Practical Inlining of Functions with Free Variables

Lars Bergstrom
John Reppy
Nora Sandler
University of Chicago
{larsberg,jhr,nlsandler}@cs.uchicago.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract
A long-standing practical challenge in the optimization of higher-order languages is inlining functions with free variables. Inlining code statically at a function call site is safe if the compiler can guarantee that the free variables have the same bindings at the inlining point as they do at the point where the function is bound as a closure (code and free variables). There have been many attempts to create a heuristic to check this correctness condition, from Shivers’ kCFA-based reflow analysis to Might’s ΔCFA and anodization, but all of those have performance unsuitable for practical compiler implementations. In practice, modern language implementations rely on hand-inlining by the programmer for anything more complicated.

This work provides the first practical, general approach for inlining functions with free variables. We also provide a proof of correctness, an evaluation of both the execution time and performance impact of this optimization, and some tips and tricks for implementing an efficient and precise control-flow analysis.

1. Introduction
Inlining is a program transformation that replaces a function call with the code body from the target of the call. This transformation can directly improve performance by eliminating both function call overheads — such as stack management, argument passing, and the jump instruction — and runtime overheads, such as garbage collection checks. Inlining can also indirectly improve performance by optimizing the code body with respect to actual arguments. In higher-order languages, function calls can also be made indirectly through a closure, which is a runtime value that contains a code pointer and an environment containing the values of bound variables; inlining of first-class functions gains additional potential performance improvements through removal of the closure value.

Unfortunately, inlining of functions with free variables requires additional safety analysis. Optimizing compilers must ensure that the dynamic environment at the static location where the code can potentially be inlined is the same as the dynamic environment at all of the closure capture locations that flow to it, up to the free variables of the function to inline. This problem is critical, as inlining of functions with free variables comes up in idiomatic uses of most higher-order languages. Consider the definition of the map function in ML, annotated with a superscript label at an interesting call site:

```ml
val x = 3
fun g i = i + x
fun map f l = case l of i::ls => (f i)\[1\]::(map f ls) | _ => []
val res = map g [1,2,3]
```

Assuming that there are no other calls to the function map, we would like to inline the function g that increments its integer argument by three into the map function at the call site labeled 1. After inlining, constant propagation, and a round of useless variable elimination, we would obtain the following code:

```ml
fun l = case l of i::ls => (i+3)::(map ls) | _ => []
val res = map l [1,2,3]
```

While safe in this specific example, inlining of functions such as g is not safe in general because of its free variables, x and (less obviously) i. The potential lack of safety comes from these free variables which, when inlined, may have had a different binding at their original capture point than they have at the inlining point. Most modern functional compilers make this simple, idiomatic example work by ignoring any free variables that are bound at the top level when making inlining decisions.

The following example exhibits the importance of reasoning about environments when performing inlining on functions with free variables. The function ε takes a boolean and a function of type `unit -> bool`, returning a function of that same type. If the boolean argument to ε is true, it will return a new function that simply calls the function it was passed. If the argument was false, then that parameter is captured in a closure and returned.

```ml
fun f (b, k) = if b then (fn () => (k {}))\[1\] else (fn () => b)\[2\]
val arg = f (false, (fn () => false))
val res = f (true, arg)
val final = res ()
```

First, the function ε is called with false, in order to capture the variable b in a closure. Then, ε is called again, this time with

---

1 Inspired by one from Might’s Ph.D. dissertation [Mig07].
the result of the first call, to produce a new closure — in an environment with a different binding for \( s \) — that wraps the one from the first call. Finally, we call that closure.

At the call site labeled 1, control-flow analysis can determine that only the anonymous function labeled 2 will ever be called. Unfortunately, if we inline the body of the anonymous function at that location, as shown in the example code below, the result value \( \text{final} \) will change from \( \text{false} \) to \( \text{true} \). The problem is that the binding of the variable \( s \) is not the same at the potential inline location as it was at its original capture location.

\[
\begin{align*}
\text{fun} & \ f (b, k) = \\
\text{if} & \ b \ \text{then} \ (f n () \Rightarrow (f n () \Rightarrow b) (1)) ^1 \\
& \ \text{else} \ (f n () \Rightarrow b)^2 \\
\text{val} & \ \text{arg} = f (\text{false}, (f n () \Rightarrow \text{false})) \\
\text{val} & \ \text{res} = f (\text{true}, \text{arg}) \\
\text{val} & \ \text{final} = \text{res}()
\end{align*}
\]

While this example is obviously contrived, this problem occurs regularly and the inability to handle the environment problem in general severely limits most compilers.

This final example shows a slightly more complicated program that defeats simple heuristics but in which inlining is safe and the techniques presented in this work enable.

\[
\begin{align*}
\text{let} & \ y = n () \\
\text{fun} & \ f = y \\
\text{fun} & \ g h = (h ()) ^1 \\
\text{in} & \ g f \\
\text{end}
\end{align*}
\]

At the call site labeled 1, it is clearly safe to inline the body of the function \( \xi \), since \( \gamma \) has the same binding at the inline location as the capture location. Since it is not a trivial idiomatic example, however, it is not commonly handled.

This work shows how to use control-flow analysis (CFA) to handle many more inlining situations than are possible in either type-directed optimizers or simpler heuristics-based optimizers. By reformulating this \textit{environmental consonance} problem as a graph reachability problem instead of a partial flow analysis, we have reduced this analysis to a single linear-log time analysis, independent of the number of potential inlining sites. This reformulation makes higher-order inlining in the presence of free variables practical, results in better code performance, and is now on by default in the Manticore system. Our contributions are:

- A practical heuristic for inlining functions with free variables.
- Proof of correctness of that heuristic.
- Timing results for both the cost of whole-program higher-order inlining analysis (\(< 3\%\) of compilation time) and its impact on program performance (up to \(8\%\) speedup).
- An additional optimization, branch elimination, that is both effective and nearly cost-free when already performing control-flow analysis.

**Roadmap** First, in the next section we provide a brief introduction to control-flow analysis and present novel information about tuning the analysis with respect to recursive datatypes. Section 3 provides an overview of the Manticore compiler, with a focus on the continuation-passing style intermediate representation of the compiler, on which this work is based. The following sections discuss some of the basic optimizations performed using the results of control-flow analysis and introduce a straightforward optimization — branch elimination — that takes advantage of a small extension to the control-flow analysis. Section 4 discusses the analysis challenges presented by environments in some more detail, and is followed by a section that describes our novel, \textit{practical} solution. Finally, we provide empirical data and conclude.

Source code for our complete implementation and all the benchmarks described in this paper is available at: [http://smlnj-gforge.cs.uchicago.edu/projects/manticore/](http://smlnj-gforge.cs.uchicago.edu/projects/manticore/)

2. Practical Control-Flow Analysis

This section provides an overview of our specific implementation of control-flow analysis, placing it in context with other implementations, both theoretical and practical. For a more general introduction to control-flow analysis, in particular the OCF style that we use, the book by Nielson \textit{et al.} provides a comprehensive introduction to both static program analysis and OCFAs [NNH99]. While many others have implemented control-flow analysis in their compilers [Sr95, CJW00, AD98], our analysis is novel in its tracking of a wider range of values, including datatypes, and its lattice coarsening to balance performance and precision.

2.1 General algorithm

Using the terminology of Midtgaard’s comprehensive survey of control-flow analysis [Mid12], our implementation is a \textit{zeroth-order control-flow analysis} (OCFA). An OCFA computes a finite map from all of the variables in a program to a conservative abstraction of the values that they can take on during the execution of the code. Ours does this by starting from an empty map and iterating over the intermediate representation of the program, merging value flow information into the map based on the expressions until the map no longer changes. In our experience, the key to keeping performance acceptable while still maintaining high precision lies in carefully choosing (and empirically tuning) the tracked abstraction of values.

2.2 Tuning the lattice

Each time we evaluate an expression whose result is bound to a variable, we need to update the map with a new abstract value that is the result of merging the old abstract value and the new value given by the analysis. In theory, if all that we care about in the analysis is the mapping of call sites to function identifiers, we could use a straightforward domain for the value map \( \mathcal{V} \), based on the powerset of the function identifiers:

\[
\mathcal{V} : \text{VarID} \rightarrow 2^{\text{FunID}}
\]

Unfortunately, this domain is insufficiently precise because of the presence of tuples and datatypes as well as the default Standard ML calling convention, in which all arguments are passed as a single argument that is a tuple containing those arguments. Without tracking more complicated data structures, we will only be able to track values in trivial code without datatypes or multi-argument functions.

The representation we use for abstract values therefore is a recursive datatype, as shown in Figure 1. The special \( \top \) (\text{top}) and \( \bot \) (\text{bot}) elements indicate either all possible values or no known values, respectively. A \text{TUPLE} value handles both the cases of tuples and ML datatype representations, which by this point in the compiler have been desugared into either raw values or tagged tuples. The \text{LAMBDAS} value is used for a set of variable identifiers, all of which are guaranteed to be function identifiers.

A lattice over these abstract values uses the \( \top \) and \( \bot \) elements as usual, and treats values of \text{TUPLE} and \text{LAMBDAS} type as incomparable. When two \text{LAMBDAS} values are compared, though, the subset relationship provides an ordering. It is this ordering that allows us to incrementally merge flow information, up to a fixed limit. The most interesting portion of our implementation is in our handling of merging two \text{TUPLE} values. In the trivial recursive solution, the analysis will fail to terminate, due to the presence of recursive datatypes.
First-order representation suitable for code generation. CPS transformation is performed in the Danvy-Filinski style [DF92]. This representation is a good fit for a simple implementation of control-flow analysis because it transforms each function return into a call to another function. The uniformity of treating all control-flow as function invocations simplifies the implementation. We also have an implementation of control-flow analysis on the BOM direct-style representation, which was used for some Concurrent ML-specific optimizations [RX07]. The BOM-based implementation is almost 10% larger in lines of code, despite lacking the optional features, user-visible controls, and optimizations described in this paper.

The primary datatypes and their constructors are shown in Figure 1. Key features of this representation are:

- Each expression has a program point associated with it, which serves as a unique label.
- It has been normalized so that every expression is bound to a variable.
- The rhs datatype, not shown here, contains only immediate primitive operations.
- The CPS constraint is captured in the IR itself — Apply and Throw are non-recursive constructors, and there is no way to sequence an operation after them.

3. Manticore

In order to provide more details on both the implementation of control-flow analysis and the optimizations that rely on it, we provide some background on the host compiler and relevant intermediate representation. The compiler operates on the whole program at once, reading in the files in the source code alongside the sources from the runtime library. As covered in more detail in an earlier paper [FRR07], there are six distinct intermediate representations (IRs) in the Manticore compiler:

1. Parse tree — the product of the parser.
2. AST — an explicitly-typed abstract-syntax tree representation.
3. BOM — a direct-style normalized λ-calculus.
4. CPS — a continuation passing style λ-calculus.
5. CFG — a first-order control-flow-graph representation.
6. MLTree — the expression tree representation used by the ML-RISC code generation framework [CGGR94].

The work in this paper is performed on only the CPS representation.

3.1 CPS

Continuation passing style (CPS) is the final high-level representation used in the compiler before closure conversion generates a

datatype value
    = TOP
    | TUPLE of value list
    | LAMBDAS of CPS.Var.Set.set
    | BOT

Figure 1. Abstract values.

(e.g., on each iteration over a function that calls the cons function, we will wrap another TUPLE value around the previous value). In practice for typical Standard ML programs, we have found that limiting the tracked depth to 5 and then failing any further additions to the T value results in good balance of performance and precision.

Note that unlike some other analyses, such as sub-zero CFA, we do not limit the maximum number of tracked functions per variable [AD98]. If the only use of our CFA were inlining, then we could also limit our CFA to a single potential function; however, as we discuss further in Section 4, this implementation of CFA is used for several optimizations that can still be performed when multiple functions flow to the same call site. Further, we did not notice any change in the runtime of the analysis when reducing the number of tracked function variables, but it severely impacts our ability to perform some of those other optimizations.

2.3 Adding booleans for additional flow-sensitivity

Normally, CFA does not track booleans and other raw values, so they are represented by the T value. To track booleans, we add true and false to our list of abstract values, and update CFA to account for their place in the lattice. When both true and false values flow to the same variable, we promote its abstract value to T. The updated datatype is shown in Figure 2.

datatype value
    = TOP
    | TUPLE of value list
    | LAMBDAS of CPS.Var.Set.set
    | BOOL of bool
    | BOT

Figure 2. Abstract values with boolean tracking.

4. Basic Optimizations

In Manticore, we use the data from control-flow analysis (CFA) to augment some of our optimizations. In this section, we list some of those optimizations and describe the differences between purely type-directed optimizations and those that rely on the results of CFA.

4.1 Argument flattening

The default calling convention in Standard ML (and many other functional languages) involves heap-allocating all arguments to a function and passing a pointer to that heap-allocated data to the function. As has been described in the Haskell literature, we could analyze the type of the function and then, based on the calling convention of that type, place the arguments in appropriate registers and evaluate the function [MP04]. In many cases, when combined with inlining and other optimizations, we can then avoid allocating some arguments in the heap. By instead using CFA as the basis for our optimizations, we can not only specialize the calling convention but have several additional optimization opportunities:
1. If the target function or functions only use a few of the parameters, we can change their calling convention and type to only pass those parameters.
2. We adjust both callers and callees to flatten tuples in the case where the caller performs the allocation and the callee simply extracts members of that tuple.

This work has been described in detail in an earlier paper [BR09].

4.2 Calling conventions
There are two interesting optimizations related to CFA in the Manticore implementation of calling conventions. In the following example, even in a type-directed optimizer, inlining will cause \( g \) to be inlined and thus the call to \( \varepsilon \) will become known and eligible for a direct jump:

\[
\begin{align*}
\text{let} & \quad \text{fun } f \text{x = x + 1} \\
& \quad \text{fun } g \text{ (h, i) = h i} \\
\text{in} & \quad g \text{ (f, 3)} \\
\end{align*}
\]

But, if \( g \) is too large to inline and if there are any other functions of the same type as \( \varepsilon \), it will be unclear to the compiler what functions could be bound to the variable \( h \), forcing it to make an indirect call through a pointer to invoke \( \varepsilon \). In Manticore, we use CFA to recognize that \( \varepsilon \) is known and perform a direct jump to it within the body of \( g \).

Even in the case where \( \varepsilon \) has free variables, and thus we need to pass a closure, an additional optimization is still available. Typically, closures include both a code pointer and the associated environment data. In this case, though, we replace the closure with just the environment data and instead perform a direct jump. Both of these optimizations remove just a single pointer indirection and a few bytes of allocation, but the availability of these optimizations means that users of the system do not have to transform their code by hand to get good performance in key inner loops.

5. Branch elimination
In Section 2.3 we showed an extension of the lattice of abstract values computed by control-flow analysis that also tracks boolean values. In this section, we describe a compiler optimization that eliminates conditional branches that will never be taken based on values computed by control-flow analysis that also tracks boolean values. Consider the following function:

\[
\begin{align*}
\text{fun } f \text{ (boolean) =} \\
& \quad \text{if boolean} \\
& \quad \text{else } g \text{ 4}
\end{align*}
\]

If the value of boolean is known, we can eliminate the conditional statement and leave only the relevant branch.

Early in the compilation process of Manticore, between the AST and BOM phases described in Section 2, the if construct on a boolean is converted into a case over booleans with constant values. The example above is therefore translated into roughly the following code, in direct-style instead of continuation-passing style for consistency with the rest of the examples in this paper:

\[
\begin{align*}
\text{fun } f \text{ (boolean) =} \\
& \quad \text{case boolean} \\
& \quad \mid \text{0x0} => g \text{ 4} \\
& \quad \mid \text{0x1} => g \text{ 3}
\end{align*}
\]

Suppose this function occurs in a larger program that only calls \( \varepsilon \) with arguments whose values are true. In that case, the control-flow analysis will have an abstract value of BOOL(true) associated with the variable boolean. Then, the branch elimination pass can eliminate all but the arm of the case expression that corresponds to the true case. After that branch elimination and a useless variable elimination, we are left with the following program:

\[
\begin{align*}
\text{fun } f \text{ () = g 3}
\end{align*}
\]

Since this optimization relies on information already computed by the control-flow analysis and requires only a single pass over the intermediate representation, as shown in Section 8, it has nearly zero cost.

6. Environment problems
Some optimizations that are straightforward in a first-order language when combined with control-flow analysis are not safe in a higher-order language using only the results of a zeroth-order CFA as described in Section 2. In that description of the CFA, the abstraction of the environment is a single, global map for each variable to a single value from the lattice. This restriction means that the CFA results alone do not allow us to reason separately about bindings to the same variable that occur along different control-flow paths of the program.

This restriction impedes inlining. Inlining a function is a semantically safe operation when a call site is to a unique function and that function has no free variables. But, in a higher-order language, inlining of a function with free variables is only safe when those free variables are guaranteed to have the same bound value at the capture location and inlining location, a property which Shivers called environmental consonance [Shi91]. For example, in the following code, if CFA determines that the function \( g \) is the only one ever bound to \( \varepsilon \), then the body of \( g \) may be inlined at the call site labeled 1.

\[
\begin{align*}
\text{val x = 3} \\
\text{fun g i = i + x} \\
\text{fun map f l =} \\
& \quad \text{case l} \\
& \quad \mid l::ls => (f l) :: (map f ls) \\
& \quad \mid _ => []
\end{align*}
\]

\[
\begin{align*}
\text{val res = map g [1,2,3]}
\end{align*}
\]

While many compilers special-case this particular situation, in which all free variables of the function are bound at the top level, even small changes break these fragile optimizers, as shown in the following code:

\[
\begin{align*}
\text{fun wrapper x = let} \\
& \quad \text{fun g i = i + x} \\
& \quad \text{fun map f l =} \\
& \quad \quad \text{case l} \\
& \quad \quad \mid l::ls => (f l) :: (map f ls) \\
& \quad \quad \mid _ => []
\end{align*}
\]

\[
\begin{align*}
\text{in} \\
& \quad \text{map g [1,2,3]} \\
\text{end} \\
\text{val res1 = wrapper 1} \\
\text{val res2 = wrapper 2}
\end{align*}
\]

Performing the inlining operation is again safe, but the analysis required to guarantee that the value of \( x \) is always the same at both the body of the function wrapper and in the call location inside of map is beyond simple heuristics.

Copy propagation A similar operation that has the identical problem is copy propagation. In this operation, instead of inlining the body of the function \( (e.g., \text{because it is too large}) \), we are attempting to remove the creation of a closure by turning an indirect call through a variable into a direct call to a target function. In the following code, the function \( g \) is passed as an argument to \text{map} and called in its body.
fun map f l = 
    case l
       of l::ls => (f l)::(map f ls)
       | _ => []

val res = map g [1,2,3]

When g either has no free variables or we know that those free
variables will always have the same values at both the capture
and inlining location, we can substitute g, potentially removing a
closure and enabling the compiler to optimize the call into a direct
jump instead of an indirect jump through the function pointer stored
in the closure record.

Interactions

These optimizations are not only important because they
remove an indirect call. If the variable that previously held the
closure is no longer used, then the useless variable elimination pass
will remove it from all calls and as a parameter to the function, as
shown in the code below, reducing register pressure.

fun map f l = 
    case l
       of l::ls => (g l)::(map ls)
       | _ => []

val res = map g [1,2,3]

7. Reflow

A theoretical solution to this environment problem that enables a
suite of additional optimizations is reflow analysis [Sh91]. Traditional
reflow analysis requires re-running control-flow analysis from the potential
inlining point and seeing if the variable bindings for all relevant free variables
are uniquely bound with respect to that sub-flow. Unfortunately, this operation
is potentially quite expensive (up to the same complexity as the original CFA, at each
potential inlining site) and no compiler performs it in practice.

We use a novel analysis that builds upon the static control-flow
graph of the program. Our goal was to build up a data structure that
could perform — in less than linear time — the same test used in
reflow analysis. The optimizations from Section 5 are safe when the
free variables of the target function are guaranteed to be the
same at its closure creation point and at the target call site. In the
original work by Shivers, this question was answered by checking
whether a binding for a variable had changed between those two
locations via a re-execution of control-flow analysis. Our analysis
instead turns that question into one of graph reachability: in the
graph corresponding to the possible executions of this program,
is there any possible path between those two locations through a
rebinding of the free variables?

This graph is built in two steps. First, build a static control-flow
graph for each function, ignoring function calls, annotated with
variable bindings and rebindings. Then, augment those individual
function graphs with edges from the call sites to the potential
target functions, as determined by the control-flow analysis. Though
we discuss only our implementation of OCFA in this work, this
approach to reflow analysis also works with other control-flow analy-

The variable bindings and rebindings in a program written in the
continuation-passing style (CPS) representation defined in Figure 3
happen in two cases:

- At the definition of the variable, which is either a let-binding
  or as a parameter of a function.
- In the case when a free variable of a function was captured in a
closure and this captured value is restored for the execution of
that function.

We capture both of these conditions through labeled nodes in
the graph for each function. One node is labeled with all of the free
variables of the function, since those are the ones that will be
rebound when the function is called through a closure. A second
node is labeled with all of the parameters to the function, since they
will also be bound when the function is called. Finally, any let-
binding in the control-flow graph will be labeled with the variable
being bound.

Augmentation of the call sites is done using the results of the
control-flow analysis described in Section 2. In the intermediate
representation, all targets of call sites are variables. In the trivial
case, that variable is the name of a function identifier, and we can
simply add an edge from the call site to that function’s entry point.
Otherwise, that variable is of function type but can be bound to
many possible functions. In that case, the control-flow analysis will
provide one of three results:

- The value ⊥, indicating that the call site can never be reached
  in any program execution. No changes are made to the program
  graph in this case.
- The value T, indicating that any call site may be reached. In
  this case, we add an edge to a special node that represents any
call site, whose optimization is discussed in Section 7.2.
- A set of function identifiers. Here, we add one edge from the
call site per function, to that function’s entry point.

At this point, the graph is complete and enables us to reformu-
late the safety property. We can now simply ask: does there exist
a path between the closure capture location and the target call site
in the graph that passes through a rebinding location for the free
variables of the function that we want to inline?

7.1 Safe example

We first revisit the first complicated, but safe, example from the
introduction, annotated with additional labels for use in the graph:

let
    val y = 2
    fun f x = y
    fun g h = (h (g x))

in
    (g f)
end

In the first stage of building the graph, we create the static control-
flow graph for each function. That graph is shown in Figure 4.
Nodes in this graph represent the labeled expressions from the
source program, and edges are the static control-flow. We also
create a separate function for toplevel bindings and the program’s
control-flow from the entry point.

Then, we augment that graph with edges from each of the call
sites to the target functions, based on the results of CFA. That graph
appears in Figure 5.

Revisiting the question of whether it is safe to inline the body of
the function f at the call site labeled 1, we now turn it into a graph
question. Does there exist a path from the closure capture location
(node 5), through either of the rebinding sites for the free variable y
(nodes 2 and 4), that terminates at the potential inlining site (node 1)?
Since one does not exist in this graph, the inlining is safe to
perform.

7.2 Unsafe example

For a negative case, we revisit the unsafe example from the intro-
duction, with additional labeling added.
interest. In the direct-style code we have used in this example, the outgoing edge from each of them corresponds to the return point of the function. In the continuation-passing style IR used in the compiler for this analysis, those have been translated into calls to functions that correspond to the return point.

In this example, we were investigating whether it was safe to inline the anonymous function defined at label 2 at the call site labeled 1.

So, in the graph in Figure 7, does there exist a path from the closure capture location (2) to the potential inlining point (1) that passes through a binding location for \( b \) (4 or 6)? Since we can find such a path in the graph (e.g., 2 \( \rightarrow \) 7 \( \rightarrow \) 10 \( \rightarrow \) 4 \( \rightarrow \) 5 \( \rightarrow \) 9 \( \rightarrow \) 12 \( \rightarrow \) 1), this inlining is potentially (and actually) unsafe, so it is disallowed under the reflow condition tested in our system.

7.3 Computing graph reachability quickly

This question about the existence of paths between nodes in the graph is a reachability problem. There are off-the-shelf \( O(n^3) \) algorithms such as Warshall’s algorithm for computing graph reachability [War62], but those are far too slow for practical use. On even small graphs of thousands of nodes, they take seconds to run. Other optimized algorithms have runtimes on the order of \( O(n \log n) \), but this is the runtime for each query and we might perform up to \( O(n) \) queries for a particular program.

Therefore, we use an approach that collapses the graph quickly into a map we can use for logarithmic-time queries of the reacha-

Knowing that \( V(k) = \{ (\text{fn} () \Rightarrow \text{false}) \} \) instead of a larger set requires an analysis more precise than 0CFA, such as our RC-CFA [Ber09] or Might’s ΓCFA [MS06b].
Strongly-connected components We use Tarjan’s algorithm for computing the strongly-connected components [Tar72], as implemented in Standard ML of New Jersey by Matthias Blume. This produces a directed acyclic graph (DAG). There are two interesting types of components for this algorithm: those that correspond to only one program point and those that correspond to more than one program point. In the single point case, control-flow from that point cannot reach itself. When there are multiple points, control-flow can reach itself. This distinction is crucial when initializing the reachability map.

Reachability in a DAG Starting from the leaf components of the graph, we add those components and everything that they can reach to the reachability set associated with each of their parent components. As those leaves are handled and removed from the worklist, their parents become leaves and we handle them iteratively until there are no components remaining in the graph. A more detailed description is shown in Algorithm 1.

Algorithm 1: Compute DAG reachability for a graph DAG

```plaintext
for node ∈ DAG do
    Done(node) ← false
    if # points ∈ node = 1 then
        R(node) ← { }
    else
        R(node) ← { node }
    end if
end for
leaves ← all leaves in DAG
while leaves is not empty do
    leaf ← a leaf in leaves
    leaves ← leaves − leaf
    Done(leaf) ← true
    for p ∈ parents(leaf) do
        R(p) ← R(p) ∪ R(leaf) ∪ { leaf }
    end if
end for
end while
```

7.4 Handling imprecision
In practical implementation, we also need to handle a variety of sources of imprecision. C foreign function calls, the entry and exit point of the generated binary itself (i.e., the main function), and the limited lattice size all contribute to situations where a call site may be through a variable whose target is T, or unknown. The obvious way to handle this situation when creating the graph is to add an edge from any call site labeled T to every possible function entry point. Unfortunately, that frequently connects the entire graph, preventing the compiler from inlining any function at all. Instead, we take advantage of the fact that a call to any function is really only a call to any function where not all its callers are known. Since those functions are a relatively small set, the graph remains useful.

7.5 Safety
We prove safety of this algorithm by building on the conditions of correctness for inlining functions with free variables from Might and Shivers in their work on ∆CFA [MS06a]. In that work, they prove that two conditions are sufficient for semantics-preserving inlining of a function:

1. All closures invoked at the given call site are to the same function.
2. The environment at the call site is equivalent up to the free variables of that function to the environment within any captured closure that reaches it.

To show the correctness of our algorithm, we therefore need to show that there is no case where our algorithm will attempt to perform an inlining operation, but the two ∆CFA inlining conditions do not hold. For contradiction, assume that there exists such an invalid inlining operation from our algorithm. Then, there are two cases to handle:

7.5.1 Same function
By construction, our OCFA-based algorithm has more conservative (i.e., coarser) results than the idealized ∆CFA algorithm. Therefore, whenever we identify a call site as only invoking closures of a given function, so must ∆CFA.
7.5.2 Environment equivalence

For the captured free variables in one of the closures to be different between its capture point and inlining location, there must have been a change in those variable bindings: if there is no path from the closure capture location to one of these binding locations and then on to the inlining point, reverting to an earlier binding happens when a different binding to a variable has been captured in a closure which is then restored through the application of that closure. However, the control-flow graph for each function is augmented with a program point annotated with any free variables used in the function. These program points are also treated as binding locations, so we allow inlining only if no path from capture location to inlining location passes through such a binding location.

Note that we are in a CPS-based representation and function calls never return. In a direct-style intermediate representation, the control-flow graph would similarly need re-binding locations for all of the variables that return into scope at each return point.

7.6 Limitations

While safe, this analysis necessarily is more limited than general formulations of higher-order inlining as shown by Shivers’ kCFA framework (for larger values of k than 0) or Moggi’s \( \Delta \)CFA approach [Shi91][MS06a]. Both of those analyses are able to distinguish environments created by different control-flow paths through the program. Our analysis collapses all different control-flow paths to each function, resulting in a potential loss of precision. For example, in the following program, which is an extension of the motivating example from the introduction, our attempt to inline at the call site labeled 1 will fail. After the first call to \( \text{callsG} \), the function \( \text{confounding} \) is in the abstract possible set of functions that can be bound to the parameter \( k \). Even though in the first call the boolean tracking avoids analyzing \( g \) and adding \( \text{confounding} \) to the list of possible values for \( n \), when the second call comes through, the function \( \tau \) is added to the possible set of values for \( k \) and then \( \text{both} \) of those are added to the set of values that could be bound to \( n \). Fundamentally, this problem is the one that stronger forms of control-flow analysis handle, though clearly there are some heuristics that could be used to increase the precision on this specific case.

```plaintext
let val y = 2
fun f _ = y
fun confounding _ = raise Fail "*
fun g h = h |
fun callsG b k = if b then g k else 0
val bad = callsG false confounding
in
  callsG true f
end
```

8. Evaluation

8.1 Experimental method

Our benchmark machine has two 8 core Intel Xeon E5-2687 processors running at 3.10 GHz. It has 64 GB of physical memory. This machine runs x86_64 Ubuntu Linux 11.10, kernel version 3.0.0-30.

We ran each benchmark experiment 30 times, and we report the median runtime and standard deviation in our tables. Times are reported in seconds.

This work has been implemented, tested, and is part of the current Manticore compiler’s default optimization suite. In addition to inlining, we also perform copy propagation, which replaces a variable that calls a function through a closure with the name of the function itself and requires an identical safety condition.

8.2 Benchmarks

For our empirical evaluation, we use six benchmark programs from our parallel benchmark suite. Each benchmark is written in a pure, functional style.

The Barnes-Hut benchmark [BH86] is a classic N-body problem solver. Each iteration has two phases. In the first phase, a quadtree is constructed from a sequence of mass points. The second phase then uses this tree to accelerate the computation of the gravitational force on the bodies in the system. Our benchmark runs 20 iterations over 400,000 particles generated in a random Plummer distribution. Our version is a translation of a Haskell program [GHC].

The Raytracer benchmark renders a 2048 × 2048 image as two-dimensional sequence, which is then written to a file. The original program was written in IDL [NH91] and is a simple ray tracer that does not use any acceleration data structures.

The Mandelbrot benchmark computes the Mandelbrot set, writing its output to an image file of size 2048 × 2048.

The Quickhull benchmark determines the convex hull of 8,000,000 points in the plane. Our code is based on the algorithm by Barber et al. [BDH96].

The Quicksort benchmark sorts a sequence of 5,000,000 integers in parallel. This code is based on the NESL version of the algorithm [Scal].

The SMVM benchmark is a sparse-matrix by dense-vector multiplication. The matrix contains 3,005,788 elements, the vector contains 10,000, and the multiplication is iterated 25 times.

8.3 Compilation performance

In Table 1 we have broken down the compilation time of these parallel benchmarks. While we have included the number of lines of code of the benchmarks, Manticore is a whole-program compiler, including the entire basis library. Therefore, in addition to the lines of code, we have also reported the number of expressions, where an expression is an individual term from the intermediate representation shown in Figure 3. By that stage in the compilation process, all un-referenced and dead code has been removed from the program.

The most important results are:

- Control-flow analysis and branch elimination are basically free.
- The reflow analysis presented in this work (which represents the majority of the time spent in both the copy propagation and inlining passes) generally makes up 1-2% of the overall compilation time.
- Time spent in the C compiler, GCC, generating final object code is the longest single stage in our compiler.

8.4 Benchmark performance

Across our already tuned benchmark suite, we see several improvements and only one statistically significant slowdown, as shown in Table 2. The largest challenge with analyzing the results of this work is that for any tuned benchmark suite, the implementers will have already analyzed and removed most opportunities for improvement. When we investigated the usefulness of these optimizations on some programs we ported from a very highly tuned benchmark suite, the Computer Language Benchmark Game [CLB13], we could find zero opportunities for further optimization.

All three of the major optimizations described in this paper — branch elimination, copy propagation, and higher-order inlining —
are applied to the optimized version, and none of them are present in the baseline. We run copy propagation before inlining in the compiler because that enables us to re-use the control-flow analysis information. If we performed inlining first, we would have to run CFA an additional time or manually update that information during the transformations before we could perform copy propagation.

Mandelbrot especially benefits from these optimizations, with a 12% improvement on one processor and 7.8% improvement in parallel. In this program, there is only one possible function that can be performed on any parallel process — the one that renders a given pixel in the output image. Since control-flow analysis is able to determine that, the runtime’s scheduler libraries themselves become specialized via copy propagation to jump directly to that function, rather than relying on an indirect jump through the work-stealing structures. This transformation also frees up many other variables that were kept by the scheduler to track the work in progress. While this program is certainly a special case, it is a good example of the usefulness of these optimizations in a whole-program compiler, as it allows specializations that are not available to the application author.

The only statistically significant negative performance impact is on the one-processor version of Quicksort. This benchmark, unfortunately, shows one of the risks of these optimizations. Copy propagation and inlining can potentially extend the live range of variables. In the one processor version of this benchmark, the live range of one variable that would otherwise have been copied into a closure once and forgotten is extended into another function that did not previously reference it. This extension not only increases the size of that function’s closure, but also requires the value be captured many times. In cases with more processors, the other optimizations balance out this one bad case, but it does demonstrate one of the risks of inlining or performing copy propagation on functions with free variables.

### 9. Related Work

The problem of detecting when two environments are the same with respect to some variables is not new. It was first given the name environment consonance in Shivers’ Ph.D. thesis [Shi91]. He proposed checking this property by re-running control-flow analysis (CFA) incrementally — at cost polynomial in the program size — at each inlining point. Might revisited the problem in the context of his Ph.D. thesis, and showed another form of analysis, ACFA, which more explicitly tracks environment representations and can check for safety without re-running the analysis at each inlining point [MS05a]. Unfortunately, this approach also only works in theory, as while its runtime is faster in practice than a full CFA (which is exponential), it is not scalable to large program intermediate representations. Might also worked on anodization, which is a more recent technique that identifies when a binding will only take on a single value, opening up the possibility of several optimizations similar to this one [Mig10].

Reps, Horowitz, and Sagiv were among the first to apply graph reachability to program analysis [RHS95], focusing on dataflow and spawning an entire field of program analyses for a variety of problems, such as pointer analysis and security. While they also present an algorithm for faster graph reachability, theirs is still polynomial time, which is far too slow for the number of nodes in our graphs. A different algorithm for graph reachability that has even better asymptotic performance than the one we present in Section 7.2 is also available [Nau94], computing reachability at the same time that it computes the strongly-connected components. However, it relies on fast language implementation support for mutation, which is not the case in our compiler’s host implementation system, Standard ML of New Jersey [AM91], so we use an algorithm that better supports the use of functional data structures.

Serrano’s use of 0CFA in the Bigloo compiler is the most similar to our work here [Ser95]. It is not discussed in this paper, but we similarly use the results of CFA to optimize our closure generation. In that paper, he does not discuss the need to track function identifiers within data types (e.g., lists in Scheme) or limit the depth of that tracking, both of which we have found crucial in ML programs where functions often are at least in tuples, due to the default calling convention. Bigloo does not perform inlining of functions with free variables.

Waddell and Dybvig use a significantly more interesting inlining heuristic in Chez Scheme, taking into account the potential impact of other optimizations to reduce the size of the resulting code, rather than just using a fixed threshold, as we do [WD97]. While they also will inline functions with free variables, they will only do so when either those variables can be eliminated or they know the binding at analysis time. Our approach differs from theirs in that we do not need to know the binding at analysis time and we support whole-program analysis, including all referenced library functions.

### 10. Conclusion

In this work, we have demonstrated the first practical and general approach to inlining and copy propagation that reasons about the environment. We hope that this work ushers in new interest and experimentation in environment-aware optimizations for higher-order languages. Additionally, we have presented tuning techniques for a control-flow analysis that tracks datatypes and an optimization — branch elimination — that is both useful and nearly free in a compiler that already performs control-flow analysis.

#### 10.1 Future work

We have not investigated other optimizations, such as rematerialization, that were presented in some of Might’s recent work on anodization [Mig10] and might have an analog in our framework. An obvious extension to branch elimination is case statement specialization based on the flow of datatypes through the program. In this
optimization, we could take an ML function that pattern matches across a datatype’s constructors and remove the arms of the pattern match that we can statically guarantee will never flow to that function.

Additionally, our control-flow analysis needs further optimizations, both to improve its runtime and its precision. We have previously investigated Hudak’s work on abstract reference counting, which resulted in improvements in both runtime and precision but that implementation is not yet mature [Hud86].

Acknowledgments

David MacQueen, Matt Might, and David Van Horn all spent many hours discussing this problem with us, and without their valuable insights this work would likely have languished.

This material is based upon work supported by the National Science Foundation under Grants CCF-0811389 and CCF-1010568, and upon work performed in part while John Reppy was serving at the National Science Foundation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[AD98] Ashley, J. M. and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. ACM TOPLAS, 20(4), July 1998, pp. 845–868.

[AAM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In PLIP ’91, vol. 528 of LNCS, Springer-Verlag, New York, NY, August 1991, pp. 1–26.

[BDH96] Barber, C. B., D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. ACM TOMS, 22(4), 1996, pp. 469–483.

[Ber09] Bergstrom, L. Arity raising and control-flow analysis in Manticore. Master’s dissertation, University of Chicago, November 2009. Available from http://manticore.cs.uchicago.edu.

[BH86] Barnes, J. and P. Hut. A hierarchical O(N log N) force calculation algorithm. Nature, 324, December 1986, pp. 446–449.

[BR09] Bergstrom, L. and J. Reppy. Arity raising in Manticore. In IFL ’09, vol. 6641 of LNCS, Springer-Verlag, September 2009, pp. 100–106.

[CJW00] Cejtin, H., S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In ESOP ’00, Springer-Verlag, 2000, pp. 56–71.

[CLB13] CLBG. The computer language benchmarks game, 2013. Available from http://benchmarksgame.alioth.debian.org/

Table 2. Performance results from branch elimination, copy propagation, and higher-order inlining optimizations. Medians and standard deviation are for the optimized version, reported in seconds.

| Benchmark    | 1 Processor | 16 Processors | Optimations |
|--------------|-------------|---------------|-------------|
|              | Speedup | Median | Std. Dev. | Speedup | Median | Std. Dev. | Branch Elim. | Copy Prop. | Inlined |
| Barnes-hut   | 0% | 17.2 | 0.03 | -1.6% | 2.1 | 0.048 | 3 | 14 | 0 |
| Mandelbrot   | 12% | 7.97 | 0.019 | 7.8% | 0.646 | 0.0445 | 1 | 3 | 0 |
| Quickhull    | 0.8% | 2.86 | 0.003 | 1.3% | 0.232 | 0.012 | 3 | 14 | 0 |
| Quicksort    | -3.3% | 14.05 | 0.02 | 0.2% | 1.18 | 0.025 | 1 | 5 | 0 |
| Raytracer    | 0.1% | 15.26 | 0.01 | 4.4% | 1.160 | 0.074 | 1 | 3 | 0 |
| SMVM         | 0.3% | 6.175 | 0.005 | -0.6% | 0.669 | 0.019 | 2 | 8 | 5 |

[DF92] Davny, O. and A. Filinski. Representing control: A study of the CPS transformation. MSCS, 2(4), 1992, pp. 361–391.

[FFR07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In ML ’07, ACM, October 2007, pp. 15–24.

[GGR94] George, L., F. Guillaume, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In CC ’94, April 1994, pp. 83–97.

[GHC] GHC. Barnes Hut benchmark written in Haskell. Available from http://darcs.haskell.org/packages/ndp/examples/barnesHut/

[Hud86] Hudak, P. A semantic model of reference counting and its abstraction (detailed summary). In LFP ’86, Cambridge, Massachusetts, USA, 1986. ACM, pp. 351–363.

[Mid12] Midtgaard, J. Control-flow analysis of functional programs. ACM Comp. Surveys, 44(3), June 2012, pp. 10:1–10:33.

[Mig07] Might, M. Environment Analysis of Higher-Order Languages. Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2007.

[Mig10] Might, M. Shape analysis in the absence of pointers and structure. In VMCAI ’10, Madrid, Spain, 2010. Springer-Verlag, pp. 263–278.

[MP04] Marlow, S. and S. Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. In ICFP ’04, ACM, September 2004.

[MS06a] Might, M. and O. Shivers. Environment analysis via ∆CFA. In POPL ’06, Charleston, South Carolina, USA, 2006. ACM, pp. 127–140.

[MS06b] Might, M. and O. Shivers. Improving flow analyses via TCF: abstract garbage collection and counting. In ICFP ’06, Portland, Oregon, USA, 2006. ACM, pp. 13–25.

[Nik91] Nikhil, R. S. ID Language Reference Manual. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[NNH99] Nielson, F., H. R. Nielson, and C. Hankin. Principles of Program Analysis. Springer-Verlag, New York, NY, 1999.

[Nuv94] Nuvitu, E. An efficient transitive closure algorithm for cyclic digraphs. IPL, 52, 1994.

[RHS95] Reps, T., S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In POPL ’95, San Francisco, 1995. ACM.

[RX07] Reppy, J. and Y. Xiao. Specialization of CML message-passing primitives. In POPL ’07, ACM, January 2007, pp. 315–326.

[Sca] Scandal Project. A library of parallel algorithms written NESL. Available from http://www.cs.cmu.edu/~scandal/nesl algorithms.html

[Ser95] Serrano, M. Control flow analysis: a functional languages compilation paradigm. In SAC ’95, Nashville, Tennessee, United States, 1995. ACM, pp. 118–122.

[Shi91] Shivers, O. Control-flow analysis of higher-order languages. Ph.D. dissertation, School of C.S., CMU, Pittsburgh, PA, May 1991.
[Tar72] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM JC*, 1(2), 1972, pp. 146–160.

[War62] Warshall, S. A theorem on boolean matrices. *JACM*, 9(1), January 1962.

[WD97] Waddell, O. and R. K. Dybvig. Fast and effective procedure inlining. In *SAS ’97*, LNCS. Springer-Verlag, 1997, pp. 35–52.