) + D|\mathcal{F}_{\mathrm{f}}(l) \cup \mathcal{B}_{\mathrm{f}}(l)| + \\
& \mathcal{S}(b) + F \\
\mathcal{S}(\mathbf{let}\,(v, e, b)) &= \mathcal{S}(e) + \mathcal{S}(b) + G \\
\mathcal{S}(\mathbf{app}\,(\mathbf{pure}\, p, x)) &= \mathcal{S}_{\mathrm{p}}(p) + \mathcal{S}_{\mathrm{V}}(x) \\
\mathcal{S}(\mathbf{app}\,(\mathbf{impure}\, p, x)) &= \mathcal{S}_{\mathrm{p}}(p) + \mathcal{S}_{\mathrm{V}}(x) \\
\mathcal{S}(\mathbf{app}\,(x_1, x_2)) &= \mathcal{S}_{\mathrm{V}}(x_1) + \mathcal{S}_{\mathrm{V}}(x_2) + H \\
\mathcal{S}(\mathbf{record}\,(x_1, \ldots)) &= I + \sum_{i=1,\ldots} \mathcal{S}_{\mathrm{V}}(x_i) \\
\mathcal{S}(\mathbf{select}\,(i, x)) &= \mathcal{S}_{\mathrm{V}}(x) + K \\
\mathcal{S}(\mathbf{if}\,(x, t, e)) &= \mathcal{S}_{\mathrm{V}}(x) + \mathcal{S}(t) + \mathcal{S}(e) + L
\end{aligned}$$

$$\mathcal{S}_{\mathrm{f}}((f_1, v_1, e_1), \ldots) = \sum_{i=1,\ldots} \mathcal{S}(e_i)$$

For Standard ML of New Jersey, we use the following coefficients:

A	B	C	D	E	F	G	H	I	K	L
0	0	0	2	1	2	0	5	1	1	4

and $S_{\mathrm{p}}(p)$ is between 1 and 4, for most p.

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.

[BDGK94] Koen De Bosschere, Saumya Debray, David Gudeman, and Sampath Kannan. Call forwarding: a simple interprocedural optimization technique for dynamically typed languages. In *POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 409–420, 1994.

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

[Blu97] Matthias Blume. *Hierarchical Modularity and Intermodular Optimization*. PhD thesis, Princeton University, expected 1997.

[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.

[GA95] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Proc. Seventh Int'l Conf. on Functional Programming and Computer Architecture*, pages 293–305. ACM Press, 1995.

[Gou94] Jean Goubault. Generalized boxing, congruences and partial inlining. In *Static Analysis Symposium '94*, number 864 in Lecture Notes in Computer Science, pages 147–161. Springer-Verlag, 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[JPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *1996 SIGPLAN International Conference on Functional Programming*, pages 3–12, May 1996.

[JW96] Suresh Jagannathan and Andrew Wright. Flow-directed Inlining. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, May 1996.

[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.

[Oho92] Atsushi Ohori. A compilation method for ml-style polymorphic record calculi. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 154–165. ACM Press, Jan 1992.

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

[Rei94] Mark B. Reinhold. Cache performance of garbage-collected programs. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, pages 206–217. ACM Press, June 1994.

[SA94] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 150–161, New York, 1994. ACM Press.

[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.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[Tol94] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 1–11. ACM Press, 1994.