Arity Raising in Manticore

Lars Bergstrom and John Reppy

University of Chicago, Chicago IL 60637, USA
{larsberg,jhr}@cs.uchicago.edu

Abstract. Compilers for polymorphic languages are required to treat values in programs in an abstract and generic way at the source level. The challenges of optimizing the boxing of raw values, flattening of argument tuples, and raising the arity of functions that handle complex structures to reduce memory usage are old ones, but take on newfound import with processors that have twice as many registers. We present a novel strategy that uses both control-flow and type information to provide an arity raising implementation addressing these problems. This strategy is *conservative* — no matter the execution path, the transformed program will not perform extra operations.

1 Introduction

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 [1]. 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 used 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 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 4, 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 present performance measurements of our implementation in Section 7 then cover the substantial related work and conclude.

2 Preliminaries

We use the direct style intermediate representation in Figure 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 maps in Figure 2, computed using a control-flow analysis, to build this graph. Our implementation uses a control-flow analysis similar to that presented by Serrano [2], which provides sufficient information to implement these maps.

$\mathcal{F}$ maps each function identifier to the list of all program points (call sites) if they are known. If a function identifier g has an unknown call site, then $\mathcal{F}(g)$ is $\emptyset$. 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.

$Exp \ni e ::= x$		variable or function name
	$\mid$ fun $g(\boldsymbol{x}) = e_1$ in e_2	function binding
	$\mid$ let $x = e_1$ in e_2	local variable binding
	$\mid$ if x then e_1 else e_2	conditional
	$\mid$ $g^l(\boldsymbol{x})$	application (labeled)
	$\mid$ $\langle \boldsymbol{x} \rangle$	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

Fig. 1. Direct style intermediate representation.

$\mathcal{F} : \text{FunID} \to 2^{\mathbb{L}}$	Function call sites
$\mathcal{C} : \mathbb{L} \to 2^{\text{FunID}}$	Called functions by call site
$\mathcal{A} : \text{FunID} \to 2^{\text{FunID}}$	Functions sharing call sites
$\mathcal{U} : \text{VarID} \to \mathbb{N}$	Variable use count

Fig. 2. Maps computed by static analysis.

$\mathcal{A}$ 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.

3 Signature Analysis

The signature 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.

3.1 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*

$$\mathcal{V} : \text{var} \rightarrow \text{path}$$

maps a variable to an access path. The notation $\mathcal{V}_f$ refers to the map $\mathcal{V}$ restricted to those variables defined within the function `f`. The initial value for each variable is .

The *path map* $\mathcal{P}_f$ maps an access path to a count of the number of times that path is directly used. The path map $\mathcal{P}_f$ is specific to function f, 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.

To illustrate these definitions, consider the following intermediate representation for the function `f`:

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

The intermediate representation for the function `f` above has the following variable and path maps:

$$\mathcal{V} = \{\mathtt{x} \mapsto 0, \mathtt{a} \mapsto 0.0, \mathtt{b} \mapsto 0.0.1\}$$
$$\mathcal{P}_{\mathtt{f}} = \{0 \mapsto 0, 0.0 \mapsto 0, 0.0.1 \mapsto 1\}$$

The variable map indicates 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. And the path map indicates that only the variable `b` is used outside of tuple selection expressions.

The imperative map $\mathcal{V}$ is filled in by the algorithm $\mathbb{V}$ in Figure 3. 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. The map $\mathcal{P}$ is defined directly.

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 initially 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.

$$
\begin{aligned}
\mathbb{V}[\![]\!] &: Exp \to Unit \\
\mathbb{V}[\![\text{fun } g(\boldsymbol{x}) = e_1 \text{ in } e_2]\!] &= \forall x_i \in \boldsymbol{x}\ (\mathcal{V}(x_i) := i); \mathbb{V}[\![e_1]\!]\ ;\ \mathbb{V}[\![e_2]\!] \\
\mathbb{V}[\![\text{let } x = \#i(y) \text{ in } e_2]\!] &= \begin{cases} \mathcal{V}(x) := \mathcal{V}(y).i; \mathbb{V}[\![e_2]\!] & \text{when } \mathcal{V}(y) \neq \emptyset \\ \mathbb{V}[\![e_2]\!] & \text{otherwise} \end{cases} \\
\mathbb{V}[\![\text{let } x = e_1 \text{ in } e_2]\!] &= \mathbb{V}[\![e_1]\!]\ ;\ \mathbb{V}[\![e_2]\!] \\
\mathbb{V}[\![\text{if } x \text{ then } e_1 \text{ else } e_2]\!] &= \mathbb{V}[\![e_1]\!]\ ;\ \mathbb{V}[\![e_2]\!] \\
\mathbb{V}[\![e]\!] &= ()
\end{aligned}
$$

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

Fig. 3. Algorithm to compute variable and path maps.

3.2 Computing Signatures

The function's signature is a list of all arguments passed to the function.[1] 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 list 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 list 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 calling function 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 from any callers of the `usesTwo` function, 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 is defined as follows:

$$
\sigma_f = \{\ p \mid p \in \rho_f \wedge (\nexists q \in \rho_f)(q \prec p)\}
$$

where $\rho_f = \{\ p \in \text{rng}(\mathcal{V}_f) \ \wedge\ \mathcal{P}_f(p) > 0\}$ is the list of all of the access paths corresponding to variables in the function `f` with non-zero use counts after substracting

[1] In the implementation of Manticore, the signature also includes the current exception handler.

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.

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 merged signature from their ideal signatures. The merged signature of two ideal signatures is a set consisting of the shortest prefix paths between the two signatures and is defined as follows:

$$\sigma_1 \uplus \sigma_2 = \{\, p \mid p \in \sigma_1 \wedge (\nexists q \in \sigma_2)(q \preceq p)\} \cup \{\, p \mid p \in \sigma_2 \wedge (\nexists q \in \sigma_1)(q \preceq p)\}$$

Note, however, that the merged signature may not be safe. Consider the pair of functions below, `usesFirst` and `usesSecond`, and assume that they share a call site.

```
fun usesFirst(param) =
  let first = #1(param)
  in first
fun usesSecond(param) =
  let second = #2(param)
  in second
```

They will have the following ideal signatures:

$$\sigma_{\texttt{usesFirst}} = \{0.1\}$$
$$\sigma_{\texttt{usesSecond}} = \{0.2\}$$

And therefore their merged signature is: $\sigma_{\texttt{usesFirst}} \uplus \sigma_{\texttt{usesSecond}} = \{0.1, 0.2\}$

Unfortunately, there is no guarantee that it is safe to perform the merged selections at all of the unshared call sites. For example, assume `usesFirst` is called in the following way:

```
let x = <2.0>
in usesFirst (x)
```

Then adding a selection of a second element as required by the shared signature would result in the following unsafe code after transformation:

```
let x = <2.0>
let first = #1(x)
let second = #2(x)
in usesFirst (first, second)
```

Since there is no second element in the allocated argument tuple, the transformation will have introduced an unsafe selection.

In an untyped setting, 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.[2] In that case, the map $\mathcal{S}$ will instead return a final signature corresponding to the default calling convention.

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

4 Transformation

Each new function signature requires the code to be transformed in three places. Figure 4 shows the transformation process on this intermediate representation via the $\mathbb{T}$ transformation.

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.

5 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

[2] See the implementation notes in Section 6 for how we avoid this limitation in Manticore through the use of type information.

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

Fig. 4. Algorithm to arity raise functions.

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 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)
```

This intermediate representation is clearly too naive to generate efficient code from. Note that we use the same mechanism, allocation, to box raw values, to allocate tuples, and to allocate datatypes. This similarity, which is manifest in the Manticore intermediate representation, allows our arity raising algorithm to treat all three mechanisms uniformly. 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 and path maps:

$$\mathcal{V} = \left\{ \begin{array}{l} \texttt{params} \mapsto 0, \texttt{dims} \mapsto 0.0,\ \texttt{fourB} \mapsto 0.1, \\ \texttt{four} \mapsto 0.1.0,\ \texttt{twoB} \mapsto 0.0.0,\ \texttt{two} \mapsto 0.0.0.0 \end{array} \right\}$$

$$\mathcal{P}_{\texttt{f}} = \{0 \mapsto 0,\ 0.0 \mapsto 0,\ 0.1 \mapsto 0,\ 0.1.0 \mapsto 1,\ 0.0.0 \mapsto 0,\ 0.0.0.0 \mapsto 1\}$$

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:

$$\sigma_{\texttt{f}} = [0.1.0, 0.0.0.0]$$

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)
```

6 Implementation

Manticore [3] is a compiler for a parallel programming language based on Standard ML. Arity raising is performed on the weakly typed, continuation-passing style (CPS)

intermediate representation. Unlike the direct style representation, the CPS representation of the program treats both function calls and returns uniformly. This uniform treatment allows arity raising to remove allocations not only on arguments to functions, but also from values returned in the original source program.

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. In Manticore, we first compute the merged signature with all functions that share call sites. Instead of then removing all unique selection paths, as presented earlier in this paper, we check the type of the provided argument at each call site to ensure that the selection path is both safe and has the same representation. The selection path is safe if the selection path is guaranteed to be valid. For example, the selection path `0.1` is valid in the type `((int * int) * any)` but is not a safe selection into an argument with type `(any * any)`. The selection path has the same representation if the type of the data selected has the same representation format. For example, since raw floating point numbers have a different representation than raw integers, even though the selection path `0.1` is valid in the types `((int * int) * any)` and `((int * float) * any)`, it would be unclear whether the argument should be passed as an integer in a general-purpose register or in a floating-point register. The selection path `0` would be used instead in this case, as the tuple types `(int * int)` and `(int * float)` share the same representation.

We are also less conservative in Manticore with branches than we presented in Section 1, where we stated that no transformations are performed inside of the arms of conditional branches. 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 that path to be added into the maps, but we do allow analysis to continue within the arms of the conditionals. Consider the following function, which extracts two parameters, performs a conditional check on one, and then performs some other operations.

```
fun f(params) =
  let x = #0(params)
  let y = #1(params)
  in if null? x
    then ...
    else ...
```

Within the arms of the conditional, if there are further selections to the paths corresponding to the variables `x` and `y`, only selections through the variable `y` will be added to the variable map. We conservatively assume that, unless the parameter type guarantees that the selection is safe, the conditional may be protecting operations based on the runtime structure of the data.

7 Results

We performed two studies on a set of four benchmarks. The first study investigates the reduction in allocated bytes of memory across the default ML function calling conven-

tion without any arity raising, an *argument-only* form of arity raising similar to what is used in most other strongly-typed functional language implementations, and the full arity raising algorithm presented in this paper. The second study examines the impact on program runtime across these three arity raising strategies.

7.1 Experimental Method

Our test machine has four quad-core AMD Opteron 8380 processors running at 2.5GHz. Each core has a 512KB L2 cache and shares 6MB cache with the other cores of the processor. The system has 32GB of RAM and is running Debian Linux (kernel version 2.6.31.1-amd64).

7.2 Benchmarks

We evaluated the three strategies on the benchmarks listed in the table below. For each benchmark, this table provides the size, in lines of code, and a small description of the type and shape of data manipulated. Since we cannot perform arity raising on any functions that the control-flow analysis determines may escape, we also provide the number of functions eligible for arity raising and the number of functions that escape. Some of the escaping functions are due to imprecision in the control-flow analysis, but many more are due to passing functions to our C runtime for storage in the scheduling queues used to implement Manticore's parallel language features.

Benchmark	Size	Eligible	Escaping	Description
barnes-hut	323	504	97	Floating point with datatypes
life	181	106	39	Integers and list operations
mandelbrot	91	228	55	Loop-heavy floating point
quickhull	138	366	76	Floating point with heavy random access
raytracer	548	308	55	Floating point with trivial parallelism

The Barnes-Hut benchmark [4] is a classic N-body problem solver. Each iteration has two phases. In the first phase, a quadtree is constructed from a sequence of mass points. The second phase then uses this tree to accelerate the computation of the gravitational force on the bodies in the system. Our benchmark runs 20 iterations over 1,000,000 particles. Our version is a translation of a Haskell program [5].

Life is a simulation of Conway's game of life. This benchmark is an example of code that this arity raising algorithm cannot optimize — the code operates over lists of tupled integers. Since there are no explicit, deep data structures to remove and access to individual data elements is guarded in conditionals, we cannot promote the data to arguments.

Mandelbrot renders a 3000×3000 image of the Mandelbrot set. This benchmark has a very tight set of inner loops over floating point values that take most of the runtime and benefits heavily from arity raising, since loops are implemented as function calls and floating points values are boxed and stored in memory rather than registers in the default ML calling convention.

The Quickhull benchmark determines the convex hull of 20,000,000 points in the plane. This algorithm is interesting for Manticore because while it is trivially parallel, the parallel subtasks are not guaranteed to be of equal work sizes. This code is based on the algorithm presented in [6].

The Raytracer benchmark renders a 1000×1000 image in parallel as two-dimensional sequence, which is then written to a file. The original program was written in ID [7] and is a simple ray tracer that does not use any acceleration data structures.

7.3 Conventions

The default calling convention in the tables below is the default calling convention specified in the user's program. All arguments are tupled and passed as a single value to the function.

The argument-only calling convention is a straightforward, type-directed translation of the function's argument tuple from a single value into the set of tupled items. This calling convention does not remove any nested tupling.

The full calling convention is the result of the arity raising strategy presented in this paper.

7.4 Allocation

The table below shows the raw allocation data in megabytes (2^{20} bytes) allocated along with percentage improvements over the default calling convention. Smaller numbers are better, and the two percentages reported are $\frac{mb_{arg-only}}{mb_{default}}$ and $\frac{mb_{full}}{mb_{default}}$, where $mb_{arg-only}$ is the argument-only arity raising algorithm, $mb_{default}$ is the default ML calling convention, and mb_{full} is the full arity raising algorithm presented in this work. In all benchmarks, argument-only allocates fewer bytes than the default convention, and the full arity raising algorithm allocates even fewer.

Benchmark	Default	Arg-Only	Full	$\frac{mb_{arg-only}}{mb_{default}}$	$\frac{mb_{full}}{mb_{default}}$
barnes-hut	325,435	298,755	282,196	91.8%	86.7%
life	32,581	32,130	31,364	98.6%	96.2%
mandelbrot	200,421	125,136	57,558	62.4%	28.7%
quickhull	72,790	67,577	62,136	92.8%	85.4%
raytracer	369,958	273,952	273,890	74.0%	74.0%

Barnes-hut and quickhull have some opportunities to perform arity raising on tight inner loops, but the use of parallel language constructs around the tight loops as mentioned in this section's introduction prevents the optimization of several inner functions. Life operates on lists of data, providing almost no opportunities to optimize the code. However, our transformation does no harm to the generated code for life. Mandelbrot has significant opportunities for optimization due to the large number of values passed around in small structures within functions that call each other in a tight loop. Both argument-only and the full strategy turn a signficiant number of the intermediate structures into values passed in registers. The raytracer's functions generally operate over

either an individual data structure or a single vector at a time. Since there is only one level of allocation to remove, the full arity raising strategy does not show significant benefit over the argument-only convention.

7.5 Execution Time

The table below shows the execution time in seconds along with percentage improvements over the default calling convention. Again, smaller numbers are better, and the two percentages reported are $\frac{t_{arg-only}}{t_{default}}$ and $\frac{t_{full}}{t_{default}}$, where $t_{arg-only}$ is the argument-only arity raising algorithm, $t_{default}$ is the default ML calling convention, and t_{full} is the full arity raising algorithm presented in this work. In all benchmarks, argument-only runs faster than the default convention, and the full arity raising algorithm runs even faster.

Benchmark	Default	Arg-Only	Full	$\frac{t_{arg-only}}{t_{default}}$	$\frac{t_{full}}{t_{default}}$
barnes-hut	10.660	10.644	10.473	99.8%	98.2%
life	4.193	4.171	4.002	99.5%	95.4%
mandelbrot	5.027	4.427	4.152	88.1%	83.6%
quickhull	4.720	4.667	4.432	98.9%	93.9%
raytracer	7.78	7.186	6.489	92.4%	83.4%

For all of the benchmarks, small reductions in allocated bytes result in small reductions in runtime, and larger reductions in allocated bytes result in larger reductions in runtime. One interesting case, though, is the speedup on the raytracer benchmark of the full arity raising algorithm over the argument-only algorithm despite a very small reduction in the number of bytes allocated. This speedup is because there are a few core functions used in inner loops whose heap-allocated parameters were turned into raw parameters. Though the values passed to these functions were only being allocated infrequently, because the functions are called with the same data repeatedly, removal of the memory access results in a significant performance improvement.

8 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.

8.1 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 [8]. 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.

Complementary work by Peyton-Jones et al. lifted `box` and `unbox` operations into the source language (Haskell) as well as the intermediate representation [9]. 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 [10]. 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 [11]. 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 [12]. 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 [13]. 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 [14]. 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[15]. 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 $n = 1$[16]) 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 [17]. 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 [18]. Regardless, they got extremely good results with a primarily type-directed approach.

8.2 Arity Raising

Hannan and Hicks produced some of the earliest formal work on the correctness of arity raising [19]. 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 [20]. 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 [21]. 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.

9 Conclusion

We have presented a strategy for arity raising that is *conservative* and have described its implementation in the Manticore compiler. Our benchmark results show that this

strategy is effective at reducing the number of bytes allocated and the runtime for several types of programs. Programs without opportunities for arity raising are not adversely affected.

References

1. Tarditi, D., Diwan, A.: Measuring the cost of storage management. In: LASC. (1994) 323–342
2. Serrano, M.: Control flow analysis: a functional languages compilation paradigm. In: SAC '95, New York, NY, ACM (1995) 118–122
3. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A heterogeneous parallel language. In: DAMP '07, New York, NY, ACM (January 2007) 37–44
4. Barnes, J., Hut, P.: A hierarchical $o(n \log n)$ force calculation algorithm. Nature **324** (December 1986) 446–449
5. GHC: Barnes Hut benchmark written in Haskell. Available from http://darcs.haskell.org/packages/ndp/examples/barnesHut/.
6. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The quickhull algorithm for convex hulls. ACM Transactions on Mathematical Software **22**(4) (1996) 469–483
7. Nikhil, R.S.: ID Language Reference Manual. Laboratory for Computer Science, MIT, Cambridge, MA. (July 1991)
8. Leroy, X.: Unboxed objects and polymorphic typing. In: POPL '92, New York, NY, USA, ACM (1992) 177–188
9. Peyton Jones, S.L., Launchbury, J.: Unboxed values as first class citizens in a non-strict functional language. In: FPCA '91, London, UK, Springer-Verlag (1991) 636–666
10. Henglein, F., Jørgensen, J.: Formally optimal boxing. In: POPL '94, New York, NY, USA, ACM (1994) 213–226
11. Thiemann, P.J.: Unboxed values and polymorphic typing revisited. In: FPCA '95, New York, NY, USA, ACM (1995) 24–35
12. Goubault, J.: Generalized boxings, congruences and partial inlining. In: First International Static Analysis Symposium (Namur, Belgium, September 1994). (1994)
13. Serrano, M., Feeley, M.: Storage use analysis and its applications. In: ICFP '96, ACM Press (1996) 50–61
14. Faxén, K.F.: Representation analysis for coercion placement. Lecture Notes in Computer Science **2477** (2002) 499–503
15. Leroy, X.: The effectiveness of type-based unboxing. In: Workshop on Types in Compilation, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03. (1997)
16. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Trans. Program. Lang. Syst. **20**(4) (1998) 845–868
17. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: a type-directed optimizing compiler for ML. In: PLDI '96, New York, NY, USA, ACM (1996) 181–192
18. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: a type-directed, optimizing compiler for ML. SIGPLAN Not. **39**(4) (2004) 554–567
19. Hannan, J., Hicks, P.: Higher-order arity raising. In: ICFP '98, New York, NY, USA, ACM (1998) 27–38
20. Ziarek, L., Weeks, S., Jagannathan, S.: Flattening tuples in an SSA intermediate representation. Higher-Order and Symbolic Computation **21**(3) (2008) 845–868
21. Bolingbroke, M.C., Peyton Jones, S.L.: Types are calling conventions. In: HASKELL '09, New York, NY, USA, ACM (2009) 1–12