THE UNIVERSITY OF CHICAGO

ARITY RAISING AND CONTROL-FLOW ANALYSIS IN MANTICORE

A PAPER SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY

LARS BERGSTROM

CHICAGO, ILLINOIS

NOVEMBER 13, 2009

ABSTRACT

Manticore is a programming language designed for the execution of general-purpose parallel programs. Manticore is based on the language Standard ML, includes a wide variety of implicit and explicit features for parallelism, and provides concurrency abstractions based upon Concurrent ML. All of these features rely on good sequential performance, which this work is focused on improving.

We present a new control-flow analysis technique, reference counted control-flow analysis (RCCFA). We compare the behavior and performance of RCCFA with several popular control-flow algorithms and provide a new result: in implementation, a collected CFA can perform more slowly than a non-collected CFA.

We also provide a novel approach to arity raising that incorporates both unboxing and datatype flattening in a single optimization. These optimizations are currently performed using either a type-directed or reduced quality control-flow analysis and are performed in separate stages by other modern functional language compilers. We show that our arity raising algorithm is effective at reducing code size, decreasing the dynamic number of bytes allocated, and speeding up execution times for different types of programs.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Halfway between the input source code (Parallel ML) and the final output binary, the Manticore compiler [FRR+07] uses a Continuation Passing Style (CPS) representation of the program that is very similar to that of SML/NJ [App92]. One optimization we perform on this representation is arity raising.

Arity raising (also known as argument flattening) is the process of transforming individual parameters of a function from heap-allocated records and tuples of data into the individual data elements that are used within that function. This optimization is performed to reduce overhead associated with accessing data in memory instead of registers. During the early stages of the compiler, all functions with more than one parameter are represented as functions with a single parameter that bundles up all of the original parameters into a tuple. As we near code generation, we want to promote appropriate members of the tuple back into parameters both so that the code generator can use registers instead of the heap to pass arguments and to reduce the number of heap allocations and selections.

Additionally, we want to use arity raising to remove unnecessary data structures and overhead on raw data types. Unnecessary data structures arise when a datatype definition is created to hold data but we can replace allocating a structure in the heap by just passing the underlying data in registers. Overhead on raw data types results from boxing, which is when a raw data type like a float or integer needs to be stored in memory and ends up wrapped in a larger piece of memory or a modified form that the runtime can understand. This boxing can be removed when we know that the raw data can be passed directly as an argument to a function.

Some modern functional language compilers, like MLton, already use control-flow analysis to drive optimizations like arity raising [ZWJ08]. Others like GHC use a purely type-directed approach [BPJ09]. In this work, we show an extension to arity raising using control-flow analysis and that simultaneously removes boxing and flattens datatypes. This new approach to arity raising reduces static code size and improves both dynamic performance and memory usage for several types of programs.

We also introduce a new control-flow analysis technique, called reference counted control-flow analysis (RCCFA). This new analysis is usually faster and always more precise than 0CFA and requires very little additional implementation work over 0CFA. We give examples of the behavior of different analysis techniques and show a novel example of when control-flow analyses that perform collection of their abstract environment will perform more slowly than analyses that do not perform collection.

CHAPTER 2

CONTROL-FLOW ANALYSIS

Control-flow analysis (CFA) is a technique for determining information about a program useful to optimizations at compile time. In the context of a functional programming language, control-flow analysis determines binding information for variables. This information can be used to directly answer questions of the form:

- What functions or values can this variable take on?
- To which variables can this function or value be bound?

A piece of code and the results of control-flow analysis are provided below. This example defines a function `double` that takes an argument and adds it to itself. The example also defines a second function, `apply`, which takes a function and an argument, applying the function to the argument. Notice the call site marked with α.

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

After running CFA on the example above, we should have results similar to these:

$\texttt{f} = \{\texttt{double}\}$

$\texttt{n} = \{\mathbb{N}\}$

$\texttt{x} = \{\mathbb{N}\}$

Control-flow analysis tells us which functions can be called at the application site α through the variable `f` — in this case, just `double`. The variables `n` and `x` will never be bound to function values and are simply natural numbers. This information is very useful for optimizations. If there is only one function called at a call site and if the callee either has no free variables or shares the same environment as the caller, then the callee can be either inlined or the function can be called directly instead of through the variable. Even if there are multiple functions that can be called at a given site, just the knowledge of the concrete list of callees enables optimization of argument passing — if the compiler knows that it is compiling both all callers and callees, the compiler can ignore the default calling convention of the platform and use one that is more efficient. The optimization of the calling convention at call sites where all of the target functions are known is the subject of Chapter 3.

Given control-flow information, further optimizations are often available. If a function is never called dynamically, it can be removed from the function and any remaining static references can be removed as well. If no functions are ever called at a call site, then that call site is part of a block of dead code that can be eliminated.

Some implementations only track function values and treat all non-function values uniformly. In Manticore we use a mix: we track both functions and datatypes, but largely ignore specific non-function values like numerics. Specific values are frequently ignored because the type information we preserve during compilation provides us with coarse-grained information (like `int` or `float`) that is sufficient for our value optimization needs. We do track boolean values of `true` and `false` because that allows control-flow analysis to limit the branches analysed to only those that could possibly be taken. For example, in the following example, if control-flow analysis tracks boolean values, then it can determine that `otherFun` is never called. If only function values are tracked by the analysis, then the analysis will conservatively analyse both arms of the conditional and assume that `otherFun` could be called.

```
let fun otherFun () = ...
and fun doIt (b) =
  if b
```

```
  then otherFun()
  else 3
in
  doIt (false)
end
```

2.1 Sources of Imprecision

An implementation of control-flow analysis can give imprecise answers for two reasons:[1]

1. Externally defined or exposed functions
2. Loss of precision due to abstraction of the value space

The first reason is straightforward, as shown in the code below.

```
fun apply (f, n) = f (n)
```

In a compiler that separately analyses each module of code, there is no way to determine in isolation what arguments the function `apply` is called with. Any modular control-flow analysis treats `f` as a function variable that can be bound to anything.[2] A whole program compiler like MLton [Wee06] can do better, but for any compiler that performs separate compilation, exposed functions must either be treated conservatively or transformed so that there is a version of the function for external calls that cannot be optimized and one for internal calls that can be optimized.

Even for compilers that are whole-program, externally defined functions prove a challenge to precision. If a language provides access to code defined in a low-level language

---

1. Apart from software implementation defects, of course.

2. The inability to fully transform or optimize public code is just one more reason to be very careful about any APIs you provide.

like C without strict guarantees on how data is handled, then any values flowing into or out of the C function must be treated conservatively. Concretely, the imprecision of C code means that most control-flow analyses assume that the return values of generic C functions could be of any form and that any value passed to a C function can escape and be stored for an unlimited time.

The second problem — abstraction of the value space — is much more challenging. Tradeoffs in precision of and management of the abstract value space are the defining characteristic of the different control-flow analysis algorithms. As one extreme example, the most-precise version of control-flow analysis would execute the program against all possible inputs, recording every value ever stored into each variable. This recording strategy would provide complete information, but is obviously intractible as a general compiler analysis strategy. Most of the different control-flow analyses vary the way they track environment information or control-flow graph information in order to change the runtime of the algorithm and the precision of the results. Environment information is used in the garbage-collecting control-flow analysis (ΓCFA) of Might [MS06] to ensure that old abstract values that are no longer reachable are removed. Control-flow graph information is used in the kCFA analyses of Shivers [Shi91] to separate abstract values by the call chain that set the value. We describe two approaches to implementing 0CFA in detail as well as a conservative approach to garbage-collected control-flow analysis.

2.2 Gathering Information

Control-flow analysis builds an *abstract environment*, mapping each variable in the program to an abstract value. Abstract values are an approximate set of values that can be taken on by variables in place of the real concrete values during an actual program execution, used to reduce the amount of time it takes for an analysis to converge on an answer. At the start of analysis, all variables are mapped to an abstract value of $\perp$ (bottom), indicating that nothing is known about the value of the variable. Any variables that are externally defined or set (i.e. globally, as arguments from the `main` entry function, or from unsafe C code) are assigned a value of $\top$ (top), indicating that the variable can take on any value. The crux of the information gathering — and the differentating factor between each of the different control-flow analysis styles — lies in how the program is traversed and the abstract environment mapping variables to abstract values is performed.

The *precision* of an abstract value is a relative comparison within the lattice of abstract values. For example, in an abstract value domain consisting of the value $\top$, $\bot$, and the powerset of the functions in a program that had `f` and `g`, we have the following lattice:

$$
\begin{tikzcd}
 & \top \arrow[d] & \\
 & \{\mathtt{f},\mathtt{g}\} \arrow[dl] \arrow[dr] & \\
\{\mathtt{f}\} \arrow[dr] & & \{\mathtt{g}\} \arrow[dl] \\
 & \bot &
\end{tikzcd}
$$

In the lattice above, the subset inclusion relationship defines the precision relationship between members of the powerset of the functions. The value $\bot$ means that no functions can be bound to a variable or call site, and is the most precise value. The value $\top$ means that any function can be bound to a variable or call site, and is the least precise value.

2.2.1 Intermediate Representation

We use the intermediate representation in Figure 2.1 to describe programs that we will perform several CFA analysis algorithms over. This representation is in direct style, rather than continuation-passing style (CPS). In Manticore, we perform control-flow analysis on a CPS intermediate representation. CFA can be performed on either representation. Some of the details of control-flow analysis change between represen-

$Exp \ni e$	$::=$	x	variable or function name
	$\mid$	$\text{fun } f(\vec{x}) = e_1 \text{ in } e_2$	function binding
	$\mid$	$\text{let } x = e_1 \text{ in } e_2$	local variable binding
	$\mid$	$\text{if } x \text{ then } e_1 \text{ else } e_2$	conditional
	$\mid$	$f^l(\vec{x})$	application (labeled)
	$\mid$	$\langle \vec{x} \rangle$	tuple creation
	$\mid$	$\#i(x)$	tuple selection
	$\mid$	b	boolean
$i \in \mathbb{N}$			literal integers
$l \in \mathbb{L}$			labels
$b \in \{\text{true}, \text{false}\}$			boolean values

Figure 2.1: Direct style intermediate representation.

tations, but the techniques and data gathered are fundamentally similar,[3] particularly if return points are annotated and tracked in the direct style [MJ09].

2.2.2 Naive Implementation

The naive implementation of control-flow analysis performs an analysis that simulates a literal execution of the program and records all concrete values bound to variables. While this strategy should never be used in a production compiler,[4] it is presented below in order to introduce notation and to provide a basis for contrasting the control-flow algorithms presented later. The largest difference between different control-flow algorithms is how they decide when and within what context to perform analysis of pieces of the program. We assume in this and all other presented control-flow analysis

3. This similarity should not be surprising because of Kelesy's work [Kel95] showing that a program in SSA form can be converted to CPS form and most CPS programs can be translated back to SSA. Interestingly, we can use the flow information computed by CFA to reconstruct the continuation labels required for his CPS $\rightarrow$ SSA transformation.

4. In addition to performing far more work than is necessary, this algorithm is not guaranteed to terminate.

algorithms that the program is valid and that all variables are uniquely named.

The goal of control-flow analysis is to build up a function $\mathcal{A}$, representing an abstract environment that maps from variables to abstract values. Variables are any of the identifiers in the program, and are introduced either on the left hand side of a let binding or in the function and parameter name positions of a function definition. A $FunID$ is a tuple of the function's name, the expression representing its body, and the list of variables that are its parameters. Finally, abstract values are taken from the set containing $\top$, $\bot$, any number of function identifiers, or a tuple of abstract values.

$\mathcal{A} : \mathit{Var} \rightarrow \mathit{AbsValue}$

$x : \mathit{Var}$

$x \in \mathit{FunId} \times \mathit{Identifier}$

$v : \mathit{AbsValue}$

$v \in \top \ \cup \ \bot \ \cup \ 2^{\mathit{FunId}} \ \cup \ \mathit{Tuple}(\mathit{AbsValue}^{*})$

Though the control-flow analysis implemented in Manticore tracks boolean values to improve precision and decrease runtime, all non-function values are tracked as the abstract value $\top$ to reduce the complexity of the presentation.

The function $\mathbb{C}$, defined in Figure 2.2, builds up the function $\mathcal{A}$ via an analysis of the program and returns an abstract value corresponding to an evaluation of the expression provided. All inputs to the $\mathcal{A}$ function initially map to $\bot$, representing unknown. Over the course of the analysis of the function $\mathbb{C}$, additional mappings from variables to abstract values are added to the function $\mathcal{A}$. The operator $\oplus$ defines abstract value merging on variable values. This operator also lifts to work over vectors of variables. The ρ parameter is a local environment mapping from variables to abstract values. The local environment is extended via operations of the form $\rho\{x \mapsto v\}$, which means that the environment ρ is extended to map requests for the variable x to the abstract value v. The notation $\rho[x]$ means to look up the abstract value associated with the variable x in the environment ρ.

Tuples are introduced with the $\langle\rangle$ notation. Tuple selection is performed with the

$$
\begin{array}{rcl}
\mathbb{C}[\![\cdot]\!] & : & \mathit{Exp} \rightarrow \mathit{Env} \rightarrow \mathit{AbsValue} \\
\mathbb{C}[\![x]\!]\rho & = & \rho[x] \\
\mathbb{C}[\![\text{fun } f(\vec{x}) = e_1 \text{ in } e_2]\!]\rho & = & \mathbb{C}[\![e_2]\!]\rho\{f \mapsto \lambda(\vec{x})e_1\} \\
\mathbb{C}[\![\text{let } x = e_1 \text{ in } e_2]\!]\rho & = & \mathcal{A}[x] := \mathcal{A}[x] \oplus \mathbb{C}[\![e_1]\!]\rho ; \mathbb{C}[\![e_2]\!]\rho\{x \mapsto \mathbb{C}[\![e_1]\!]\rho\} \\
\mathbb{C}[\![\text{if } x \text{ then } e_1 \text{ else } e_2]\!]\rho & = & \mathbb{C}[\![e_1]\!]\rho \oplus \mathbb{C}[\![e_2]\!]\rho \\
\mathbb{C}[\![f(\vec{x})]\!]\rho & = & \forall f' \in \rho[f](\mathbb{C}[\![f'.\mathit{body}]\!]\{f'.\mathit{params} \ \mapsto \vec{x}\}) \\
\mathbb{C}[\![\langle\vec{x}\rangle]\!]\rho & = & \langle\vec{x}\rangle \\
\mathbb{C}[\![\#i(x)]\!]\rho & = & \#i(\rho[x]) \\
\mathbb{C}[\![b]\!]\rho & = & b
\end{array}
$$

$$
\begin{array}{rcl}
\oplus & : & (\mathit{AbsValue} \times \mathit{AbsValue}) \rightarrow \mathit{AbsValue} \\
\bot \oplus v & = & v \\
v \oplus \bot & = & v \\
f_1 \oplus f_2 & = & f_1 \cup f_2 \text{ where } f_1, f_2 \in 2^{FunID} \\
\langle\vec{v}_1\rangle \oplus \langle\vec{v}_2\rangle & = & \vec{v}_1 \oplus \ \vec{v}_2 \\
\mathit{otherwise} & = & \top
\end{array}
$$

Figure 2.2: Naive control-flow analysis.

notation $\#i(v)$, meaning to select the i'th member of the tuple from the abstract value v. In the cases where the value is not a tuple, selection returns $\bot$ for the value $\bot$ and $\top$ otherwise. Selection of the body expression and parameter list from a *FunId* are performed by v.body and v.params, respectively.

The largest problem with the naive control-flow analysis implementation is in the analysis of function application. Each time a function application is uncovered, the bodies of any functions that could be called from that point are reanalysed in the context of an environment with the parameters mapped to the passed-in values. For example, the naive control-flow analysis will fail to terminate on the following example because at the recursive call site α the analysis will restart evaluation of the function `fact`.

```
let fun fact (n) =
  if n = 0
  then 1
  else n * factα (n-1)
in
```

```
  fact (1)
end
```

2.3 Shivers' 0CFA

Shivers' work on control-flow analysis [Shi88] brought to light the high quality optimizations that could be performed on scheme [ADH^{+}98] code when control-flow analysis was applied carefully to higher-order language constructs. The 0CFA algorithm he introduced in that work has not been widely implemented directly as presented, but led to follow-on algorithms that have been used extensively, one of which is described in Section 2.4. The source listing in Section 2.3.1 highlights interesting aspects of the 0CFA algorithm as presented in Shivers' work.

The algorithm is a walk over the program, similar to the naive analysis in Figure 2.2. Rather than keeping an explicit environment, the abstract value function $\mathcal{A}$ is used to look up values. Every time a variable is assigned a new value, we merge the new value to the old values bound to that variable. At each call site, for each of the potential function targets (as determined by the binding of the variable being called through), a check is made to see if that target function has been analysed with those arguments by checking a global store. If they have already been analysed, then the walk through the program continues. If the analysis has not been performed on the provided values, then a recursive walk is performed starting in the target function with those target argument values. The *Time-Stamp Approximation* used in the implementation presented is based on numbering the updates to the variable store and using the update number as a proxy for whether or not a target function has been processed with a set of values. This conservative approximation was introduced in Shivers' thesis [Shi91].

This analysis performs only one top-level pass over the program. Depending on how long it takes for the parameters to no longer grow, in the worst case each call site can potentially cause a recursive evaluation of the entire program, for polynomial complexity. In practice, 0CFA converges very quickly on most programs and operates within a small constant factor of a single pass over the IR.

2.3.1 Implementation Details

Instead of an environment parameter ρ, we now pass in a timestamp value, `t`. The timestamp is zero at the start of the analysis and is incremented during analysis of expressions that could add a new mapping to the abstract value function $\mathcal{A}$. The function $\mathcal{A}$ now also returns a timestamp along with the abstract value bound to a variable. If a variable not yet bound is requested, the pair $\{\perp,\ 0\}$ is returned. The other definitions of basic types are unchanged from the naive implementation.

$\mathcal{A} : \mathit{Var} \rightarrow \mathit{AbsValue}\ \times\ \mathit{Integer}$

The implementation of this algorithm, along with the necessary changes to the $\oplus$ function to support retaining the maximum timestamp value of the two encountered values, is in Figure 2.3. There is also a new function, $\mathcal{R}$ that returns the latest result seen from a function with an approximation of value arguments. This shortcut, applied when performing analysis of function applications in the intermediate representation, is the key to limiting the execution time of the kCFA family of control-flow analysis algorithms as presented by Shivers. By limiting reprocessing of function bodies to only happen when there is a newer timestamp on one of the provided arguments (and therefore there exist new bindings for the abstract value of one of those arguments), an upper bound is placed on the execution time.

At each variable introduction site, we stamp the variable with the current timestamp and continue analysis with an incremented timestamp. When we process function application sites, we first get the set of $FunId$ tuples and check the timestamp of each of them against the timestamps associated with the variables storing the arguments. If the timestamp on the function is newer than all of the timestamps of the arguments, then we simply return the previous computed result for the function. If the timestamp on any of the arguments is newer than the timestamp associated with the $FunId$, then we reanalyse the body of the function. When analysing conditionals, we traverse

$$
\begin{array}{rcl}
\mathbb{C}[\![\cdot]\!] & : & \mathit{Exp} \to \mathit{Integer} \to \mathit{AbsValue} \\
\mathbb{C}[\![x]\!]\mathrm{t} & = & \mathcal{A}[x] \\
\mathbb{C}[\![\mathrm{fun}\ f(\vec{x}) = e_1\ \mathrm{in}\ e_2]\!]\mathrm{t} & = & \mathcal{A}[f] := \mathcal{A}[f] \oplus \{\lambda(\vec{x})e_1, \mathrm{t}\}; \mathbb{C}[\![e_2]\!]\mathrm{t}' \\
 & & \qquad \text{where } \mathrm{t}' = \ \mathrm{t}\ + 1 \\
\mathbb{C}[\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2]\!]\mathrm{t} & = & \mathcal{A}[x] := \mathcal{A}[x] \oplus \{\mathbb{C}[\![e_1]\!]\mathrm{t},\mathrm{t}\}; \mathbb{C}[\![e_2]\!]\mathrm{t}' \\
 & & \qquad \text{where } \mathrm{t}' = \ \mathrm{t}\ + 1 \\
\mathbb{C}[\![\mathrm{if}\ x\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2]\!]\mathrm{t} & = & \mathbb{C}[\![e_1]\!]\mathrm{t} \oplus \mathbb{C}[\![e_2]\!]\mathrm{t'} \\
\mathbb{C}[\![f(\vec{x})]\!]\mathrm{t} & = & \displaystyle\bigoplus_{f' \in \mathcal{A}[f]} \begin{cases} \mathcal{R}[f'] & \text{when } \mathcal{A}[f'] \text{ newest} \\ & \\ \mathcal{A}[y_i] := \mathcal{A}[y_i] \oplus \mathcal{A}[x_i] & \text{otherwise} \\ \quad \text{where } y_i \text{ is the } i\text{th param of } f' & \\ \quad \text{and } x_i \text{ is the } i\text{th arg to } f' & \\ \mathcal{R}[f'] := \mathbb{C}[\![f'.\mathrm{body}]\!]\mathrm{t} & \\ \mathbb{C}[\![f'.\mathrm{body}]\!]\mathrm{t} & \end{cases} \\
\mathbb{C}[\![\langle \vec{x} \rangle]\!]\mathrm{t} & = & \langle \vec{x} \rangle \\
\mathbb{C}[\![\#i(x)]\!]\mathrm{t} & = & \#i(\mathcal{A}[x]) \\
\mathbb{C}[\![b]\!]\mathrm{t} & = & b
\end{array}
$$

$$
\begin{array}{rcl}
\oplus & : & ((\mathit{AbsValue} \times \mathit{Integer}) \times (\mathit{AbsValue} \times \mathit{Integer})) \to \\
 & & (\mathit{AbsValue} \times \mathit{Integer}) \\
(\bot, t_1) \oplus (v, t_2) & = & (v, \max(t_1, t_2)) \\
(v, t_1) \oplus (\bot, t_2) & = & (v, \max(t_1, t_2)) \\
(F_1, t_1) \oplus (F_2, t_2) & = & (F_1 \cup F_2, , \max(t_1, t_2)) \text{ where } F_1, F_2 \in 2^{\mathit{FunID}} \\
(\langle \vec{v}_1 \rangle, t_1) \oplus (\langle \vec{v}_2 \rangle, t_2) & = & (\vec{v}_1 \oplus \ \vec{v}_2, \max(t_1, t_2)) \\
(\mathit{other}, t_1) \oplus (\ \mathit{other}, t_2) & = & (\top, \max(t_1, ts_2))
\end{array}
$$

Figure 2.3: Shivers' control-flow analysis.

both arms of the conditional, using the timestamp resulting from analysing the first arm when we begin processing the other arm.

2.4 Serrano's 0CFA

Following up on Shivers' 0CFA work [Shi91], Serrano's CFA algorithm [Ser95] has the same runtime complexity and gets similar results, but uses a different method of analysis. While both analyses track all the values that flow to each variable and merge them into an abstract value, the Serrano algorithm takes multiple passes over the whole program and stops when there are no longer any additions made to any of the variables.

In the case of a call site, the Shivers algorithm either recursively begins evaluation of the function body again or not, based on whether that function body has been evaluated with respect to the current argument value bindings. The Serrano algorithm will not begin recursive re-evaluation of a function that is in the call chain of the current analysis. Instead, it adds the new argument values to the potential values for parameters to the called function and relies on the next pass over the whole program to evaluate the function in the widened context. The Serrano algorithm also does not keep around a timestamp or list of arguments seen before by a function and can perform analysis of a function using values already evaluated.[5] Details of the algorithm are in Section 2.4.1.

2.4.1 Implementation Details

The $\oplus$ operator performs identically to the basic version shown in Figure 2.2, determining a new combined abstract value. There is now a global reference cell, `changed`, that tracks whether there has been an update on this pass processing the program. We also no longer store a timestamp into the individual variables representing at what point during the analysis they were last updated. Unlike Shivers' algorithm, which is only called once, this algorithm starts with the `changed` variable set to `false` and is re-run until the `changed` flag is not set to `true` during a pass.

This implementation reuses the $\mathcal{R}$ function as defined in Section 2.3.1. The function $\mathcal{A}$, which maps from variables to abstract values, is nearly identical to that of the original naive control-flow analysis.

---

5. There is a note in the original presentation [Ser95] that a "stamp technique" can be used, but any timestamp similar to that of Shivers' original presentation does not require a fixpoint loop re-interpretation of the program. The implementation below reflects how several real implementations have interpreted the non-termination issue when implementing this algorithm.

$$
\begin{array}{rcl}
\mathbb{C}[\![\cdot]\!] & : & \mathit{Exp} \rightarrow \mathit{FunID}^* \rightarrow \mathit{AbsValue} \\
\mathbb{C}[\![x]\!]\mathrm{a} & = & \mathcal{A}[x] \\
\mathbb{C}[\![\text{fun } f(\vec{x}) = e_1 \text{ in } e_2]\!]\mathrm{a} & = & \mathcal{A}[f] := \mathcal{A}[f] \oplus \lambda(\vec{x})e_1; \mathbb{C}[\![e_2]\!]\mathrm{a} \\
\mathbb{C}[\![\text{let } x = e_1 \text{ in } e_2]\!]\mathrm{a} & = & \mathcal{A}[x] := \mathcal{A}[x] \oplus \mathbb{C}[\![e_1]\!]\mathrm{a}; \mathbb{C}[\![e_2]\!]\mathrm{a} \\
\mathbb{C}[\![\text{if } x \text{ then } e_1 \text{ else } e_2]\!]\mathrm{a} & = & \mathbb{C}[\![e_1]\!]\mathrm{a} \oplus \mathbb{C}[\![e_2]\!]\mathrm{a} \\
\mathbb{C}[\![f(\vec{x})]\!]\mathrm{a} & = & \displaystyle\bigoplus_{f' \in \mathcal{A}[f]} \begin{cases} \mathcal{R}[f'] & \text{when } f' \in \ \mathrm{a} \\ & \\ \mathcal{A}[y_i] := \mathcal{A}[y_i] \oplus \mathcal{A}[x_i] & \text{otherwise} \\ \quad \text{where } y_i \text{ is the } i\text{th param of } f' & \\ \quad \text{and } x_i \text{ is the } i\text{th arg to } f' & \\ \mathcal{R}[f'] := \mathbb{C}[\![f'.\mathrm{body}]\!]\mathrm{a} \cup \{f'\} & \\ \mathbb{C}[\![f'.\mathrm{body}]\!]\mathrm{a} \cup \{f'\} & \end{cases} \\
\mathbb{C}[\![\langle\vec{x}\rangle]\!]\mathrm{a} & = & \langle\vec{x}\rangle \\
\mathbb{C}[\![\#i(x)]\!]\mathrm{a} & = & \#i(\mathcal{A}[x]) \\
\mathbb{C}[\![b]\!]\mathrm{a} & = & b
\end{array}
$$

Figure 2.4: Serrano's control-flow analysis.

$\mathcal{A} : \mathit{Var} \rightarrow \mathit{AbsValue}$

The only difference in the $\mathcal{A}$ function is that now whenever an abstract value is changed in the store, the flag `changed` is set to `true`.

2.5 Environment Analysis Limitations

While control-flow analysis provides information about what function each variable can be bound to, these basic 0CFA implementations do *not* distinguish in their output information based on the environment in which the function was captured. In other words, instead of tracking the captured environment (closure) when a function is used as a first-class value, 0CFA just captures the function's name.

So, in the code below, 0CFA can tell us that at the point where we apply the result of `f(5)` to `2` we are invoking the function `g`, but 0CFA does not distinguish between a version of `g` with a binding of `outer` to `5` or any other value it might have gotten elsewhere in the program.

```
let fun f (outer) = let
```

```
          fun g(x) = outer + x
     in
          g
     end
in
  f(5)(2)
end
```

While this imprecision limits the set of optimizations we can perform — in general, we can not replace a call site directly with the function or functions we know will be called there — we can still use 0CFA information to perform many useful optimizations.

2.6 kCFA and ΓCFA

There are two basic approaches to improving precision of control-flow analysis. The kCFA framework, also presented in Shivers' Ph.D. dissertation [Shi91], shows how we can increase our precision by tracking the call stack up to depth k at each value binding site. So, rather than remembering that a function `f` can be called with either `false` or `true`, we would analyse `f` and remember a different binding environment for each of its call sites. This strategy corresponds to $k = 1$, or 1CFA. For larger values of k, we increase the stack depth. Unfortunately, even for $k = 1$, Van Horn and Mairson showed [VHM08] that deciding 1CFA is exponential. In practice, Might [MS06] showed that 1CFA was intractibly slow on even small programs.

Another approach that has been used is to increase the precision of the control-flow analysis by removing abstract bindings from a variable when those bindings are no longer in scope. This approach is called abstract garbage collection, and is incorporated into Might's ΓCFA framework [MS06]. During the control-flow analysis, as variables are bound they are also associated with a binding time. When an environment is captured in a closure, the binding time for each variable free in the function's body is then associated with that closure. At each step of the analysis, bindings are potentially created or removed from scope, so abstract garbage collection is used to remove any bindings that are no longer available.

In the example shown in Figure 2.6, once ΓCFA reached the first call to `id`, the analysis would bind `false` as a potential input and output for the function `id`. After the call completes, however, none of those bindings are available. At the time that the analysis reaches the second call site `id`, there is no longer any information associated with `id`, so the analysis is free to return `true` for the result of the program. Neither the Shivers nor the Serrano algorithms for 0CFA were able to determine that.

Luckily, we have not increased the complexity of the algorithm,[6] but have increased the precision of the output. This correlation between increased precision and frequently decreased runtime through environment inspection is examined in greater detail by Might [Mig07]. We provide an examples of both decreased and increased runtime and discuss the conditions under which collecting control-flow analysis performs more slowly in Chapter 4.

The abstract garbage collection process only makes sense in the context of an analysis that uses environment information as it examines control-flow. This requirement for environment information makes it directly applicable to the Shivers-style CFA, but it is unclear how abstract garbage collection would perform on a Serrano-style framework. In the default Serrano control-flow implementation, all environment information apart from the immediate lexical scope is ignored. Additionally, converging to a fixpoint by repeatedly running over the whole program graph means a complete restart each time in a straightforward application of abstract garbage collection.

2.7 Reference Counted CFA

Even if the intermediate representation contains sufficient environment information, garbage collectors can be a challenge to write and debug. Therefore, within Manticore,

6. Except for any costs associated with abstract garbage collection.

we have implemented a simpler approximation of ΓCFA. Instead of doing full abstract garbage collection, we add an abstract reference count to each of the variables. When a variable is allocated as a parameter or on the left-hand side of a `let` construct, the variable's reference count is set to 1. If the variable is rebound in a recursive call or captured by a closure, then the variable's reference count is set to ∞. When the variable goes out of scope,[7] we decrement the reference count associated with that value. If the reference count of the variable reaches zero, then we do not incorporate the value bound to that variable in any future portions of the analysis. By clearing the value associated with that variable, old values that were bound to the variable will not reduce the precision of any new abstract value assigned to that variable in the future.

Our algorithm is based on ideas from Hudak's abstract reference counting [Hud86], and was inspired by a comment in the Future Work section of Might's abstract garbage collection work [MS06]. Since the reference counts do not affect the convergance rate, we do not technically need to use Hudak's method of restricting the range of reference count values to a finite set, but if we do not restrict the range then we end up having to recreate the more involved binding environment of the ΓCFA analysis.

Since it is easy to confuse the use of the word *abstract* between Might and Hudak's work, abstract reference counting refers to counting concrete objects using abstracted counts (i.e. members of the set $\{0, 1, \infty\}$); abstract garbage collection refers to standard garbage collection over the abstracted object space that is used to track values taken on by variables during a control-flow analysis.[8]

Abstract referencing counting CFA is interesting because it adds to the precision of basic 0CFA implementations without requiring either a huge amount of additional implementation effort or runtime complexity. Implementing many of the other post-

7. Out of scope is determined by the `throw` or `apply` constructs, since we are in a CPS IR in Manticore. There is no function return.

8. This algorithm could therefore more properly be called abstract abstract-reference counted control-flow analysis.

0CFA proposals requires significantly more effort.

Section 2.7.1 illustrates the implementation of abstract reference counting.

2.7.1 Implementation Details

Abstract reference counts are implemented as a member of the set $\{0, 1, \infty\}$. If there is ever a second addition to a variable due to a recursive binding or closure capture, we never clear the variable's value. When we decrement a variable's reference count, if the new reference count is zero, then we clear out the locally cached data associated with the variable. We want to remember globally everything that has been stored into, for example, parameters to a function. For the purposes of control-flow analysis, we want to clear out the abstract values that we are propagating into other variables in the program, as a reference count of zero indicates that the current abstract value is no longer live in the program. The implementation defines the `inc` and `dec` functions as used in the implementation. The `dec` function will zero out the reference count and clear the stored abstract value in the case where the reference count has reached zero. The function $\mathcal{N}$ stores the reference count associated with a variable.
$\mathcal{N} : Var \rightarrow \{0, 1, \infty\}$

Reference counts are handled by incrementing the parameters upon entry to a function and decrementing them upon exit. The variable `l` contains the set of live variables bound within the context of the current function body. This variable is set to the parameters of the function upon entry and is used to decrement the value reference count associated with each variable when the function terminates. In a direct-style language like this one presented we use the entry and exit of the function body as points to reset and clear the bound variables. To capture function return in the direct-style language, we augment the end of the function body with a `free` statement. In the continuation-passing style intermediate representation in Manticore, we use the entry point and the function call at the end of the block instead.

When a closure is captured, we increment the reference count associated with each of

	Naive	Shivers	Serrano	RC
`id`	{`id`}	{`id`}	{`id`}	{`id`}
`x`	$\top$	$\top$	$\top$	$\top$
`t`	true	true	true	true
`f`	false	false	false	false
`i`	false	false	$\top$	false
`j`	true	$\top$	$\top$	true

Table 2.1: Comparison of accuracy of control-flow analysis algorithms

the free variables (determined below by the function `FV`). Incrementing the reference count on captured variables avoids clearing bindings that might still be accessible later in the program's execution.

This implementation reuses the $\mathcal{A}$ and $\mathcal{R}$ functions as defined in Section 2.3.1. The $\oplus$ operator and timestamping cutoff perform identically to that of Shivers' 0CFA implementation in Figure 2.3. The implementation is defined in Figure 2.5.

2.8 Analysis Comparison

The differences in precision and execution of the Shivers, Serrano, and reference counting control-flow analyses are clear in an example. In the rest of this section, we run and observe the output from the algorithms on the program in Figure 2.6. We assume that the abstract value set has been augmented with the values of `true` and `false`. Boolean values are only used for tracking purposes and ease of reading the program, not to track conditional control flow.

This program is interesting to control flow analysis because it uses the `identity` function twice. Even with such a simple program, we can display the difference in precision and behavior of all of the control-flow analyses presented in this chapter.

The output of Shivers' 0CFA algorithm on Figure 2.6 is in Table 2.1. It is particularly worth noticing that the variable `i` has a value of `false` but `j` has a value of $\top$. This results because when the algorithm first encounters the function `id`, the valid

$$
\begin{array}{rcl}
\mathbb{C}[\![\cdot]\!] & : & \mathit{Exp} \rightarrow \mathit{FunID}^{*} \rightarrow \mathit{Integer} \rightarrow \mathit{AbsValue} \\[1em]
\mathbb{C}[\![x]\!]\mathrm{l}\ \mathrm{t} & = & \forall f' \in \mathcal{A}[x](\forall x' \in \mathrm{FV}(f')(\mathrm{inc}\ x')) \\
 & & \mathcal{A}[x] \\[1em]
\mathbb{C}[\![\mathrm{fun}\ f(\vec{x}) = e_1\ \mathrm{in}\ e_2]\!]\mathrm{l}\ \mathrm{t} & = & \mathcal{A}[f] := \mathcal{A}[f] \oplus \{\lambda(\vec{x}).\mathrm{let}\ x' = e_1\ \mathrm{in\ free}\ x', \mathrm{t}\} \\
 & & \quad \text{where } x' \text{ fresh} \\
 & & \mathbb{C}[\![e_2]\!]\mathrm{l}\ \mathrm{t}' \\
 & & \quad \text{where } \mathrm{t}' = \ \mathrm{t}\ + 1 \text{ if } \mathcal{A}[f] \text{ changed else t} \\[1em]
\mathbb{C}[\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2]\!]\mathrm{l}\ \mathrm{t} & = & \mathcal{A}[x] := \mathcal{A}[x] \oplus \{\mathbb{C}[\![e_1]\!]\mathrm{l}\ \mathrm{t}, \mathrm{t}\} \\
 & & \mathrm{inc}\ x \\
 & & \mathbb{C}[\![e_2]\!]\mathrm{l} \cup x\ \mathrm{t}' \\
 & & \quad \text{where } \mathrm{t}' = \ \mathrm{t}\ + 1 \text{ if } \mathcal{A}[x] \text{ changed else t} \\[1em]
\mathbb{C}[\![f(\vec{x})]\!]\mathrm{l}\ \mathrm{t} & = & \displaystyle\bigoplus_{f' \in \mathcal{A}[f]} \begin{cases}
\mathcal{R}[f'] & \text{when } \mathcal{A}[f \\
 & \text{newest} \\
\mathrm{l}' = \vec{y} & \text{otherwise} \\
\quad \text{where } \vec{y} \text{ are the parameters of} f' & \\
\forall x' \in \mathrm{l}'(\ \mathrm{inc}\ x') & \\
\forall x_i \text{ in } \vec{x} & \\
\quad \mathcal{A}[y_i] := \mathcal{A}[y_i] \oplus \{\mathcal{A}[x_i], \mathrm{t}\} & \\
\qquad \text{where } y_i \text{ is the } i\text{th parameter of } f' & \\
\quad \mathrm{t} = \ \mathrm{t}\ + 1 \text{ if } \mathcal{A}[y_i] \text{ changed else t} & \\
\mathcal{R}[f'] := \mathbb{C}[\![f'.\mathrm{body}]\!]\mathrm{l}'\ \mathrm{t} & \\
\mathbb{C}[\![f'.\mathrm{body}]\!]\mathrm{l}'\ \mathrm{t} &
\end{cases} \\[1em]
\mathbb{C}[\![\mathrm{if}\ x\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2]\!]\mathrm{l}\ \mathrm{t} & = & \mathbb{C}[\![e_1]\!]\mathrm{l}\ \mathrm{t} \oplus \mathbb{C}[\![e_2]\!]\mathrm{l}\ \mathrm{t}' \\
\mathbb{C}[\![\langle\vec{x}\rangle]\!]\mathrm{l}\ \mathrm{t} & = & \langle\vec{x}\rangle \\
\mathbb{C}[\![\#i(x)]\!]\mathrm{l}\ \mathrm{t} & = & \#i(\mathcal{A}[x]) \\
\mathbb{C}[\![b]\!]\mathrm{l}\ \mathrm{t} & = & b \\
\mathbb{C}[\![\mathrm{free}\ x]\!]\mathrm{l}\ \mathrm{t} & = & (\forall x' \in l\ (\mathrm{dec}\ x')); \mathcal{A}[x] \\[1em]
\mathrm{inc}\ x & = & \begin{cases} \mathcal{N}[x] := 1 & \text{when } \mathcal{N}[x] = 0 \\ \mathcal{N}[x] := \infty & \text{otherwise} \end{cases} \\
\mathrm{dec}\ x & = & \begin{cases} \mathcal{N}[x] := \infty & \text{when } \mathcal{N}[x] = \infty \\ \mathcal{A}[x] := \bot; \mathcal{N}[x] := 0 & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2.5: Reference counted control-flow analysis.

```
let
  fun id x = x
  val i = id false
  val j = id true
in
  j
end
```

Figure 2.6: Simple identity example program.

return values are only `false`. After the second call, though, 0CFA has no way of distinguishing the call site and must conservatively assume that anything seen so far in the analysis could be returned from `id`.

Notice the difference in the Serrano algorithm from Shivers' algorithm's results in Table 2.1. The variable `i` now has a value of $\top$, instead of `false`. This imprecision is because of the repeated iteration of the algorithm over the whole program. The second time we analyse the binding to the variable `i`, we have seen the function `id` return both `true` and `false`. Even though it is impossible for the `id` function to return a `true` value this early in the program, because of the way the analysis is executed, we lose precision with respect to the call to `id` whose result is bound to the variable `i`.

Finally, the results of abstract reference counting-enabled control-flow analysis are compared in Table 2.1. We are now able to tell that the variable `j` is assigned the value of `true` because after the call to `id` completes and binds the value `false` to `i`, the reference count on the local variable `x` is set to zero, clearing the way to remember only the *true* result, `true`.

2.9 Related work

Early work in control-flow analysis focused on finding linear time approximations to the quadratic runtime of straightforward control-flow analysis on higher-order languages. Heintze and McAllester formulated an approach called linear-time subtransitive control-flow analysis (LT-ST-CFA) [HM97]. The key idea of LT-ST-CFA is that since many compiler optimizations only need local information, a linear approximation of restricted queries should be possible. The list of functions invoked at an individual call site and identification of single-use functions are two such restricted queries. A *bounded type* is one in which the size of the polymorphic type is bounded by the same constant of the monomorphic types of the program. By using the bounded types in their control-flow analysis technique to limit execution time, Heintze and McAllester hypothesized that they could achieve linear time query answers. Unfortunately, as Saha, Heintze, and Oliva later showed, most ML programs of any size have unbounded types and without significant heuristic tuning, LT-ST-CFA is not practical [SHO98].

Ashley and Dybvig demonstrated a control-flow analysis they titled sub-0CFA [AD98]. They use small constant parameters to fix the size of the abstract value domain so that the analysis stays linear-time. Tied to each program point is a counter indicating the number of times that the value for that point has been updated. Once the counter hits a constant, the value associated with the program point becomes $\top$. This limitation of the counter bounds the analysis to a constant number of passes over the program. It provides even lower precision than 0CFA, but does provide speedup for some cases.

Jagannathan and Wright introduced a control-flow analysis based on 0CFA and polymorphic splitting of binding sites [JW95]. Basically, they index their variable bindings based on the type of the binding that is in the scope of the analysis. This strategy based on type does not exhibit the same blow-up in runtime in practice that 1CFA does, even though polymorphic splitting does require retaining information in closures about the types of the captured variables. This analysis increases the precision of 0CFA without dramatically increasing analysis time, but polymorphic splitting is a more expensive method of performing the reachability analysis than ΓCFA. The most interesting part of this work, though, is a detailed comparison of the performance of run-time check removal when implemented atop 0CFA, a type inference algorithm, 1CFA, and their polymorphic splitting. Their results show that most of the time

0CFA does better than type inference, and that 1CFA and polymorphic splitting are even better.

Reppy provided an extension to control-flow analysis that handles escaping values from abstract types [Rep06]. Since any abstract type is known within the module that it escapes from, control-flow analysis can provide an approximation of the concrete types that can be passed in where that abstract type is expected. This additional knowledge allows optimization based on not only the concrete type, but also the number of value allocations performed.

Midtgaard and Jensen recently extended a direct-style 0CFA analysis based on abstract interpretation in a stack machine context to track function return points as well as call sites [MJ09]. Since the return location in a direct-style intermediate representation is the same as the return continuation in a CPS intermediate representation, they proved that there exists a correspondance between the two. Their work ignores environment information and promotes value abstraction in the same manner as 0CFA, so it would be possible to apply abstract reference counting in their context and possibly increase the precision of their results.

2.10 Types and control-flow analysis

If control-flow analysis provides so much information, why is it not more widely used? Many compilers switched over to type-driven analyses after Palsberg and O'Keefe showed a connection between type systems and flow analyses in the context of safety analysis [PO95]. Since there is a close relationship between the information gathered during type inference and control-flow analysis, it is often a short step to take an algorithm that worked on control-flow information and make it instead work on type information. Additionally, compilers at this time were making the shift from dynamically typed languages like scheme to statically typed languages like ML. Therefore, types were already associated with values in the program, and even though the types typically provide no more information than basic control-flow analyses, since the type information is good enough to perform most first-order optimizations, there was little motivation to perform analyses with quadratic worst-case performance.

As we show in the next chapter, though, some of the work done by the type system to turn values into sets with equivalent types loses the distinction between the individual elements that is present in the control-flow analysis. In particular, we look at arity raising in a higher-order context and show how control-flow analysis is able to do a much better job than a type-directed version, because of the presence of more precise information about function call sites.

CHAPTER 3

ARITY RAISING

Arity is the number of arguments that a function accepts. The arity raising transformation takes a function of n arguments and turns it into a function of $>= n$ arguments. By increasing the number of arguments to a function, we increase the opportunity for the compiler to store values associated with those arguments in registers instead of in heap-allocated data. Reducing the amount of heap-allocated data both reduces pressure on the garbage collector and removes overhead associated with writing and reading data in memory.

There are two major sources of extra memory allocations that we focus on removing.

1. Raw data, such as integers and floating-point numbers, stored in a heap objects
2. Datatypes and tuples, which package up a set of data into a single structure in memory

Both of these sources of memory allocations and memory access have been shown to be very expensive by Tarditi and Diwan [TD94]. In fact, the overhead associated with reading and writing a uniform representation and extra checks to see if the garbage collector needs to run often cost more than the garbage collection process itself. In their work using a simulator to collect instruction counts, they showed that 19-46% of the execution time of a program in Standard ML of New Jersey was spent in tasks related to storage management.

The first source of extra memory allocations is commonly known as *boxing*. By storing raw data into heap objects, the rest of the system does not need to worry about the format of the raw object. The garbage collector treats all values in registers and the stack as pointers and can trace them uniformly. Polymorphic functions operate on values of any type without taking special action based on the underlying object type. But this uniform treatment comes at a cost — allocating and accessing raw data in the heap can be very expensive, especially for small and frequently updated data. Our implementation of arity raising determines where it is safe to pass the raw object value instead and removes the creation of the box object.

The second source of memory allocations is tuples and datatypes. If the user has created a very deeply nested set of datatype definitions or tuples but functions commonly only need few pieces of data deep within that datatype, it can be expensive to create and traverse the whole structure just to handle those few pieces of data. Our implementation of arity raising determines when only a few pieces of a datatype are being used and allocates and passes just those pieces, rather than the entire structure.

This paper describes a strategy for arity raising that allows the compiler to safely increase the number of parameters to a function and remove allocations due to both boxing operations and data structures. This strategy is *conservative* — it will not change the program in a way that could degrade the performance by introducing extra operations. We restrict ourselves to transforming expressions along a code path without branches. Those transformations move expressions and eliminate matching allocation and selection pairs.

After presenting some preliminary notation we use in our arity raising strategy, in Section 3.2 we describe the analysis of function bodies. This analysis provides information on when it is useful to transform data stored in heap objects into directly passed parameters. In Section 3.3, we show how to use the gathered information to transform function definitions and call sites. Following an example of the analysis and transformation, we discuss implementation details of this arity raising strategy within the Manticore compiler. We cover the substantial related work, and performance measurements are presented later in Section 4.2.

3.1 Preliminaries

We use the direct style intermediate representation in Figure 2.1 for this presentation. We assume that all bound variables are unique and that associated with each application call site is a *program point*, labeled with a superscript l, that is a unique label for the expression. Booleans, tuples, and functions are the only values that variables can take on in this language. Integers may only be used in selections.

We assume the presence of the below maps from a control-flow analysis to build this graph. Our implementation uses a control-flow analysis similar to that presented by Serrano [Ser95], which provides sufficient information to implement these maps. We assume the following maps are provided by the control-flow analysis.

$\mathcal{F} : \text{FunID} \rightarrow 2^{\mathbb{L}}$

$\mathcal{C} : \mathbb{L} \rightarrow 2^{\text{FunID}}$

$\mathcal{A} : \text{FunID} \rightarrow 2^{\text{FunID}}$

$\mathcal{F}$ maps function identifiers to either a list of program points or $\emptyset$ for unknown. A function *escapes* if it has any potentially unknown call sites and $\mathcal{F}$ maps those functions to $\emptyset$. We cannot safely perform a translation on any functions with unknown call sites.

$\mathcal{C}$ lists the set of functions that can be called at a given program point or $\emptyset$ if the set is unknown. A call site with unknown target functions can not be transformed.

$\mathcal{A}$ maps a function to the set of all the functions that could potentially share call sites with it. This map can be computed from the $\mathcal{F}$ and $\mathcal{C}$ maps provided by control-flow analysis.

The use count of a variable is the number of times that the variable occurs in any position other than its binding occurrence. The map $\mathcal{U}$ provides the use count of a variable.

$\mathcal{U} : \text{VarID} \rightarrow \mathbb{N}$

3.2 Analysis

The analysis phase of this optimization contains almost all of the complexity. Control-flow analysis is run over the whole program before we begin execution. Any function with unknown call sites is ignored. For all functions with only known call sites, we gather information from the body of the function and then compute a signature based on whether or not call sites are shared with other functions.

3.2.1 Gathering Information

An *access path* is a series of tuple selection operations performed on a parameter. Access paths are zero-based and the selections occur in left-to-right order. The access path 0.1.2 means to take the first parameter to the function, select the second item from it, and then select the third item from that. The *variable map* $\mathcal{V}$ maps a variable to an access path.
$\mathcal{V}$: var $\rightarrow$ path
The notation $\mathcal{V}_f$ refers to the map from variables to access paths restricted to those variables defined within the function `f`. Variables are assumed to be unique.

The *path map* maps an *access path* to a count of the number of times that path is directly used. The path map is specific to an individual function, as access paths are relative to the parameters of the function and have a different meaning within different scopes. The path map is equal to the use count of the variable associated with that path minus any uses of that variable as the target of a selection.
$\mathcal{P}_f$: path $\rightarrow$ int

Consider the following intermediate representation fragment:

```
fun f(x) =
  let a = #0(x)
  let b = #1(a)
  in b
```

...

The fragment for the function `f` above has the following *variable map*, indicating that `x` is the first parameter, `a` is the first slot of the first parameter and that `b` is the second slot of the first slot of the first parameter:
$\mathcal{V} = \{\mathtt{x} \mapsto 0, \mathtt{a} \mapsto 0.0, \mathtt{b} \mapsto 0.0.1\}$

The fragment for the function `f` has the following *path map*, indicating that only the variable `b` is used outside of tuple selection expressions.
$\mathcal{P}_f = \{0 \mapsto 0, 0.0 \mapsto 0, 0.0.1 \mapsto 1\}$

The map $\mathcal{V}$ is filled in by the algorithm $\mathbb{V}$ in Figure 3.1. The map $\mathcal{P}$ is defined directly. Where a more specific case appears earlier in the algorithm, that case is to be run in place of the more general one later. The most important two cases are function definition and variable binding where the right hand side is a selection. The operation $\prec$ is a binary operator that is true if the first access path is a prefix of the second. For example, the access path 0.1 is a prefix of 0.1.3 but is not a prefix of 0.2.

$$
\begin{array}{rcll}
\mathbb{V}[\![]\!] & : & \mathit{Exp} \to \mathit{Unit} & \\
\mathbb{V}[\![\text{fun } f(\vec{x}) = e_1 \text{ in } e_2]\!] & = & \forall x_i \in \vec{x}\ (\mathcal{V}(x_i) := i); \mathbb{V}[\![e_1]\!]\ ;\ \mathbb{V}[\![e_2]\!] & \\
\mathbb{V}[\![\text{let } x = \#i(y) \text{ in } e_2]\!] & = & \left\{ \begin{array}{l} \mathcal{V}(x) := \mathcal{V}(y).i; \mathbb{V}[\![e_2]\!] \\ \mathbb{V}[\![e_2]\!] \end{array} \right. & \begin{array}{l} \text{when } \mathcal{V}(y) \neq \emptyset \\ \text{otherwise} \end{array} \\
\mathbb{V}[\![\text{let } x = e_1 \text{ in } e_2]\!] & = & \mathbb{V}[\![e_1]\!]\ ;\ \mathbb{V}[\![e_2]\!] & \\
\mathbb{V}[\![\text{if } x \text{ then } e_1 \text{ else } e_2]\!] & = & \mathbb{V}[\![e_1]\!]\ ;\ \mathbb{V}[\![e_2]\!] & \\
\mathbb{V}[\![e]\!] & = & () &
\end{array}
$$

$$
\mathcal{P}_f(p) = \sum_{x \mid \mathcal{V}_f(x) = p} \Big(\mathcal{U}(x) - \big| \{ y \mid x \prec y \text{ and } \mathcal{V}(y) \neq \emptyset \} \big| \Big)
$$

Figure 3.1: Algorithm to compute variable and path maps.

Consider the algorithm $\mathbb{V}$ applied to the example function `f` at the beginning of this section. The maps $\mathcal{V}$ and $\mathcal{P}_f$ are empty. Analysing the function binding, we add all of the parameters to the map $\mathcal{V}$, binding them to their corresponding index. The function binding for `f` defines a single parameter, `x`, so the variable map is set to $\{\mathtt{x} \mapsto 0\}$. At each local variable binding whose right hand side is a selection, the path represented by that selection statement and base variable is entered in the map $\mathcal{V}$ as corresponding to that variable. After processing the two `let` bindings within the body of `f`, the variable map $\mathcal{V} = \{\mathtt{x} \mapsto 0, \mathtt{a} \mapsto 0.0, \mathtt{b} \mapsto 0.0.1\}$. The map $\mathcal{P}_f$ is now valid on those three paths, returning the path map described earlier.

3.2.2 Computing Signatures

Given the maps $\mathcal{V}$ and $\mathcal{P}$, we can compute an individual function's ideal arity-raised signature and final arity-raised signature. A function's ideal signature is the signature that promotes the variables corresponding to selection paths that are used in the function's body up to parameters — but only if another parameter is not a prefix of the proposed new parameter. This ideal signature is a set of of selection paths. A function's final signature is a list of access paths, sorted in lexical order. The final signature of a function also differs from the ideal signature in that it is the same as all other functions that it shares a signature with.

The ideal signature reduces the set of selection paths because if one variable's path is a prefix of another variable's path, the variable that is a prefix will already require the caller to do an allocation of all of the intermediate data. For example, in the function `usesTwo` below, it may be worth promoting the variable `first` to a parameter, but we will not also promote the variable `deeper` to a parameter. Promoting `deeper` will not open up any opportunities to remove allocated data, but will introduce more register pressure. There is a possibility that we could avoid a memory fetch if there was a spare register and we could directly pass `deeper` instead of performing a selection from `first`, but since our algorithm is conservative and aggressive promotion results in huge numbers of parameters in practice, we will not promote variables like `deeper`.

```
fun usesTwo (param) =
  let first = #1(param)
  let deeper = #2(first)
```

```
  in otherFun (first, deeper)
```

The ideal signature for a function `f` is denoted by σ_f and computed as follows:
$\rho_f = \{\ p \in \text{rng}(\mathcal{V}_f)\ \wedge\ \mathcal{P}_f(p) > 0\}$
$\sigma_f = \{\ p \mid p \in \rho_f \wedge (\nexists q \in \rho_f)(q \prec p)\}$
The first set, ρ_f, is the list of all of the access paths corresponding to variables in the function `f` with non-zero use counts after substracting their uses in tuple selections. The ideal signature is computed by selecting all of the paths that do not have a prefix in ρ_f.

The map $\mathcal{S}$ is from a set of function identifiers to either a new signature or $\emptyset$, indicating that the function will not have its parameter list or any passed arguments transformed.
$\mathcal{S} : 2^{\text{FunID}} \rightarrow \text{signature}$

We build up the map $\mathcal{S}$ by using the $\mathcal{A}$ map provided by control-flow analysis to determine the set of all functions that share call sites and computing the safe merger of their ideal signatures. The safe merger of two ideal signatures is defined by the binary operator $\uplus$ below. This operator creates a set consisting of the shortest prefix paths between the two signatures.
$\sigma_1 \uplus \sigma_2 = \{\ p \mid p \in \sigma_1 \wedge (\nexists q \in \sigma_2)(q \preceq p)\} \cup \{\ p \mid p \in \sigma_2 \wedge (\nexists q \in \sigma_1)(q \preceq p)\}$

Since the intermediate representation used in this presentation has no type information available, we need to be conservative with our path selections. For any path that is in one signature to be safe, it needs to be a prefix of or equal to a path in the other signature. If either of the sets σ_1' or σ_2' below are non-empty, we cannot compute a common signature for this pair of functions using this algorithm.[1] In that case, the map $\mathcal{S}$ will instead return a signature corresponding to the default calling convention.

---

1. See the implementation notes in Section 3.5 for how we avoid this limitation in Manticore

$$\sigma_1' = \{\ p \mid p \in \sigma_1 \wedge (\nexists q \in \sigma_2)(p \preceq q \vee q \preceq p)\}$$
$$\sigma_2' = \{\ p \mid p \in \sigma_2 \wedge (\nexists q \in \sigma_1)(p \preceq q \vee q \preceq p)\}$$

3.3 Transformation

Each new function signature requires the code to be transformed in three places. Figure 3.2 shows the tranformation process on this intermediate representation via the transformation $\mathbb{T}$.

For each function that is a candidate for arity raising, we transform the parameter list of the function definition to reflect its new signature. That new signature is made up of the variables corresponding to the paths that are part of the final signature in $\mathcal{S}$. The parameters are ordered by the lexical order of the paths as returned by $\mathcal{S}$.

The parameter to the transformation `ys` is the set of variables that have been lifted to parameters of functions. We add variables to this set at any function definition where we add a variable to the parameter list. When we encounter a variable binding for a member of the set `ys`, we skip that binding since the variable is already in scope at the parameter binding.

At each location where the function is called, we replace the call's argument list with a new set of arguments selected from the original ones based on the new signature. There is one procedure not defined: in the case of a call to a function that is being arity raised, we construct a series of `let` bindings for the new arguments based on the final signature of the functions sharing that call site, represented by the variable `sels`.

For example, if the function `f` has an entry in the map $\mathcal{S}$ with a value of $[0.0, 0.1.0]$, then a call to the function `f` will be transformed from

```
f(arg)
```

into

```
let a1 = #0(arg)
let t1 = #1(arg)
let a2 = #0(t1)
in f(a1, a2)
```

Transformation of the code is performed in a single pass over the intermediate representation.

$$
\begin{array}{rcll}
\mathbb{T}[\![]\!] & : & (Exp \times Vars) \rightarrow Exp & \\
\mathbb{T}[\![\text{fun } f(\vec{x}) = e_1 \text{ in } e_2]\!]\text{ys} & = & \begin{cases} \text{fun } f(\vec{x}) = \mathbb{T}[\![e_1]\!]\text{ys} & \\ \qquad \text{in } \mathbb{T}[\![e_2]\!]\text{ys} & \text{when } \mathcal{S}(f) \ = \ \emptyset \\ & \\ \text{fun } f(\vec{z}) = \mathbb{T}[\![e_1]\!]\vec{z} \cup \text{ys} & \\ \qquad \text{in } \mathbb{T}[\![e_2]\!]\text{ys} & \text{where } \vec{z} = \{\mathtt{z} | (\exists p)(p \in \mathcal{S}(f) \wedge \mathcal{V}(\end{cases} \\
\mathbb{T}[\![\text{let } x = e_1 \text{ in } e_2]\!]\text{ys} & = & \begin{cases} \mathbb{T}[\![e_2]\!]\text{ys} & \text{when } x \in \text{ys} \\ & \\ \text{let } x = \mathbb{T}[\![e_1]\!]\text{ys in } \mathbb{T}[\![e_2]\!]\text{ys} & \text{otherwise} \end{cases} \\
\mathbb{T}[\![\text{if } x \text{ then } e_1 \text{ else } e_2]\!]\text{ys} & = & \text{if } x \text{ then } \mathbb{T}[\![e_1]\!]\text{ys else } \mathbb{T}[\![e_2]\!]\text{ys} & \\
\mathbb{T}[\![f^l(\vec{x})]\!]\text{ys} & = & \begin{cases} f^l(\vec{x}) & \text{when } \mathcal{C}(l) = \emptyset \text{ or } \mathcal{S}(\mathcal{C}(l)) = \emptyset \\ & \\ \text{let } \mathtt{new} \ = \ \mathtt{sels} \text{ in} f(\mathtt{new}) & \text{where } \mathtt{sels} \text{ is the } \mathcal{S}(\mathcal{C}(l)) \text{ paths} \end{cases} \\
\mathbb{T}[\![\langle\vec{x}\rangle]\!]\text{ys} & = & \langle\vec{x}\rangle & \\
\mathbb{T}[\![\#i(x)]\!]\text{ys} & = & \#i(x) & \\
\mathbb{T}[\![x]\!]\text{ys} & = & x & \\
\mathbb{T}[\![b]\!]\text{ys} & = & b &
\end{array}
$$

Figure 3.2: Algorithm to arity raise functions.

3.4 An Example

To better understand the intermediate representation, what the optimization looks at and attempts to remove, and what the desired generated code looks like, we present an example that exhibits both of the types of memory allocations listed in the introduction. Raw floating point numbers are boxed and there is a user-defined type. This code defines an ML function that takes a pair of parameters — a datatype with two reals, and another real. The function then extracts the first item from the datatype and adds it to the second parameter. The second member of the datatype is unused.

```
datatype dims = DIM of real * real;
fun f(DIM(x, _), b) = x+b;
f (DIM(2.0, 3.0), 4.0)
```

This code transforms into the following intermediate representation, as presented in Figure 2.1 but augmented with reals and the addition operator. Temporary variables have been given meaningful names in the example to aid understanding.

```
fun f(params) =
  let dims = #0(params)
  let fourB = #1(params)
  let four = #0(fourB)
  let twoB = #0(dims)
  let two = #0(twoB)
  let six = two+four
  in <six>
let twoB = <2.0>
let threeB = <3.0>
let fourB = <4.0>
let dims = <twoB, threeB>
let args = <dims, fourB>
in f (args)
```

It is clear that this transformed code is not what we should generate real code from. Notice that allocations are used to box raw values, to allocate tuples, and to allocate datatypes. This similarity is exactly how it works within the intermediate representation of Manticore, and that similarity in allocation and access patterns allows our arity raising algorithm to treat them uniformly and avoids what might otherwise be a large increase in code complexity. Even though boxing of types, tuples, and datatype definitions will ultimately have different output from the code generator, uniform treatment in the intermediate representation enables optimizations in arity raising and elsewhere in the compiler.

The function `f` above has the following *variable map*:
$\mathcal{V} = \{\texttt{params} \mapsto 0, \texttt{dims} \mapsto 0.0,\ \texttt{fourB} \mapsto 0.1,\ \texttt{four} \mapsto 0.1.0,\ \texttt{twoB} \mapsto 0.0.0,\ \texttt{two} \mapsto 0.0.0.0\}$

The function `f` has a *path map* as follows.
$\mathcal{P}_f = \{0 \mapsto 0,\ 0.0 \mapsto 0,\ 0.1 \mapsto 0,\ 0.1.0 \mapsto 1,\ 0.0.0 \mapsto 0,\ 0.0.0.0 \mapsto 1\}$

Since there is only one function and its call site is immediate, the control-flow analysis information is not too interesting. The ideal signature for this function is: $[0.1.0, 0.0.0.0]$

After running the transformation $\mathbb{T}$, the code is now:

```
fun f(two, four) =
  let val six = two+four
  in <six>
let twoB = <2.0>
let threeB = <3.0>
let fourB = <4.0>
let dims = <twoB, threeB>
let args = <dims, fourB>
let dims' = #0(args)
let fourB = #1(args)
let four = #0(fourB)
let twoB = #0(dims')
let two = #0(twoB)
in f (two, four)
```

After Manticore's standard local cleanup phase to remove redundant allocation and selection pairs and unused variables, we have the following intermediate code:

```
fun f(two, four) =
   let six = two+four
   in <six>
f(2.0, 4.0)
```

3.5 Implementation

Manticore [FRR+07] is a compiler for a parallel programming language based on Standard ML. Manticore has a weakly typed intermediate representation. After types are inferred on the original source program, we both preserve and check them through each transformation in our intermediate representation. Monomorphic types are preserved exactly, but polymorphic types are weakened to an unknown type. This type information is sufficient to provide a better solution to the *incompatible paths* problem mentioned during Section 3.2. In Manticore, instead of checking selection paths against the other functions we are merging signatures with, we check the selection path against the type of the argument provided to the function. Since the types are frequently wider than the usage pattern, it is far more likely that functions that share a call site will be able to share an arity raised signature. Rather than requiring strict type equality between signatures, we require runtime representation equality. For example the functions `takesRaw` and `takesInts` below do not have the same runtime representation of their argument, but `takesInts` and `takesFloats` do and can safely share a call site.

```
val takesRaw = fn : real -> unit
val takesInts = fn : (int * int) -> unit
val takesFloats = fn : (float * float) -> unit
```

We also perform trivial flow analysis on conditionals within Manticore. If a conditional statement is a direct check against a property of a selection from a parameter path, then we do not permit any paths derived from it to be added into the maps, but we do allow analysis to continue within the arms of the conditionals.

3.6 Related Work

Optimizations to reduce the amount of overhead introduced by the language or execution model abound. Boxing optimizations change programs to deal with raw values directly instead of either storing them in an altered format or in a heap-allocated structure, introducing coercions between a boxed and unboxed format and increasing the amount of knowledge the generated code has about the specific type of the values in the program.

Datatype flattening reduces the overhead introduced by structuring raw data into heap-allocated objects. In cases where a set of values is placed into an object in the heap just to be passed to a method and subsequently pulled back out into their raw forms, avoiding the intermediate allocation saves a significant amount of overhead.

The related work over the last twenty years has mostly used either type or control-flow to drive their optimizations and most has addressed either the problem of optimizing boxing or flattening datatypes. Our presented work is unique because it uses both type and control-flow information and, by treating boxes and datatypes identically, both flattens datatypes and optimizes boxing. Our work also looks at data usage patterns within called functions, which has to this point been ignored.

3.6.1 Boxing Optimizations

One of the earliest pieces of formal work on the correctness of a system that handles boxed and unboxed versions of raw data types in the same program was done by Leroy [Ler92]. He introduced operators for boxing and unboxing and extended the type system to handle either boxed values or unboxed values. He then showed how to construct a version of the program that has changed all monomorphic functions — the functions where the raw type is known — to use unboxed values. Calls to his `box` and `unbox` operations (called `wrap` and `unwrap` in this paper) are introduced around polymorphic functions, as anywhere that the type is unknown the value must be in the uniform, boxed representation. Leroy then showed that the version of the program that purely used boxed types computes the same thing as the version of the program that uses a mixed representation. This strategy of mixed representations driven purely by the type system was then directly implemented in their compiler.

Complimentary work by Peyton-Jones et al. lifted `box` and `unbox` operations into the source language (Haskell) as well as the intermediate representation [PJL91]. They showed a signficiant number of transformations that can be performed in an ad-hoc manner within the compiler once the boxing information is available — not only canceling matched pairs of coercions, but also avoiding repeated coercion of the same value. Since they were working with a lazy language, they also provided several valuable insights into the interaction of strictness analysis and unboxed types. Most importantly, whenever an argument is strict (always going to be evaluated), it is safe to change that from a boxed to an unboxed argument, as it will be available at the time of the call. If the argument were not strict, then we would need to instead have an unboxed slot so that we could hold the code that will lazily produce the value instead.

Henglein also made all of the boxing and unboxing operations explicit in the intermediate representation of his program [HJr94]. He then provides a set of reduction rules that move coercions to places that are considered more formally optimal by his framework. By moving coercions until `box` and `unbox` operations are adjacent, it is possible to cancel out the pair of coercions. Depending on the order of the cancellations, this order choice corresponds to either keeping the data in its raw unboxed form or leaving the data in boxed form. Keeping data in raw form is good for monomorphic function calls, which can take arguments in raw form. Keeping data in boxed form is good for polymorphic function calls in this framework, as polymorphic calls required raw types in boxed form in order to dispatch properly. This work did not address which strategy was preferred, nor did this work provide an implementation or benchmarks. Unfortunately, this notion of optimality is based on the static number of coercions in the program, and even the decisions of whether to optimize first for raw form arguments or first for boxed form arguments is based on static determination of the number of polymorphic versus monomorphic functions in the program. Dynamic execution behavior is not considered in this framework.

A few years later, Thiemann revisited the theoretical work done by Henglein and provided a deterministic set of reduction relations for determining coercion placement [Thi95]. In particular, he chose a strategy of attempting to push unboxings toward calls in tail position. Since his work is primarily on the intermediate representation, this strategy ensured that there would be a register available to hold the value and that it would not have to be spilled (and thus boxed).

Ignoring type information, work by Goubault performs intraprocedural data-flow analysis to cancel nearby `box` and `unbox` pairs [Gou94]. While this strategy seems like it would intuitively be much worse than the previous whole program approach, Goubault also introduced a method called *partial inlining.* This method takes the bit of wrapper code that includes the `box` or `unbox` operations, which occur at the start of the called function and moves them into the caller. By moving those operations out of the called function, if there was an operation that can now be cancelled in the calling function, this allows that pair of operations to be canceled. While this strategy was not implemented, this work is important because it pushed the idea of splitting out the prologue and epilogue of a function and inlining them at the call sites.

Serrano uses a control-flow analysis approach based on 0CFA to gather information about data values and where they are used [SF96]. Where functions are called monomorphically, he specializes the functions to use raw types. In practice on a wide set of examples, he saw significant speedups and complete removal of boxing, in many cases. This optimization worked very well for the untyped scheme language he was compiling and produced results similar to those reported for type-directed approaches.

Also ignoring type information is the work on the placement of `box` and `unbox` operations is work by Faxén [Fax02]. His work performs whole-program control-flow analysis (CFA) and does the usual cancelling of matched coercion operations. He also uses the CFA information to identify potential sensitive locations in the program (like loops) where moving a `box` or `unbox` operation that was not inside of the sensitive area into that area could cause a major change in performance. By respecting the control-flow of the program, his implementation did not exhibit some of the significant reductions in performance that were shown by the previous works on coercion placement when the benchmark included a mix of polymorphic and monomorphic functions.

Attempting to tie up the argument between the type directed and control-flow directed work is a paper comparing practical results in the implementation of the Objective Caml compiler by Leroy[Ler97]. He provided performance data showing that a combined strategy, using the type information in restricted area based on a control-flow analysis (provided by an analysis similar to sub-0CFA with $n = 1$[AD98]) provided the best results. Relying exclusively on type information lead to extremes in either moving or not moving the coercion operators and results in performance that is worse than doing nothing at all on certain benchmarks. By preserving types during compilation, they could be used when the coercion optimization were going to be applied within functions.

The TIL compiler by Tarditi and Morrisett used a combination of explicit representation of coercions, type dispatch in polymorphic functions, and compiler optimizaitons to remove boxing [TMC$^+$96]. The compiler keeps around type information throughout the compilation process. After inlining, it is possible to have a call that boxes and then uses the type dispatch of the polymorphic function immediately in sequence, which their compiler then removes. Since TIL is a whole-program compiler and inlined very aggressively, it was so effective at removing all boxing on the small benchmarks they used that it is hard to tell if this approach would scale to larger programs [TMC$^+$04]. Regardless, they got extremely good results with a primarily type-directed approach.

3.6.2 Arity Raising

Hannan and Hicks produced some of the earliest formal work on the correctness of arity raising [HH98]. They showed that if a type system captures pairs in its types, then functions that take a single argument that is a pair can be transformed to instead take two arguments (consisting of the individual elements of the pair), all of the call sites can be fixed up, and the program still computes the same value. This transformation was very straightforward and did arity raising whenever it was possible, in a purely type directed manner. Like this work, they rely on the type system to determine where arity raising is safe. Unlike our work, they flatten all types that it is possible to flatten without regard for what the function uses.

Recently, Ziarek et al. showed an enhanced arity raising transformation for MLTon [ZWJ08]. The MLTon compiler performs a full defunctorization, monomorphisation, and defunctionalization of the program. This set of transformations means that there are no polymorphic functions left at the time that arity raising is being performed. Their approach to arity raising involves passing both the original version of the argument and all of the flattened arguments (relying on useless variable elimination to remove the original version). They provide and compare three different strategies for arity raising: *flatten-all* flattens every data type completely into arguments, *argument-only* just flattens the first level of the argument to functions, and *bounded* attempts to flatten tuples up to a fixed depth only if they were created in the calling function. This heuristic is similar to our combination of control-flow analysis and target usage to determine which parts of the data structure to flatten; we never leave the original version of the argument around, however, as our analysis guarantees that we pass the necessary substructures of the data to the called function. This work has been implemented and benchmarked in the context of MLTon, but is not part of the standard distribution.

Bolingbroke and Peyton-Jones came up with a new intermediate representation for their implementation of Haskell, GHC, that is strict [BPJ09]. By translating laziness into thunks, lazy evaluation is captured in a function and then forced at any use site.

Now, strictness analysis is not required for the classic optimizations discussed earlier in this and the previous section on boxing — they can be used as-is. Their proposal for arity raising uses only lexically apparant arguments and type information available at the call site. They perform neither control-flow analysis nor defunctionalization, so they will not be able to handle higher-order arguments in a rich way. This work has not yet been implemented.

CHAPTER 4

RESULTS

4.1 Reference Counted Control-Flow Analysis

In Manticore, there is a program point associated with each term in the intermediate representation. In this analysis, we assess the runtime performance of the algorithm by counting the reduction in the number of program points visited on a variety of programs.

Benchmark	Base	Shivers^{TS}	$\text{Shivers}^{TS}_{Cond}$	Shivers^{Set}	$\text{Shivers}^{Set}_{Cond}$	RC^{Set}	RC^{Set}_{Cond}
barnes-hut	6793	1165	1165	1035	1035	795	216
life	7778	3632	3541	1927	1847	1689	969
raytracer	14310	383	383	381	381	827	827
mandelbrot	5355	849	849	733	733	417	417
quickhull	12438	9903	10187	7323	6977	395	395

The implementation of 0CFA we are comparing against is in the Serrano style. Smaller numbers of program points indicates better performance. The different algorithms are all implementations of 0CFA, but differ in the following ways:

- Base: Serrano-style
- Shivers^{TS}: Shivers-style using timestamps to determine whether to re-evaluate functions
- $\text{Shivers}^{TS}_{Cond}$': As above, but also tracking booleans and using them to reduce the number of conditional arms followed

- Shivers^{Set}: Shivers-style tracking values passed to functions to determine whether to re-evaluate them
- Shivers$^{Set}_{Cond}$: As above, with boolean handling
- RC^{Set}: Reference-counted style tracking values passed to functions to determine whether to re-evaluate them
- RC$^{Set}_{Cond}$: As above, with boolean handling

With the exception of the raytracer benchmark, the reference-counted control-flow analysis with conditional handling visits the fewest number of program points during analysis. While the performance of RC$^{Set}_{Cond}$ is still excellent relative to the base implementation, in the case of the raytracer benchmark this algorithm does not perform as well as the Shivers family of algorithms. The Shivers family visit less overall program points because of their interaction with implementation-defined cutoff values. To prevent the growth of tracked abstract tuple values from getting out of hand and approaching the theoretical polynomial runtime, within Manticore we limit the overall size of tuple values that we will track. Once the size of the tracked abstract value exceeds a given size, all future values added to that abstract value are assumed to escape and are assigned the least-precise value of $\top$ for their call sites. For the following example, assume that a hypothetical implementation only tracks a maximum of one abstract value.

```
let fun double (x) = x+x
    and veryLong (y) = ...
    and apply (f, n) = f¹(n)
in
    apply (double, 2);
    apply (veryLong, 3)
end
```

When the reference counted analysis encounters the second call to the `apply` function, the only abstract value associated with the variable `f` will be `veryLong`. Therefore, the reference counted strategy (and a garbage-collected strategy, like Might's ΓCFA as well) will perform analysis of the `veryLong` function. A Shivers-style analysis, however, would add the value `veryLong` to the abstract value associated with `apply`'s parameter `f` and encounter the hypothetical implementation limit above of two items. Both functions associated with the parameter `f` would therefore be assumed to escape, and both functions (and their bodies) would be assumed to be called from anywhere. Only a linear program text scan is required to mark a function as callable from anywhere, whereas tracking the call sites associated with a given function can be substantially more expensive, as it is in the raytracer example in the table above.

4.2 Arity Raising

4.2.1 Benchmarks

Barnes-hut is based on the GHC benchmark, and is a floating point heavy benchmark with a large number of datatypes. We gain significant speedup in time, reduction in program size, and reduction in heap allocations on this benchmark. Life is a simulation of Conway's game of life. This benchmark is the worst-case example — the code operates over lists of tupled integers. Fortunately, though, our arity raising algorithm does no harm to it. Interestingly, the last three benchmarks all already had decent register usage even without arity raising on, due to later compiler transformations including closure conversion. In these cases, we have still managed to move a significant number of heap allocations to instead be done as spilled data. This transition is beneficial in Manticore because it costs fewer bytes to store several objects as spilled data in the continuation closure than to individually heap allocate each object. While the transition away from heap to stack allocated data does not show significant improvements in the sequential executions of the program, we believe that for long running and parallel benchmarks reducing pressure on the garbage collector will improve performance.

4.2.2 Default Calling Convention

Comparing our arity raising algorithm against the default ML calling convention, one benchmark has a speedup in time and nearly all have significant reductions in overall heap-allocated data. The percentages are $\frac{t_{full}}{t_{default}}$, where t_{full} is the full arity raising algorithm presented in this work, and $t_{default}$ is the default ML calling convention. Smaller is better.

Benchmark	Size	Allocation	Time
barnes-hut	47%	53%	60%
life	100%	97%	99%
mandelbrot	99%	17%	94%
quickhull	100%	85%	96%
raytracer	100%	74%	94%

4.2.3 Argument-Only Convention

Though these numbers are impressive, there is no serious ML compiler we are aware of that skips removal of argument tupling where possible. To assess the performance of our arity raising algorithm in a more realistic context, we compare it against an implementation of *argument-only* arity raising within the context of Manticore. This implementation of argument-only arity raising is significantly more aggressive than a type-based implementation and also uniformly handles datatype flattening and value unboxing. The percentages are $\frac{t_{full}}{t_{argument}}$, where t_{full} is the full arity raising algorithm presented in this work, and $t_{argument}$ is argument-only arity raising. Again, smaller percentages are better.

Benchmark	Size	Allocation	Time
barnes-hut	100%	99%	100%
life	100%	100%	98%
mandelbrot	100%	34%	100%
quickhull	100%	100%	100%
raytracer	99%	98%	100%

There is almost no change in execution time for any of the benchmarks. The mandelbrot benchmark had a significant reduction in memory usage. Other benchmarks only had a small reduction in heap allocated bytes because within these benchmarks there are few places where it is profitable to remove several levels of memory indirection.

REFERENCES

[AD98] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 20(4):845–868, 1998.

[ADH+98] H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams IV, D.P. Friedman, E. Kohlbecker, G.L. Steele Jr., D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *Higher-order and Symbolic Computation*, 11(1):7–105, August 1998.

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

[BPJ09] Maximilian C. Bolingbroke and Simon L. Peyton Jones. Types are calling conventions. In *Proceedings of the 2009 Haskell Workshop*, pages 1–12, New York, NY, USA, 2009. ACM.

[Fax02] Karl-Filip Faxén. Representation analysis for coercion placement. *Lecture Notes in Computer Science*, 2477:499–503, 2002.

[FRR+07] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 37–44, New York, NY, January 2007. ACM.

[Gou94] Jean Goubault. Generalized boxings, congruences and partial inlining. In *In First International Static Analysis Symposium (Namur, Belgium, September 1994)*, 1994.

[HH98] John Hannan and Patrick Hicks. Higher-order arity raising. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, New York, NY, USA, 1998. ACM.

[HJr94] Fritz Henglein and Jesper Jø rgensen. Formally optimal boxing. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 213–226, New York, NY, USA, 1994. ACM.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)*, pages 261–272, New York, NY, USA, 1997. ACM.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Conference record of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363, New York, NY, USA, 1986. ACM.

[JW95] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Second International Symposium, SAS '95, Glasgow, UK, September 25-27 1995*, pages 207–224. Springer, 1995.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on intermediate representations*, pages 13–22, New York, NY, USA, 1995. ACM.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL '92)*, pages 177–188, New York, NY, USA, 1992. ACM.

[Ler97] Xavier Leroy. The effectiveness of type-based unboxing. In *In Workshop on Types in Compilation, Amsterdam, June 1997. ACM SIGPLAN.*

Published as Boston College Computer Science Dept. Technical Report BCCS-97-03, 1997.

[Mig07] Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2007.

[MJ09] Jan Midtgaard and Thomas P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, New York, NY, USA, 2009. ACM.

[MS06] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 13–25, New York, NY, USA, 2006. ACM.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture (FPCA '91)*, pages 636–666, London, UK, 1991. Springer-Verlag.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 367–378, New York, NY, USA, 1995. ACM.

[Rep06] John Reppy. Type-sensitive control-flow analysis. In *Proceedings of the 2006 ACM SIGPLAN Workshop on ML*, pages 74–83, New York, NY, USA, 2006. ACM.

[Ser95] Manuel Serrano. Control flow analysis: a functional languages compilation paradigm. In *Proceedings of the 1995 ACM symposium on Applied Computing*, pages 118–122, New York, NY, 1995. ACM.

[SF96] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 50–61. ACM Press, 1996.

[Shi88] Olin Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '88)*, pages 164–174, New York, NY, June 1988. ACM.

[Shi91] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991.

[SHO98] Bratin Saha, Nevin Heintze, and Dino Oliva. Subtransitive CFA using types. Technical report, Yale University, 1998.

[TD94] David Tarditi and Amer Diwan. Measuring the cost of storage management. In *Lisp and Symbolic Computation*, pages 323–342, 1994.

[Thi95] Peter J. Thiemann. Unboxed values and polymorphic typing revisited. In *Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 24–35, New York, NY, USA, 1995. ACM.

[TMC+96] Dave Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, pages 181–192, New York, NY, USA, 1996. ACM.

[TMC+04] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: a type-directed, optimizing compiler for ML. *SIGPLAN Not.*, 39(4):554–567, 2004.

[VHM08] David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International*

Conference on Functional Programming, pages 275–282, New York, NY, USA, 2008. ACM.

[Wee06] Stephen Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 ACM SIGPLAN Workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM.

[ZWJ08] Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. Flattening tuples in an SSA intermediate representation. *Higher-Order and Symbolic Computation*, 21(3):845–868, 2008.