Practical and Effective Higher-Order Optimizations

Lars Bergstrom
Mozilla Research [*]
larsberg@mozilla.com

Matthew Fluet
Matthew Le
Rochester Institute of Technology
{mtf,ml9951}@cs.rit.edu

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

Abstract

Inlining is an optimization that replaces a call to a function with that function's body. This optimization not only reduces the overhead of a function call, but can expose additional optimization opportunities to the compiler, such as removing redundant operations or unused conditional branches. Another optimization, copy propagation, replaces a redundant copy of a still-live variable with the original. Copy propagation can reduce the total number of live variables, reducing register pressure and memory usage, and possibly eliminating redundant memory-to-memory copies. In practice, both of these optimizations are implemented in nearly every modern compiler.

These two optimizations are practical to implement and effective in first-order languages, but in languages with lexically-scoped first-class functions (aka, closures), these optimizations are not available to code programmed in a higher-order style. With higher-order functions, the analysis challenge has been that the environment at the call site must be the same as at the closure capture location, up to the free variables, or the meaning of the program may change. Olin Shivers' 1991 dissertation called this family of optimizations *Super-β* and he proposed one analysis technique, called *reflow*, to support these optimizations. Unfortunately, reflow has proven too expensive to implement in practice. Because these higher-order optimizations are not available in functional-language compilers, programmers studiously avoid uses of higher-order values that cannot be optimized (particularly in compiler benchmarks).

This paper provides the first practical and effective technique for Super-β (higher-order) inlining and copy propagation, which we call unchanged variable analysis. We show that this technique is practical by implementing it in the context of a real compiler for an ML-family language and showing that the required analyses have costs below 3% of the total compilation time. This technique's effectiveness is shown through a set of benchmarks and example programs, where this analysis exposes additional potential optimization sites.

[*] Portions of this work were performed while the author was at the University of Chicago.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-2873-9/14/09...$15.00.
http://dx.doi.org/10.1145/2628136.2628153

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Optimization

Keywords control-flow analysis, inlining, optimization

1. Introduction

All high level programming languages rely on compiler optimizations to transform a language that is convenient for software developers into one that runs efficiently on target hardware. Two such common compiler optimizations are copy propagation and function inlining. Copy propagation in a language like ML is a simple substitution. Given a program of the form:

```
let val x = y
in
    x*2+y
end
```

We want to propagate the definition of `x` to its uses, resulting in

```
let val x = y
in
    y*2+y
end
```

At this point, we can eliminate the now unused `x`, resulting in

```
y*2+y
```

This optimization can reduce the resource requirements (*i.e.*, register pressure) of a program, and it may open the possibility for further simplifications in later optimization phases.

Inlining replaces a lexically inferior application of a function with the body of that function by performing straightforward β-substitution. For example, given the program

```
let fun f x = 2*x
in
    f 3
end
```

inlining `f` and removing the unused definition results in

```
2*3
```

This optimization removes the cost of the function call and opens the possibility of further optimizations, such as constant folding. Inlining does require some care, however, since it can increase the program size, which can negatively affect the instruction cache performance, negating the benefits of eliminating call overhead. The importance of inlining for functional languages and techniques for providing predictable performance are well-covered in the context of GHC by Peyton Jones and Marlow [PM02].

Both copy propagation and function inlining have been well-studied and are widely implemented in modern compilers for both first-order and higher-order programming languages. In this paper, we are interested in the *higher-order* version of these optimizations, which are not used in practice because of the cost of the supporting analysis.

For example, consider the following iterative program:

```
let
  fun emit x = print (Int.toString x)
  fun fact i m k =
    if i=0 then k m
    else fact (i-1) (m*i) k
in
  fact 6 1 emit
end
```

Higher-order copy propagation would allow the compiler to propagate the function `emit` into the body of `fact`, resulting in the following program:

```
let
  fun emit x = print (Int.toString x)
  fun fact i m k =
    if i=0 then emit m
    else fact (i-1) (m*i) emit
in
  fact 6 1 emit
end
```

This transformation has replaced an indirect call to emit (via the variable `k`) with a direct call. Direct calls are typically faster than indirect calls,[1] and are amenable to function inlining. Furthermore, the parameter `k` can be eliminated using *useless variable elimination* [Shi91], which results in a smaller program that uses fewer resources:

```
let
  fun emit x = print (Int.toString x)
  fun fact i m =
    if i=0 then emit m
    else fact (i-1) (m*i)
in
  fact 6 1
end
```

Function inlining also has a similar higher-order counterpart (what Shivers calls Super-β). For example, in the following program, we can inline the body of `pr` at the call site inside `fact`, despite the fact that `pr` is not in scope in `fact`:

```
let
  val two = 2
  fun fact i m k =
    if i=0
    then k m
    else fact (i-1) (m*i) k
  fun pr x = print (Int.toString (x*two))
in
  fact 6 1 pr
end
```

Inlining `pr` produces

```
let
  val two = 2
  fun fact i m k =
    if i=0
    then print (Int.toString (m*two))
    else fact (i-1) (m*i) k
  fun pr x = print (Int.toString (x*two))
in
  fact 6 1 pr
end
```

This resulting program is now eligible for constant propagation and useless variable elimination.

[1] Direct calls to known functions can use specialized calling conventions that are more efficient and provide more predictability to hardware instruction-prefetch mechanisms.

While some compilers can reproduce similar results on trivial examples such as these, implementing either of these optimizations in their full generality requires an environment-aware analysis to prove their safety. In the case of copy propagation, we need to ensure that the variable being substituted has the same value at the point that it is being substituted as it did when it was passed in. Similarly, if we want to substitute the body of a function at its call site, we need to ensure that all of the free variables have the same values at the call site as they did at the point where the function was defined. Today, developers using higher-order languages often avoid writing programs that have non-trivial environment usage within code that is run in a loop unless they have special knowledge of either the compiler or additional annotations on library functions (*e.g.*, `map`) that will enable extra optimization.

This paper presents a new approach to control-flow analysis (CFA) that supports more optimization opportunities for higher-order programs than are possible in either type-directed optimizers, heuristics-based approaches, or by using library-method annotations. We use the example of *transducers* [SM06], a higher-order and compositional programming style, to further motivate these optimizations and to explain our novel analysis.

Our contributions are:

- A novel, practical environment analysis (Section 5) that provides a conservative approximation of when two occurrences of a variable will have the same binding.
- Timing results (Section 9) for the implementation of this analysis and related optimizations, showing that it requires less than 3% of overall compilation time.
- Performance results for several benchmarks, showing that even highly tuned programs still contain higher-order optimization opportunities.

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/`.

2. Manticore

The techniques described in this paper have been developed as part of the Manticore project and are implemented in the compiler for *Parallel ML*, which is a parallel dialect of Standard ML [FRRS11]. In this section, we give an overview of the host compiler and intermediate representation upon which we perform our analysis and optimizations. The compiler is a whole-program compiler, reading in the files in the source code alongside the sources from the runtime library. As covered in more detail in an earlier paper [FFR$^+$07], 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 [GGR94].

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

2.1 CPS

Continuation-passing style (CPS) is the final high-level representation used in the compiler before closure conversion generates a first-order representation suitable for code generation. Our 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. As a point of contrast, we have also implemented control-flow analysis on the BOM direct-style representation to support optimization of message passing [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. such as arithmetic and allocation of heap objects.
- 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.

```
datatype exp = Exp of (ProgPt.ppt * term)
and term
  = Let of (var list * rhs * exp)
  | Fun of (lambda list * exp)
  | Cont of (lambda * exp)
  | If of (cond * exp * exp)
  | Switch of (var * (tag * exp) list * exp option)
  | Apply of (var * var list * var list)
  | Throw of (var * var list)
and lambda = FB of {
  f : var,
  params : var list,
  rets : var list,
  body : exp
}
and ...
```

Figure 1. Manticore CPS intermediate representation.

3. Control-Flow Analysis

This section provides a brief background on control-flow analysis (CFA) along with an overview of our specific implementation techniques to achieve both acceptable scalability and precision. A more general introduction to control-flow analysis, in particular the 0CFA style that we use, is available in the book by Nielson *et al.* [NNH99]. For a detailed comparison of modern approaches to control-flow analysis, see Midtgaard's comprehensive survey [Mid12].

In brief, while many others have implemented control-flow analysis in their compilers [Ser95, CJW00, AD98], our analysis is novel in its tracking of a wider range of values — including boolean values and tuples — and its lattice coarsening to balance performance and precision.

3.1 Overview

A control-flow analysis 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. That is, it computes a finite map

$$\mathcal{V} : \texttt{VarID} \overset{\text{fin}}{\to} \texttt{value}$$

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

Figure 2. Abstract values.

where the `value` type is defined as a recursive datatype similar to that shown in Figure 2. The special $\top$ (`TOP`) and $\bot$ (`BOT`) elements indicate either all possible values or no known values, respectively. A `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 `LAMBDAS` value is used for a set of variable identifiers, all of which are guaranteed to be function identifiers. The `BOOL` value tracks the flow of literal boolean values through the program.

As an example, consider the following code:

```
let fun double (x) = x+x
    and apply (f, n) = f(n)
in
    apply (double, 2)
end
```

After running CFA on this example, we have

$$\begin{array}{rcl} \mathcal{V}(\texttt{f}) & = & \texttt{LAMBDAS}(\{\textit{double}\}) \\ \mathcal{V}(\texttt{n}) & = & \top \\ \mathcal{V}(\texttt{x}) & = & \top \end{array}$$

These results indicate that the variable `f` must be bound to the function `double`. This example and the `value` representation in Figure 2 do not track numeric values, which is why `n` and `x` are mapped to $\top$. We are planning to track a richer set of values, including datatype-specific values, in the future in order to enable optimizations beyond the ones discussed in this paper.

3.2 Implementation

Our CFA implementation is straightforward and similar in spirit to Serrano's [Ser95]. We start with an empty map and walk over the intermediate representation of the program. At each expression, we update $\mathcal{V}$ by merging the value-flow information until we reach a fixed point where the map no longer changes. The most interesting difference from Serrano's implementation is that we use our tracked boolean values to avoid merging control-flow information along arms of conditional expressions that can never be taken. In our experience, the key to reducing the runtime of control-flow analysis while still maintaining high precision lies in carefully choosing (and empirically tuning) the tracked abstraction of values.

3.2.1 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 simple domain for the value map ($\mathcal{V}$) based on just the powerset of the function identifiers. Unfortunately, this domain is insufficiently precise in practice because of the presence of tuples and datatypes. Furthermore, SML treats all functions has having a single parameter, which means that function arguments are packed into tuples at call sites and then extracted in the function body. Thus, the domain of abstract values needs to support tracking of information as it moves into and out of more complicated data structures.

We build a lattice over these abstract values using the $\top$ and $\bot$ elements as usual, and treating values of TUPLE and LAMBDAS type as incomparable. When two LAMBDAS values are compared, the subset relationship provides an ordering. It is this ordering that allows us to incrementally merge flow information, up to a finite limit. The most interesting portion of our implementation is in the merging of two TUPLE values. In the trivial recursive solution, the analysis may fail to terminate because of the presence of recursive datatypes (*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 mapping any further additions to $\top$ results in a good balance of performance and precision.

Note that unlike some other analyses, such as sub-zero CFA [AD98], we do not limit the maximum number of tracked functions per variable. Avoiding this restriction allows us to use the results of our analysis to support optimizations that can still be performed when multiple functions flow to the same call site (unlike inlining). Furthermore, we have found that reducing the number of tracked function variables has no measurable impact on the runtime of the analysis, but it removes many optimization opportunities (*e.g.*, calling convention optimization across a set of common functions).

4. The Environment Problem

All but the most trivial optimizations require program analysis to determine when they are safe to apply. Many optimizations that only require basic data-flow analysis when applied to first-order languages are not safe for higher-order languages when based on a typical CFA such as that described in Section 3. In that version of CFA, the abstraction of the environment is a single, global map that maps each variable to a single abstract 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.

For example, this restriction impedes higher-order inlining. First-order inlining of functions (simple β-reduction) is always a semantically safe operation. But, in a higher-order language, inlining a call through a closure that encapsulates a function and its environment is only safe when the free variables are guaranteed to have the same bound value at the capture location and the inlining location, a property that 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 the parameter f, then the body of g may be inlined at the call site labeled 1.

```
val x = 3
fun g i = i + x
fun map f l =
  case l
   of h::t => (f h)^1::(map f t)
    | _ => []

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

While some compilers handle this particular special-case, in which all the free variables of the function are bound at the top level, the resulting optimizers are fragile and even small changes to the program can hinder optimization, as shown in the following code:

```
fun wrapper x = let
  fun g i = i + x
  fun map f l =
    case l
     of h::t => (f h)^1::(map f t)
      | _ => []
in
  map g [1,2,3]
end
val res1 = wrapper 1
val res2 = wrapper 2
```

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 Higher-order copy propagation suffers from the same problem. In this case, instead of inlining the body of the function (*e.g.*, 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 known function. In the following code, the function g is passed as an argument to map and called in its body.

```
fun map f l =
  case l
   of h::t => (f h)::(map f t)
    | _ => []

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

When g is in scope at the call site inside map and either g has no free variables or we know that those free variables will always have the same values at both the capture point (when it is passed as an argument to map) 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 indirect calls. Applying them can also enable unused and useless variable elimination, as illustrated in the code resulting from the removal of the useless variable f:

```
fun map l =
  case l
   of h::t => (g h)::(map t)
    | _ => []

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

4.1 A Challenging Example

While most of the higher-order examples to this point could have been handled by more simple lexical heuristics and careful ordering of compiler optimization passes, those heuristics must be careful not to optimize in unsafe locations. The following example illustrates the importance of reasoning about environments when performing higher-order inlining on functions with free variables. The function mk takes an integer and returns a pair of a function of type (int -> int) * int -> int and a function of type int -> int; note that both of the returned functions capture the variable i, the argument to mk.

```
fun mk i =
  let
    fun g j = j + i
    fun f (h : int -> int, k)=
      (h (k * i))^1
  in
    (f, g)
  end
val (f1, g1) = mk 1
val (f2, g2) = mk 2
val res = f1 (g2, 3)
```

First, the function mk is called with 1, in order to capture the variable i in the closures of f1 and g1. Next, the function mk is called with 2, again capturing the variable i (but with a different value) in the closures of f2 and g2. Finally, we call f1 with the pair (g2, 3).

At the call site labeled 1, a simple 0CFA can determine that only the function `g` will ever be called. Unfortunately, if we inline the body of `g` at that location, as shown in the example code below, the result value `res` will change from `5` to `4`. The problem is that the binding of the variable `i` is not the same at the potential inline location as it was at its original capture location.

```
fun mk i =
  let
    fun g j = j + i
    fun f (h : int -> int, k) =
      ((k * i) + i)
  in
    (f, g)
  end
val (f1, g1) = mk 1
val (f2, g2) = mk 2
val res = f1 (g2, 3)
```

While this example is obviously contrived, this situation occurs regularly in idiomatic higher-order programs and the inability to handle the environment problem in general is a limit in most compilers, leading developers to avoid higher-order language features in performance-critical code.

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

```
let
  val y = m ()
  fun f _ = y
  fun g h = (h ())^1
in
  g f
end
```

At the call site labeled 1, it is clearly safe to inline the body of the function `f`, since `y` 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 even by compilers that perform CFA-based optimizations.

4.2 Reflow Analysis

A theoretical solution to this environment problem that enables a suite of additional optimizations is *reflow analysis* [Shi91]. This 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.

5. Unchanged Variable Analysis

The major contribution of this work is an *unchanged variable analysis*. Instead of performing reflow at each call site, we use a novel analysis that builds upon the approximate control-flow graph of the program given by a control-flow analysis to enable us to perform an inexpensive test at each call site. The optimizations from Section 4 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 Shivers' reflow analysis [Shi91], 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 approximate control-flow graph corresponding to the possible executions of this program, is there a path between those two locations through a rebinding of any of the free variables?

5.1 Building the approximate control-flow graph

The approximate control-flow graph is built in two steps. First, build a static control-flow graph for each function, ignoring function calls through variables, with vertices annotated with variable bindings and rebindings. Then, augment those individual function control-flow graphs with edges from the call sites through variables to the potential target functions, as determined by the control-flow analysis. Though we only discuss our implementation of 0CFA in this work, this alternative to reflow analysis also works with other control-flow analyses.

The variable bindings and rebindings in a program written in the continuation-passing style (CPS) representation defined in Figure 1 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 vertices in the graph for each function. One vertex 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 vertex is labeled with all of the parameters to the function, since they will also be bound when the function is called. Finally, any vertex corresponding to a **`let`**-binding in the control-flow graph will be labeled with the variable being bound.

Call sites are augmented using the results of the control-flow analysis described in Section 3. 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 $\bot$, 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 $\top$, indicating that any call site may be reached. In this case, we add an edge to a special vertex that represents any call site, whose optimization is discussed in Section 5.4.
- 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 reformulate 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 (re)binding location for any of the free variables of the function that we want to inline? If such a path exists, then any optimization that relies on the free variables maintaining their bindings between those locations may be unsafe.

For a program of size n (with $O(n)$ functions and $O(n)$ call sites), the approximate control-flow graph has $O(n)$ vertices and $O(n^2)$ edges. The worst-case quadratic number of edges corresponds to the situation where the control-flow analysis determines that every function in the program could be called from every call-site in the program. In practice, though, we expect the number of edges to be closer to linear in the size of the program, due to the fact that the utility of control-flow analysis is the ability to determine that only a small number of functions are called from each call-site. Hence, we will express the subsequent graph-algorithm complexities in terms of the number of vertices $|V|$ and the number of edges $|E|$ in the approximate control-flow graph. With the standard $O(n^3)$ 0CFA algorithm, constructing the approximate control-flow graph is $O(n^3) + O(|V| + |E|) = O(n^3)$. Note that running the control-flow analysis is required in order to both build the approximate control-flow graph and to identify candidate inlining opportunities.

5.2 Computing graph reachability quickly

This question about the existence of paths between vertices in the graph is a reachability problem. There are off-the-shelf $O(|V|^3)$ 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 vertices, they take seconds to run.

Therefore, we use an approach that collapses the graph quickly into a map we can use for logarithmic-time queries of the reachability between two vertices. Our approach performs two steps. First, we take the potentially cyclic graph and reduce it into a set of strongly-connected components. Then, we use a bottom-up approach to compute reachability in the resulting DAG. All queries are then performed against the resulting map from source component to set of reachable components.

Strongly-connected components We use Tarjan's $O(|V| + |E|)$ algorithm for computing the strongly-connected components [Tar72], as implemented in Standard ML of New Jersey by Matthias Blume.[2] This produces a directed acyclic graph (DAG), with $O(|V|)$ vertices (each corresponding to a strongly-connected component) and $O(|E|)$ edges (each corresponding one or more edges between vertices in the approximate control-flow graph that belong to distinct strongly-connected components); the collection of strongly-connected components are produced in topological sorted order. It also produces a map from each vertex in the approximate control-flow graph to its strongly-connected component in the DAG. There are two interesting types of components for this algorithm: those that correspond to exactly one vertex (program point) in the approximate control-flow graph and those that correspond to more than one vertex (program point). In the single vertex case, control-flow from that program point cannot reach itself. In the multiple vertex case, control-flow from each program point *can* reach itself. This distinction is crucial when initializing the reachability map.

Reachability in a DAG We compute a map from each strongly-connected component to its set of reachable components by processing the DAG in reverse topological sorted order. For each strongly-connected component, we initialize the reachability map for that component according to its size and then we add each successor component and everything that the successor component can reach. A more detailed description is shown in Algorithm 1.

Algorithm 1 Compute DAG reachability for a graph DAG

```
for $n \in \mathsf{Vertices}(DAG)$ in reverse topological sorted order do
    if $\mathsf{SCCSize}(n) = 1$ then
        $\mathsf{R}(n) \leftarrow \{\}$        ▷ Program point in $n$ cannot reach itself
    else
        $\mathsf{R}(n) \leftarrow \{n\}$ ▷ Each program point in $n$ can reach itself
    end if
    for $s \in \mathsf{Succs}(n)$ do
        $\mathsf{R}(n) \leftarrow \mathsf{R}(n) \cup \{s\} \cup \mathsf{R}(s)$
    end for
end for
```

We use a red-black tree to represent the set of reachable components, where the ordering of two strongly-connected components is given by the ordering of their "root" (i.e., representative) vertices. Furthermore, the implementation provides an $O(|s_1| + |s_2|)$ union operation [Hin99], better than a naive $O(|s_1| \log |s_2|)$ union operation via singleton inserts. Thus, the above is an $O(|E| * |V|)$ algorithm, dominated by the $\mathsf{R}(n) \cup \{s\} \cup \mathsf{R}(s)$ that is executed once per edge.

[2] This implementation uses a red-black tree to maintain per-vertex information and so incurs a cost of $O(\log |V|)$ to access the successor vertex's information when handling an edge, leading to an overall running time of $O(|V| + |E| \log |V|)$.

5.3 Performing the safety check

Having built the approximate control-flow graph and computed the strongly-connected components and reachability map, we can efficiently check the safety of a candidate inlining opportunity. We maintain a map from each variable to its set of (re)binding locations (vertices in the approximate control-flow graph). Given a candidate inlining opportunity, with a function-binding location and a call-site location, we check whether there exists a path from the function-binding location to a (re)binding location of a free variable and from the re(binding) location to the call-site location. Each of these path-existence checks is an $O(\log |E|)$ operation, performed by an $O(1)$ map from the source and destination locations (vertices in the approximate control-flow graph) to their strongly-connected components and an $O(\log |E|)$ query of the destination component in the set of components reachable from the source component. In practice, each candidate inlining opportunity has a small number of free variables that are (re)bound at a small number of locations, leading to analysis times that are less than 3% of total compilation time (Section 9.3).

5.4 Handling imprecision

In a 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 $\top$, or unknown. The obvious way to handle this situation when creating the graph is to add an edge from any call site labeled $\top$ to every possible function entry point. Unfortunately, that approach frequently connects the entire graph, preventing the compiler from proving that any variables remain unchanged through any non-trivial portions of the graph.

Instead, we take advantage of the fact that a call to an unknown function is really only a call to one of the functions that has unknown callers. We therefore add an edge from any call site labeled $\top$ to any function whose callers are not all known. These functions are identified during control-flow analysis, which in addition to computing the potential values that a variable can take on also tracks when a function is passed into a portion of the program that we cannot precisely analyze. Fortunately, that set of functions is small even for large programs, so the graph remains useful.

5.5 Limitations

While safe, this analysis necessarily is more limited than general formulations of higher-order inlining as shown by Shivers' kCFA framework (for $k > 0$) or Might's ΔCFA approach [Shi91, MS06]. 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, our attempt to inline at the call site labeled 1 will fail.

```
let val y = 2
  fun f _ = y
  fun confounding _ = raise Fail ""
  fun g h = (h ())$^1$
  fun callsG b k = if b then g k else 0
  val bad = callsG false confounding
in
  callsG true f
end
```

After the first call to `callsG`, the function `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 `confounding` to the list of possible values for `h`, when the second call comes through, the function `f` is added to the possible set of values for `k` and then *both* of those are added to the set of values that could be bound to `h`. Fundamentally, this problem is the one that stronger forms of control-flow analysis handle, though there are clearly heuristics that could be used to increase the precision in this specific case.

6. Safe example

In the introduction, we discussed an iterative version of the factorial function and pointed out that we might like to transform the argument that consumes the result of the computations to be either a direct call or inlined. A slightly modified version of that example appears below:

```
let
  fun fact i m k =
    if i = 0
    then k m
    else fact (i-1) (m*i) k
in
  fact 6 1 (fn i => h i)
end
```

In this example, the consuming function is an anonymous function that makes a call to another variable, `h`, which is free in this block. We would like to optimize this code by performing a higher-order inlining of that code to produce the following output:

```
let
  fun fact i m k =
    if i = 0
    then h m
    else fact (i-1) (m*i) k
in
  fact 6 1 (fn i => h i)
end
```

In order for that operation to be safe, though, we need to show two properties:

1. The variable `h` is in scope at the inlining location.
2. The variable `h` has the same binding at its inlining point as it did at the point where the closure would have captured it.

The first property is lexically immediate. In the rest of this section, we will demonstrate how unchanged variable analysis allows us to verify the second property.

6.1 Building the control-flow graph

The first step in unchanged variable analysis is construction of a control-flow graph. In order to make that graph easier to visualize, we have normalized the source code, broken bindings of arguments onto separate lines from bindings of function identifiers, and annotated the example with line numbers; the resulting program is given in Figure 3. The line numbers will be used in the rest of this section in the graph visualizations.

The static control-flow graph is shown with the solid lines in Figure 4. Note that this graph separates the actions of binding a variable of function type (such as `fact` in line 2) from the operation of actually running its body, which starts on line 3 of the listing with the binding of any free variables (in this case, none) and continues on line 4 with the binding of the parameters to arguments.

```
 1  let
 2      fun fact
 3          (* FV: *)
 4          i m k =
 5        if i = 0
 6        then let in  k m
 7             end
 8        else let val i' = i-1
 9                 val m' = m*i
10             in  fact i' m' k
11             end
12        (* fi *)
13      fun clos
14          (* FV: h *)
15          i =
16        let in  h i
17        end
18  in  fact 6 1 clos
19  end
```

Figure 3. Normalized source code for a safe example.

After running CFA on the factorial example above, we determine the following:

$$\begin{aligned}
\mathcal{V}(\texttt{fact}) &= \{\texttt{LAMBDAS}(\{\texttt{fact}\})\}\\
\mathcal{V}(\texttt{clos}) &= \{\texttt{LAMBDAS}(\{\texttt{clos}\})\}\\
\mathcal{V}(\texttt{k}) &= \{\texttt{LAMBDAS}(\{\texttt{clos}\})\}\\
\mathcal{V}(\texttt{i}) &= \{\top\}\\
\mathcal{V}(\texttt{i'}) &= \{\top\}\\
\mathcal{V}(\texttt{m}) &= \{\top\}\\
\mathcal{V}(\texttt{m'}) &= \{\top\}\\
\mathcal{V}(\texttt{j}) &= \{\top\}\\
\mathcal{V}(\texttt{h}) &= \{\top\}
\end{aligned}$$

The only interesting values in this finite map are the binding of the variable `k` to the function value `clos` and the binding of the variable `h` to the unknown target $\top$. These binding allow us to annotate the graph with higher-order control-flow paths from line 6 to line 14 and from line 17 to line 7 (corresponding to the call of and return from `clos` at the call site `k m` on line 6) and from line 16 to $\top$ and from $\top$ to line 17 (corresponding to the call of and return from an unknown function at the call site `h i` on line 16). These higher-order control-flow paths are shown with dotted lines in Figure 4.

Finally, the non-singleton strongly-connected components for the control-flow graph are show with boxed subgraphs in Figure 4.

6.2 Performing the unchanged variable analysis

Now, we are at a point where we know that `clos` is the only function being called at line 6, making it a candidate for inlining. But, its free variables (`h`) were captured at line 13, so we now need to check the graph for the following property:

> Does there exist a path starting from vertex 13 and ending at vertex 6 that passes through any vertex that rebinds variable `h`?

In this example, that property trivially holds, as there are *no* vertices in this subgraph where `h` is rebound and so the inlining is safe.

7. Unsafe example

For a negative case, we revisit the unsafe example from Section 4, repeated here:

```
fun mk i =
  let
    fun g j = j + i
    fun f (h : int -> int, k)=
      (h (k * i))¹
  in
```

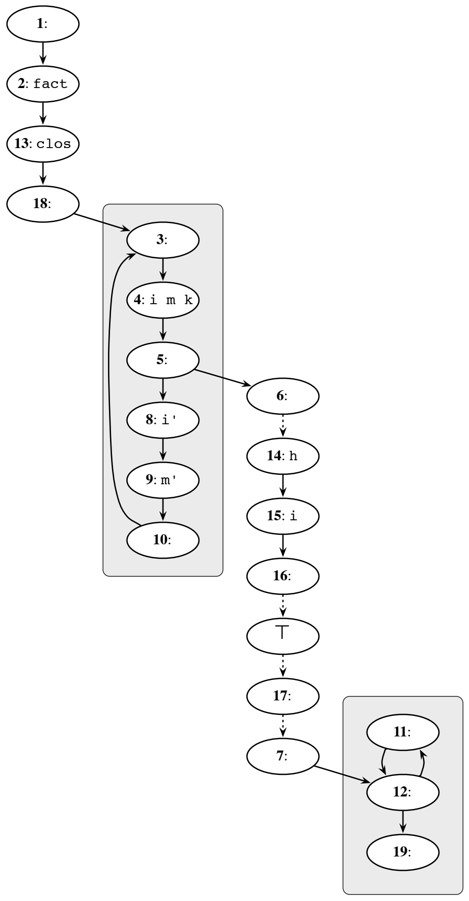

Figure 4. Control-flow graph for safe example.

```
    (f, g)
  end
val (f1, g1) = mk 1
val (f2, g2) = mk 2
val res = f1 (g2, 3)
```

In this example, control-flow analysis will determine that the function `g` is the only function that will be called at the call site labeled 1, making it a candidate for inlining. But, we need to determine whether or not it is safe to do so. That is, is `i`, the free variable of `g`, the same when `h` is invoked as when it was captured by `g`?

As in the safe example, the first step in our presentation is to normalize and annotate the example to aid in the visualization of the control-flow graph; the resulting program is given in Figure 5.

7.1 Building the control-flow graph

As in the previous section, we build a static control-flow graph (solid lines) and add higher-order control-flow paths (dotted lines) where control-flow analysis was able to determine the target of the call through a variable with function type. This graph is shown in Figure 6.

The interesting parts of the control-flow analysis results map are the following:

$$\begin{aligned}\mathcal{V}(\texttt{f1}) &= \{\texttt{LAMBDAS}(\{\texttt{f}\})\}\\ \mathcal{V}(\texttt{g1}) &= \{\texttt{LAMBDAS}(\{\texttt{g}\})\}\\ \mathcal{V}(\texttt{f2}) &= \{\texttt{LAMBDAS}(\{\texttt{f}\})\}\\ \mathcal{V}(\texttt{g2}) &= \{\texttt{LAMBDAS}(\{\texttt{g}\})\}\\ \mathcal{V}(\texttt{h}) &= \{\texttt{LAMBDAS}(\{\texttt{g}\})\}\end{aligned}$$

```
1   fun mk
2       (* FV: *)
3       i =
4     let
5         fun g
6             (* FV: i *)
7             j =
8           let val t1 = j + i
9           in  t1  end
10        fun f
11            (* FV: i *)
12            (h, k) =
13          let val t2 = k * i
14          in  h t2
15          end
16    in  (f, g)  end
17  val (f1, g1) = let in  mk 1
18                 end
19  val (f2, g2) = let in  mk 2
20                 end
21  val res = let in  f1 (g2, 3)
22            end
```

Figure 5. Normalized source code for an unsafe example.

These binding allow us to annotate the graph with higher-order control-flow paths from line 21 to line 11 and from line 15 to line 22 (corresponding to the call of and return from `f` at the call site `f1 (g2, 3)` on line 21) and from line 14 to line 6 and from line 9 to line 15 (corresponding to the call of and return from `g` at the call site `h t2` on line 14).

7.2 Performing the unchanged variable analysis

Control-flow analysis has informed us that we should be able to inline the body of function `g` at line 14. But, we now need to check the graph for the following property:

> Does there exist a path starting from vertex 5 and ending at vertex 14 that passes through any vertex (in this case, vertices 3 and 11) that rebinds variable `i`?

Since there exists such a path in the graph (*e.g.*, $5 \rightarrow 10 \rightarrow 16 \rightarrow 20 \rightarrow 21 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14$), this inlining is potentially (and actually!) unsafe, so it is disallowed under the unchanged variable condition tested in our system.

8. Example — Transducers

Transducers are program fragments that perform four tasks within an infinite loop: receive input, compute on that input, output a result, and loop back around. These fragments can then be composed together to form pipelined or stream processing programs which are used extensively in networks, graphics processing, and many other domains. Writing programs in this style gives developers modularity in the sense that when new functionality needs to be added, they can simply add a new transducer to the pipeline, eliminating the need to modify any substantial portion of existing code. Shivers and Might showed that if these transducers are implemented in a continuation-passing style, a number of standard optimizations, along with Super-β inlining, can effectively merge composed transducers into one loop that contains all the computation of the entire pipeline [SM06]. In this section, we show that unchanged variable analysis along with higher-order inlining is the first practical compiler implementation capable of performing these optimizations on transducers.

Figure 7 provides a library for building and composing transducers. Channels are used for passing information between transducers. Specifically, they are represented as continuations that take a value of type `'a` and another `chan`. A `dn_chan` is used for outputting information to the next transducer and an `up_chan` is

```
(* Type for channels *)
  datatype ('a, 'b) chan = Chan of ('a * ('b, 'a) chan) cont

(* Types for the specific kinds of channels *)
  type 'a dn_chan = ('a, unit) chan
  type 'a up_chan   = (unit, 'a) chan

(* Source/Sink (first/last in the chain) *)
  type ('a, 'r) source = 'a dn_chan -> 'r
  type ('a, 'r) sink   = 'a up_chan -> 'r

(* transducer (middle in the chain) *)
  type ('a, 'b, 'r) transducer = 'a up_chan * 'b dn_chan -> 'r

(* change control upstream or downstream *)
  fun switch (x : 'a, Chan k : ('a, 'b) chan) : 'b * ('a, 'b) chan = callcc (fn k' => throw k (x, Chan k'))

(* Put value x on down channel dnC *)
  fun put (x : 'a, dnC : 'a dn_chan) : 'a dn_chan = (case switch (x, dnC) of ((), dnC') => dnC')

(* Get a value from up channel upC *)
  fun get (upC : 'a up_chan) : 'a * 'a up_chan = (case switch ((), upC) of (x, upC') => (x, upC'))

(* Compose sources, transducers, and sinks. *)
  fun sourceToTrans (source : ('a, 'r) source, trans : ('a, 'b, 'r) transducer) : ('b, 'r) source =
        fn (dnC : 'b dn_chan) => callcc (fn k =>
          source (case callcc (fn upK => throw k (trans (Chan upK, dnC))) of (_, upC') => upC'))


  fun transToSink (trans : ('a, 'b, 'r) transducer, sink : ('b, 'r) sink) : ('a, 'r) sink =
        fn (upC : 'a up_chan) => callcc (fn k =>
          trans (upC, case callcc (fn upK => throw k (sink (Chan upK))) of ((), dnC') => dnC'))

  fun sourceToSink (source : ('a, 'r) source, sink : ('a, 'r) sink) : 'r =
        callcc (fn k => source ( case callcc (fn upK => throw k (sink (Chan upK))) of ((), dnC) => dnC))
```

Figure 7. Transducer library code.

used for receiving information from the previous transducer. The put function throws to a dn_chan, giving it a value and its current continuation wrapped in a chan. When this continuation is invoked, it will return a new dn_chan. The get function throws to an up_chan, giving it a unit value and its current continuation wrapped in a Chan constructor. When this continuation is invoked, it will return the value being passed down to the transducer as well as a new up_chan.

8.1 Simple composition

A pipeline is composed of a source at the beginning, zero or more transducers in the middle, and a sink at the end. The sourceToTrans function is used to link a source to a transducer, yielding a new source. Similarly, the transToSink function is used to link a transducer to a sink, yielding a new sink. Linking a source to a sink with the sourceToSink function executes the transducer pipeline.

Figure 8 illustrates a simple stream of transducers, where the source infinitely loops, outputting the value 5 to the sink, which then prints this value each time. These two functions are then composed using the sourceToSink function. Ideally, we would like to generate code that merges these transducers together, yielding one tight loop that simply prints the value five in each iteration, rather than passing control back and forth between these two co-routines.

8.2 Optimization

In order to fuse these two co-routines, we need to be able to inline the calls to the co-routines, which requires Super-β analysis, as noted by Shivers and Might [SM06]. Running the analysis and inlining performed in this paper successfully fuses those co-routines and removes the creation of the closure across that boundary. For example, running the transducer shown in Figure 8 for 10,000 steps, we reduce the overall memory usage from 4.6M to 3.9M, for a savings of roughly 15%. The remaining memory usage is almost entirely in internal library calls due to the print function (which is not well-optimized in Manticore).

9. Evaluation

In this section, we show that this analysis is both practical and effective. In Section 9.3, we show that the compile-time cost of adding this analysis is under 3% of the total compilation time. Section 9.4 provides support that the optimizations provided by this analysis are both found and typically result in performance improvements in our benchmarks.

9.1 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 12.04.3, kernel version 3.2.0-49. We ran each benchmark experiment 30 times, and speedups are based upon the median runtimes. Times are reported in seconds.

This work has been implemented, tested, and is part of the current Manticore compiler's default optimization suite.

9.2 Benchmarks

For our empirical evaluation, we use seven benchmark programs from our parallel benchmark suite and one synthetic transducer benchmark. 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

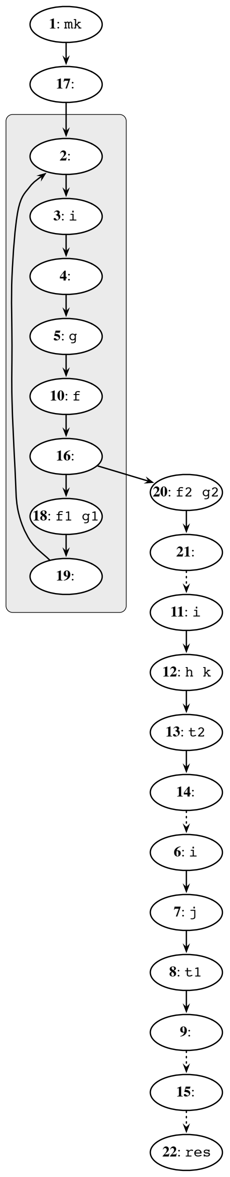

Figure 6. Control-flow graph for unsafe example.

```
(* Source *)
fun putFive (dnC : int dn_chan) =
      putFive (put (5, dnC))

(* Sink *)
fun printVal (upC : int up_chan) =
      let val (x, upC') = get upC
          val _ = print (Int.toString x)
      in  printVal upC'
      end

(* Run *)
val _ = sourceToSink (putFive, printVal)
```

Figure 8. Transducer example.

runs 20 iterations over 400,000 particles generated in a random Plummer distribution. Our version is a translation of a Haskell program [GHC].

The DMM benchmark performs dense-matrix by dense-matrix multiplication in which each matrix is 600×600.

The Raytracer benchmark renders a 2048×2048 image as a two-dimensional sequence, which is then written to a file. The original program was written in ID [Nik91] 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 4096×4096.

The Quickhull benchmark determines the convex hull of 12,000,000 points in the plane. Our code is based on the algorithm by Barber *et al.* [BDH96].

The Quicksort benchmark sorts a sequence of 10,000,000 integers in parallel. This code is based on the NESL version of the algorithm [Sca].

The SMVM benchmark performs a sparse-matrix by dense-vector multiplication. The matrix contains 3,005,788 elements, and the vector contains 10,000, and the multiplication is iterated 75 times.

In addition to the parallel benchmarks, the transducer benchmark is the sequential benchmark described in Section 8. For benchmarking purposes, we simulate running the transducer through 2,000,000 iterations.

9.3 Compilation performance

In Table 1, we have broken down the compilation time of the larger 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 1. By that stage in the compilation process, all unreferenced and dead code has been removed from the program.

The most important results are:

- Control-flow analysis is basically free.
- The unchanged variable 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.

9.4 Benchmark performance

Across our already tuned benchmark suite, we see several improvements and only one statistically significant slowdown, as shown in Table 2. It might seem strange that the number of inlinings is different for the sequential and parallel implementations of each benchmark, but this is due to the fact that the parallel implementations use more sophisticated runtime library functions, exposing more opportunities for optimization. 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. So, the primary result that we have to show in this section for existing benchmarks is that this optimization, even performed using only a simple size-based heuristic, does not harm our tuned performance by more than 0.5% in the worst case (and within one standard deviation of the performance) and in some cases results in

Benchmark	Lines	Expressions	Total (s)	CFA (s)	Copy Prop. (s)	H-O Inline (s)	GCC (s)
Barnes-hut	334	17,400	8.79	0.042	0.175	0.198	2.56
Raytracer	501	12,800	6.54	0.019	0.112	0.124	2.64
Mandelbrot	85	9,900	5.06	0.013	0.091	0.098	1.70
Quickhull	196	15,200	7.67	0.039	0.182	0.177	2.05
Quicksort	74	11,900	5.49	0.022	0.111	0.122	1.11
SMVM	106	13,900	7.25	0.033	0.131	0.123	2.52

Table 1. Benchmark program sizes, both in source lines and total number of expressions in our whole-program compilation. Costs of the analyses and optimizations are also provided, in seconds.

gains of around 1%. This optimization can result in slowdowns, due to increasing the live range of variables and the resulting increase in register pressure.

In the one example program that has not already been tuned so far that there are no higher-order optimization opportunities within hot code — the transducer benchmark described in Section 8 — we see a speedup of 4.7% due to removing the need for a closure within an inner loop.

10. 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, ΔCFA, which more explicitly tracks environment representations and can check for safety without re-running the analysis at each inlining point [MS06]. Unfortunately, this approach also only works in theory — while its runtime is faster in practice than a full 1CFA (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 vertices in our graphs. A different algorithm for graph reachability that has even better asymptotic performance than the one we present in Section 5.2 is also available [Nuu94], 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.

The Glasgow Haskell Compiler has an extremely sophisticated inliner that has been tuned for many years, using a variety of type-, annotation-, and heuristic-based techniques for improving the performance of programs through effective inlining [PM02]. However, even after inlining and final simplification, this compiler cannot inline the straightforward higher-order example in Section 6.

11. Conclusion

In this work, we have demonstrated the first *practical* and general approach to higher-order inlining and copy propagation. We hope that this work ushers in new interest and experimentation in environment-aware optimizations for higher-order languages.

11.1 Limitations

As with all optimizations, this analysis and optimization are fragile with respect to changes to the code being optimized. Making things even more unpredictable for the developer, the output of control-flow analysis can also be affected by non-local changes if those changes cause the analysis to hit performance cutoffs and default to conservative worst-case partial results. We believe that adopting monomorphization, as used in the MLton compiler [Wee06], would both increase the precision of the analysis' results and remove the largest sources of imprecision in control-flow analysis — large numbers of polymorphic uses of common combinators such as `map` and `fold`.

11.2 Future work

This work identifies opportunities for performing optimizations, but does not investigate the space of heuristics for when they are beneficial. We currently perform the copy propagation unconditionally and perform the higher-order inlining using the same simple code-growth metric that we use for standard inlining. But, these optimizations could introduce other negative impacts on some programs, as it might increase the live range of variables. Identification of these negative impacts and heuristics for avoiding them is left to future work.

We have also provided an implementation of an analysis that shows when free variables are unchanged along a control-flow path, but we have not generated a formal proof that these optimizations are correct.

Further, we have not investigated other optimizations, such as rematerialization, that were presented in some of Might's recent

	Sequential			16 Processors		
Benchmark	Speedup	Copy Prop.	Inlined	Speedup	Copy Prop.	Inlined
Barnes-hut	1.2%	12	11	0%	15	17
DMM	0.3%	3	11	0.8%	6	17
Mandelbrot	-0.3%	0	3	0.3%	3	9
Quickhull	0.3%	12	11	0.3%	15	17
Quicksort	1.5%	2	4	0%	5	9
Raytracer	-0.3%	0	3	-0.2%	3	9
SMVM	0.4%	2	14	-0.5%	5	21
Transducer	4.7%	1	3	*N/A*	*N/A*	*N/A*

Table 2. Performance results from copy propagation and higher-order inlining optimizations.

work on anodization [Mig10] and might have an analog in our framework.

Finally, 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 [Hud86], which resulted in improvements in both runtime and precision,[3] but that implementation is not yet mature [Ber09].

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. One anonymous reviewer commented extensively on an earlier draft of this work, substantially improving its presentation.

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.

[AM91] 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.

[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/`.

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

[FFR+07] 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.

[FRRS11] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, **20**(5–6), 2011, pp. 537–576.

[GGR94] George, L., F. Guillame, 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/`.

[Hin99] Hinze, R. Constructing red-black trees. In *WAAAPL'99: Workshop on Algorithmic Aspects of Advanced Programming Languages*, Paris, France, 1999. pp. 89–99.

[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.

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

[MS06] Might, M. and O. Shivers. Environment analysis via ΔCFA. In *POPL '06*, Charleston, South Carolina, USA, 2006. ACM, pp. 127–140.

[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.

[Nuu94] Nuutila, E. An efficient transitive closure algorithm for cyclic digraphs. *IPL*, **52**, 1994.

[PM02] Peyton Jones, S. and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, **12**(5), July 2002.

[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.

[3] Best results were achieved when using a `maxrc` of 1.

[SM06] Shivers, O. and M. Might. Continuations and transducer composition. In *PLDI '06*, Ottawa, Ontario, Canada, 2006. ACM, pp. 295–307.

[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.

[Wee06] Weeks, S. Whole program compilation in MLton. Invited talk at ML '06 Workshop, September 2006.