A Graph-Based Higher-Order Intermediate Representation

Roland Leißa Marcel Köster Sebastian Hack

Department of Computer Science, Saarland University

{leissa, koester, hack}@cs.uni-saarland.de

Abstract

Many modern programming languages support both imperative and functional idioms. However, state-of-the-art imperative intermediate representations (IRs) cannot natively represent crucial functional concepts (like higher-order functions). On the other hand, functional IRs employ an explicit scope nesting, which is cumbersome to maintain across certain transformations.

In this paper we present Thorin: a higher-order, functional IR based on continuation-passing style that abandons explicit scope nesting in favor of a dependency graph. This makes Thorin an attractive IR for both imperative as well as functional languages. Furthermore, we present a novel program transformation to eliminate the overhead caused by higher-order functions. The main component of this transformation is *lambda mangling*: an important transformation primitive in Thorin. We demonstrate that lambda mangling subsumes many classic program transformations like tail-recursion elimination, loop unrolling or (partial) inlining. In our experiments we show that higher-order programs translated with Thorin are consistently as fast as C programs.

1. Introduction

Nowadays, many (existing and novel) programming languages (C++11, Java 8, Scala, Go, Rust, etc.) support imperative *and* functional programming. Essentially, a function f may be passed to another *higher-order function (HOF)* g while f may contain free variables. The common way to implement a function value is a closure: A record that contains the pointer to a function and value bindings of its free variables. For example, the C++ code in Figure 1a results into the (stylized) code in Figure 1b. This transformation is called *closure-conversion* [3, chap. 10]. However, implementing closures straightforwardly can incur a significant performance penalty. Ideally, the code in the example is compiled into a simple loop.

The IRs used in compilers for imperative languages (LLVM, Java Bytecode, etc.) are too low-level to represent free variables directly. Therefore, these compilers implement closure-conversion already in the front-end. This has several drawbacks:

1. The implementation is language-specific and can hardly be reused in another front-end.
2. The IR code is significantly bloated. For every function abstraction a new struct is created. For example, the LLVM code for the simple example consists of over 600 lines.
3. Finally, it is inelegant and inefficient to lower constructs that have to be restored later on. LLVM, for example, uses a combination of carefully coordinated analyses and transformations to eliminate closures: Inline the call to the closure's function pointer to be able to SSA-construct (mem2reg) the closure struct and finally dissolve the struct to scalar values. This strategy, for instance, fails to optimize recursive HOFs, like the one in the example above: After optimizations, the example in Figure 1a is still more than 250 lines of LLVM bitcode long.

IRs for functional languages [e.g. 3, 17] represent free variables naturally. Closure conversion is much less involved on these IRs because they contain closures as first-class citizens. Therefore, the compiler does not need to re-discover closures from their implementation as in imperative IRs.

However, one particular point makes the use of existing functional IRs unattractive: They rely on scope nesting to bind the use of a variable to its definition. Scopes are necessary to disambiguate a variable use when the variable name is defined more than once in the program. Although functional programs share with SSA the property that every use has exactly one reaching definition, they allow reusing the same variable name. Figure 2a shows an imperative program, (b) the SSA form of it in a stylized imperative IR, (c) the corresponding version in a stylized functional IR that uses continuation-passing style (CPS). Because the scope nesting is a feature of the *syntax* of the IR, all program transformations must maintain a correct scope nesting, which involves, among other things, proper renaming of the variables to avoid name capturing. This makes control-flow restructuring program transformations cumbersome to implement.

In this paper, we present the IR Thorin that blends concepts of imperative and functional IRs and is equally well-suited to represent imperative as well as functional programs. Thorin represents functions as first-class citizens and exploits the long-known correspondence between SSA and CPS [16] to *uniformly* represent control flow by continuations: jumps to basic blocks, function calls, generators, exceptions, etc. In that respect, Thorin is similar to an IR of a compiler for functional languages.

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00 ©2015 IEEE

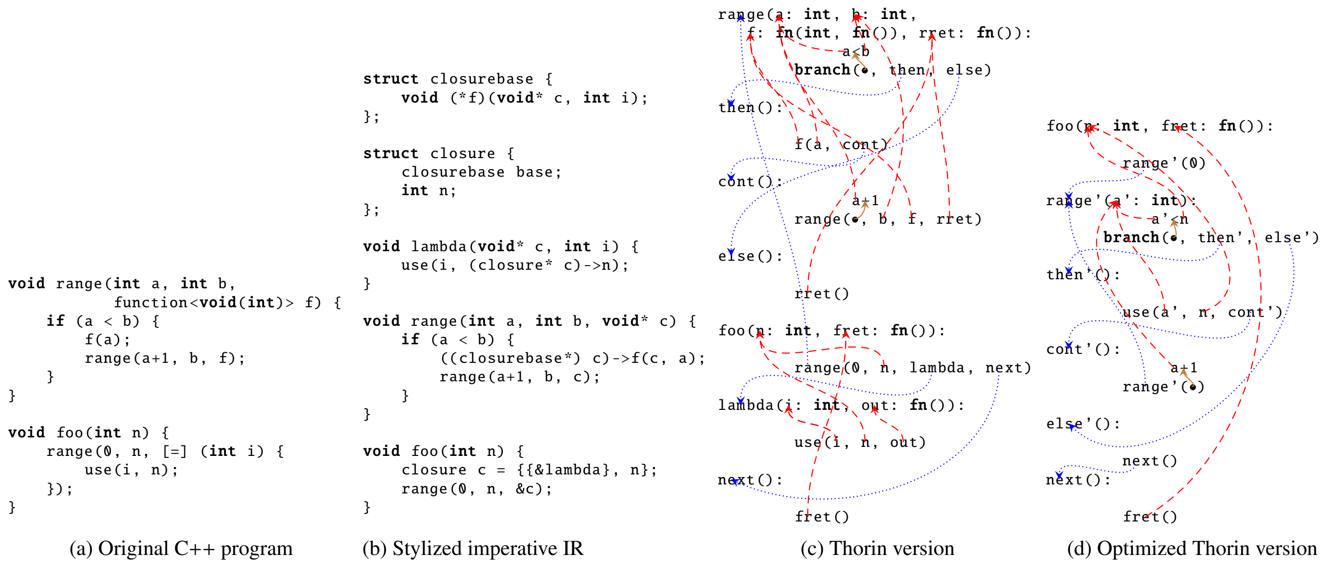


Figure 1. The higher-order function `range` and a call site within `foo`.

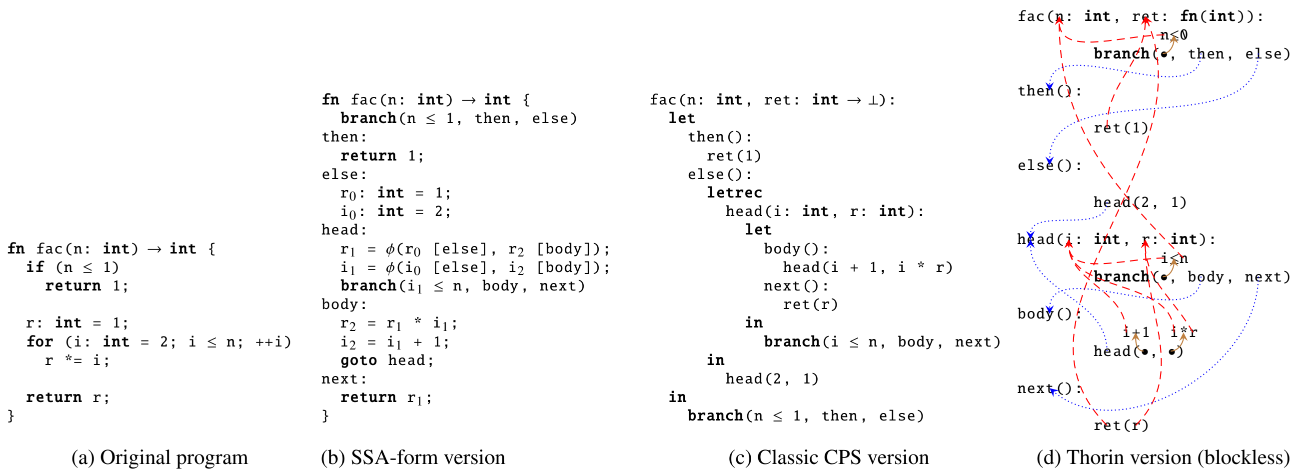


Figure 2. Factorial in different versions. In Thorin, each definition is a node. Every use is an edge to this node. Names are solely present for readability. They have no semantic meaning.

However, like low-level imperative IRs but unlike functional IRs, Thorin has no scope nesting. Instead, Thorin is graph-based like progressive imperative IRs (e.g. HotSpot [7, 22] or Firm [5]). That is, Thorin does not use named variables. In Thorin, each value is a node in a graph and every reference to a value is an edge to this node. Therefore, Thorin does not require explicit scope nesting. Figure 2d shows the Thorin graph for the example: A Thorin program consists of a set of functions. A function introduces parameters and in turn solely consists of a call. As Thorin uses CPS, the call does not return and, hence, function types (denoted by `fn(T)`) do not have a return type. The callee and the arguments to that call reference other definitions: functions (dotted edges), function parameters (dashed edges) or primops (solid edges). A primitive operation (primop) is a simple operation which references other definitions to produce a new value. Note that in this paper we nevertheless use names in Thorin programs to make the presentation more accessible for humans. Names have no meaning otherwise.

In summary, this paper makes the following contributions:

- We discuss Thorin formally by presenting its syntax as well as its static and dynamic semantics (Section 3).
- We present *lambda mangling* (Section 4), a novel program transformation that partially in- and outlines functions. Partial inlining is the crucial step in closure elimination. We show that lambda mangling is substantially simpler to implement on Thorin than on existing IRs because Thorin does not have explicit scope nesting. Furthermore, we show how CPS helps to reduce other important control flow transformations, such as tail-recursion elimination, loop unrolling, and loop peeling, to lambda mangling.

We show that this simplifies the implementation of these transformations significantly.
- We define the set of *control flow form (CFF)* Thorin programs. These can be translated to control flow graphs (CFGs) without dynamically-allocated closures. We define the set of *CFF-convertible* programs, a superset of CFF programs, and present an algorithm that is guaranteed to convert them to CFF by using lambda mangling (Section 5). This superset includes typical higher-order programming idioms like `map`, `fold` and generators. Coming back to the `range` example, Figure 1c is CFF-convertible and Figure 1d shows the program in CFF after applying the algorithm.
- We evaluate our approach on *The Computer Benchmark Game*, a set of mid-sized programs to benchmark performance across various programming languages, imperative as well as functional ones. We ported the benchmark suite to our research language Impala that provides imperative as well as functional paradigms and uses Thorin as its IR. We show that Impala's implementations using HOFs match the performance of the corresponding C implementations (Section 6).

2. Overview

In this section, we briefly recap important properties of existing IRs for functional languages and informally show how Thorin differs from them.

2.1 Name Capture

A well-known problem when transforming functional programs is *name capture* [4]. The following naïve β-reduction of λx is incorrect:

$$\lambda a.(\lambda x.(\lambda a.x))\ a \quad \Rightarrow \quad \lambda a.\lambda a.a$$

Originally, the variable a referred to the outer λa but now it refers to the inner λa. One solution is to introduce new names during β-reduction: $\lambda a.\lambda a'.a$. This technique is used by the inliner of the Glasgow Haskell Compiler (GHC) [23], for example. The downside is compile-time and implementation overhead in the compiler.

A simpler solution is to give up on names and use a graph. Each use directly points to its definition:

$$\lambda \bullet(\lambda \bullet(\lambda \bullet \bullet))\bullet) \quad \Rightarrow \quad \lambda \bullet \lambda \bullet \bullet$$

The β-reduction becomes straightforward now. It is not necessary to rename anything: We simply update the body of the inner function to point to the substituting argument. The reference in the substituting argument to the outer function still points to its proper definition.

2.2 SSA, CPS, and Scopes

The similarities between SSA form and CPS have been noted by Kelsey and Appel [2, 16]. Because understanding them is important for the rest of the paper, we briefly recap them by means of the example in Figure 2. SSA form introduces ϕ-functions in order to merge values from different predecessors (see the definitions of $\mathtt{r}_1$ and $\mathtt{i}_1$ in Figure 2b). The CPS version on the other hand introduces parameters for a local function (parameters `i` and `r` of the local function `head` in Figure 2c). The arguments to the ϕ-function in the SSA-form version appear as arguments to a call of `head` in the CPS version.

The local function `next` in Figure 2c makes use of `fac`'s higher-order parameter `ret`. This is possible as inner functions can access values defined in outer functions. In the scope of the inner function these references appear as free variables. Assume `next` is not defined locally in `fac`. In this case the callers of `next` must pass `ret` as an additional parameter to `next`. To keep the number of parameters low, functions are nested. Nesting functions according to the dominance tree results in the minimal number of parameters [9, sec. 4.1].

The disadvantage of nesting becomes clear in the example in Figure 3. Assume we want to add a conditional branch in `a` to `b` and an additional function `z`, which in turn branches to `c`. The original nesting must be repaired since `c` needs to be visible from both `b` and `z` (see Figure 3b). *Block/let floating* has to be applied to float `c` into `a`. The reverse transformation, i.e. sinking `c` into `b`, is known as *block sinking*. Such situations occur, for example, during jump threading.

Thorin removes the concept of nesting and names (Figure 2d). In Thorin, the function nesting is *implicitly* given by data dependencies between functions. If a function f uses a variable defined in another function g, f is *implicitly* nested in g. Since Thorin does not have nesting, it abolishes block floating and sinking (Figure 3c-d).[1] We say Thorin is *blockless*.

Another reason to abandon nesting is that a given nesting is not necessarily minimal. Suppose a compiler wants to inline the following function:

```
let f a b = let g x = x+1 in a + (g b)
```

The inliner would inline `g` as well although `g` does not use any of `f`'s parameters. This is why compilers for functional languages usually perform the aforementioned let floating transformation beforehand [10, chap. 3.4]:

```
let g x = x+1
let f a b = a + (g b)
```

3. Thorin

In this section, we present Thorin formally by describing its syntax, semantics, and type system (see Figure 4). We use the common notation $\overline{a}$ to denote a list $a_1, \ldots, a_n$. We write $a \preceq b$ if a is a subtree in the syntax tree of b.

For the purpose of the paper we restrict Thorin's type system to three types: the zeroth order types `int` and `bool`, and the higher-order function type $\mathtt{fn}(\overline{t})$. For example, the type `fn(int, fn(bool))` denotes a type of a second-order

[1] For the remainder of the paper we waive edges and use unique names to resolve the declaration of a definition in order to increase readability.

```
a(x: int, ret: int → ⊥) → ⊥
let
  b() → ⊥
  let
    c() → ⊥
      ret(x)
  in
     c()
in
  b()
```

(a) Classic CPS version

```
a(x: int, ret: int → ⊥) → ⊥
let
  b() → ⊥
    c()
  z() → ⊥
    c()
  c() → ⊥
    ret(x)
in
  branch(x = 0, b, z)
```

(b) Classic CPS with new branch `z`

```
a(x: int, ret: fn(int)):
  b()
b():
  c()


c():
  ret(x)
```

(c) Thorin version

```
a(x: int, ret: fn(int)):
  branch(x = 0, b, z)
b():
  c()
z():
  c()
c():
  ret(x)
```

(d) Thorin version with new branch `z`

Figure 3. This example illustrates Thorin's blockless representation.

$$
\begin{array}{rrlll}
\text{type} & t ::= & \texttt{int} \mid \texttt{bool} & & \text{(int/bool type)} \\
 & \mid & \texttt{fn}(\bar{t}) & & \text{(function type)} \\
\text{program} & p ::= & \mathcal{L} \to \texttt{fn}(\bar{t}) : b & & \\
\text{body} & b ::= & e(\bar{e}) & & \\
\text{expression} & e ::= & \boxtimes(\bar{e}) & & \text{(primop)} \\
 & \mid & \ell & & \text{(abstraction)} \\
 & \mid & \ell_i & i \in \mathbb{N} & \text{(parameter)} \\
\text{value} & v ::= & c \in \mathcal{V} & & \text{(literal)} \\
 & \mid & \langle \ell, \sigma \rangle & & \text{(closure)}
\end{array}
$$

Body: $\boxed{p, \sigma, \ell \Rightarrow \sigma', \ell'}$

$$
\text{E-Body}\ \frac{p[\ell] = _ : e_0(e_1, \ldots, e_n) \qquad \sigma, e_0 \Downarrow \langle \ell', \sigma' \rangle \quad \sigma, e_1 \Downarrow v_1 \quad \cdots \quad \sigma, e_n \Downarrow v_n}{p, \sigma, \ell \Rightarrow \sigma'[\ell \mapsto v_1, \ldots, v_n], \ell'}
$$

Expression: $\boxed{\sigma, e \Downarrow v}$

$$
\text{E-Primop}\ \frac{\sigma, e_1 \Downarrow v_1 \quad \cdots \quad \sigma, e_n \Downarrow v_n}{\sigma, \boxtimes(\bar{e}) \Downarrow \otimes(\bar{v})} \qquad \text{E-Abs}\ \frac{}{\sigma, \ell \Downarrow \langle \ell, \sigma \rangle}
$$

$$
\text{E-Param}\ \frac{\sigma[\ell] = v_1, \ldots, v_n \quad 1 \le i \le n}{\sigma, \ell_i \Downarrow v_i}
$$

Program: $\boxed{\vdash p}$

$$
\text{T-Prg}\ \frac{p := [\ell_1 \mapsto _ : b_1, \ldots, \ell_n \mapsto _ : b_n] \quad p \vdash b_1 \quad \cdots \quad p \vdash b_n}{\vdash p}
$$

Body: $\boxed{p \vdash b}$

$$
\text{T-Body}\ \frac{p \vdash e_0 : \texttt{fn}(\bar{t}) \quad p \vdash e_1 : t_1 \quad \cdots \quad p \vdash e_n : t_n}{p \vdash e_0(\bar{e})}
$$

Expression: $\boxed{p \vdash e : t}$

$$
\text{T-Primop}\ \frac{t = check_{\boxtimes}(\bar{e})}{p \vdash \boxtimes(\bar{e}) : t} \qquad \text{T-Abs}\ \frac{p[\ell] = \texttt{fn}(\bar{t}) : _}{p \vdash \ell : \texttt{fn}(\bar{t})}
$$

$$
\text{T-Param}\ \frac{p[\ell] = \texttt{fn}(t_1, \ldots, t_n) : _ \quad 1 \le i \le n}{p \vdash \ell_i : t_i}
$$

$\boxed{p \vdash b \text{ live } b'}$

$$
\text{L-Param}\ \frac{\ell \ne \ell' \quad p[\ell'] = _ : b_{\ell'} \quad \exists i : \ell_i \le b_{\ell'}}{p \vdash \ell \text{ live } \ell'}
$$

$$
\text{L-Abs}\ \frac{\ell \ne \ell' \quad p \vdash \ell \text{ live } \ell'' \quad p[\ell'] = _ : b_{\ell'} \quad \ell'' \le b_{\ell'}}{p \vdash \ell \text{ live } \ell'}
$$

Figure 4. Syntax, Semantics, Typing and Liveness in Thorin

function which expects two arguments: An integer and a first-order function which expects a boolean. As functions do not return (because of CPS), they have no return type. Note that function types are at least of first order. Parameters can be of zeroth order or higher.

A Thorin *program* consists of a map which maps a label $\ell \in \mathcal{L}$ to a function $\texttt{fn}(\bar{t}) : b$. Note that we use the label ℓ only in the formalism of Thorin to refer to the node that contains the function $\texttt{fn}(\bar{t}) : b$. In Thorin's implementation, labels do not exist. In addition, the textual representation in the examples introduces parameter names. A function consists of its type or signature $\texttt{fn}(\bar{t})$ and its *body* $b = e_0(\bar{e})$ which is a call to e_0 with arguments $\bar{e}$. The examples use a function `branch` of type $\texttt{fn}(\texttt{bool}, \texttt{fn}(), \texttt{fn}())$ for conditional branches. An *expression* can be

1. a *primop* $\boxtimes(\bar{e})$, where $\boxtimes$ is an operator (e.g. $+, -, \ldots$),
2. the ith *parameter* ℓ_i of a function ℓ or
3. the abstraction of a function ℓ.

Typing. Rule T-Prg checks the body of each function in the program. Rule T-Body gathers all argument types and checks whether each matches the type of the callee. Note that the body itself does not possess a type as the call never returns. A primop only expects specific input types and in turn has a specific output type. The function $check_{\boxtimes}(\bar{e})$ in T-Primop handles these primop-specific rules. The type of a function is resolved by looking up its signature in the program map (T-Abs). A reference to a parameter projects the corresponding type from the function's signature (T-Param).

Semantics. Expression evaluation is defined by a big-step semantics. Memory operations or side-effects are explicitly tracked by a functional store [27, 28]. The environment $\sigma : \mathcal{L} \to \mathcal{V} \times \cdots \times \mathcal{V}$ maps a function label to the *values* of its parameters. Each *value* is either some literal c of a set $\mathcal{V}$ (here: integers and booleans) or a *closure*. This is a pair consisting of a function label and environment at the moment the function is captured (E-Abs). E-Primop evaluates all arguments to values and determines a new value (operator $\oplus$ denotes the arithmetic operation as opposed to the syntactic terminal $\boxtimes$). The expression ℓ_i extracts the ith parameter value from ℓ's parameter list (E-Param).

Bodies are evaluated in small step: We evaluate the arguments $\bar{e}$ to values. The calling expression e_0 must evaluate to a closure ℓ' to which the program steps to. The new state σ' is formed by replacing the old parameter values of ℓ' by the newly evaluated arguments.

The Scope: Identifying Function Dependencies. As outlined in previous sections, functions in a Thorin program are not explicitly nested. For example, in Figure 2d the functions `then` and `next` *directly* depend on `fac` as both functions reference `fac`'s parameter `ret`. Likewise, `head` *directly* depends on `fac` because `head` uses `fac`'s parameter `n`. In Figure 2d function `else` depends on `fac` albeit `else`'s body does not directly use any of `fac`'s parameters. However, `else` invokes `head`, which directly depends on `fac`. Therefore, function `else` *indirectly depends* on `fac` and, hence, is also *implicitly* nested in `fac`. For many analyses and transformations (including lambda mangling, as presented in Section 4), we need to know all direct and indirect dependences from the view of a function. We obtain this information by applying a liveness analysis. It holds $p \vdash \ell$ live ℓ' if function ℓ is live in ℓ'. L-Param finds direct dependencies whereas L-Abs transitively determines indirect dependencies. We call all functions which are live from the view of an another function ℓ_e the *scope of* ℓ_e: $scope_p(\ell_e) := \{\ell \mid p \vdash \ell_e \text{ live } \ell\}$. We call ℓ_e the *entry function* of that scope.

4. Lambda Mangling

In this section, we present Thorin's main transformation primitive: *lambda mangling*—a combination of lambda lifting [15] and dropping [9]. We demonstrate how many traditional compiler optimizations can be implemented with this transformation and how to deal with simple as well as mutual recursion.

When reading Figure 5a from left to right, we see how function `g` is *lambda-lifted* out of `f` by introducing a new parameter `ret'` which eliminates `g`'s free variable `ret`. The new lifted version is called `g'`. Likewise—when reading Figure 5a from right to left—we see how function `g'` is *lambda-dropped* into the body of `f` by eliminating `g'`'s parameter `ret'` and introducing a free variable `ret`.

Figure 5b depicts the same procedure for a Thorin program. Since Thorin programs are blockless, we do not have to move `g`/`g'` out of/into `f`. The scope analysis (Section 3) will identify that `g` is a function nested in `f` (in the left box) whereas `g'` and `f` will be discovered as two independent functions (in the right box).

4.1 Combining Lambda Lifting and Dropping

The local function `pow(a, b)` in Figure 6a computes $\mathtt{a}^{\mathtt{b}}$. The function has two call sites: in `calcx` and `calcy`. As both callers pass `3` for parameter `b` to `pow`, we want to specialize `pow`. For this reason, we drop `pow`'s parameter `b` and substitute each occurrence of `b` with `3`. In doing so, we also apply local optimizations (constant propagation, common subexpression elimination, etc.). Thus, the check `b = 0` and related blocks are eliminated in the new function `pow_d` (Figure 6b). Moreover, we update the call sites in `calcx` and `calcy` to call the new function `pow_d` instead.

Body: $\boxed{M, b \triangleright b'}$

$$\text{M-Body}\ \frac{M, e_0 \triangleright e_0' \quad \cdots \quad M, e_n \triangleright e_n'}{M, e_0(e_1, \ldots, e_n) \triangleright e_0'(e_1', \ldots, e_n')}$$

Expression: $\boxed{M, e \triangleright e'}$

$$\text{M-Mapped}\ \frac{M[e] = e'}{M, e \triangleright e'} \qquad \text{M-Primop}\ \frac{\boxtimes(e_1, \ldots, e_n) \notin dom(M) \quad M, e_1 \triangleright e_1' \quad \cdots \quad M, e_n \triangleright e_n'}{M, \boxtimes(e_1, \ldots, e_n) \triangleright \boxtimes(e_1', \ldots, e_n')}$$

$$\text{M-Abs}\ \frac{\ell \notin dom(M)}{M, \ell \triangleright \ell} \qquad \text{M-Param}\ \frac{\ell_i \notin dom(M) \quad M, \ell \triangleright \ell'}{M, \ell_i \triangleright {\ell_i}'}$$

Figure 7. Term rewriting for lambda mangling

function $mangle(p, \ell_e, t, M)$
    foreach $\ell \in scope_p(\ell_e) \setminus \ell_e$ **do** $M[\ell] \leftarrow$ *new label*
    foreach $\ell \in scope_p(\ell_e) \setminus \ell_e$ **do**
        $\mathtt{fn}(\bar{t}) : b \leftarrow p[\ell]$ ◃ *get ℓ's signature and body*
        $\ell' = M[\ell]$ ◃ *get ℓ's associated new label*
        $p[\ell'] \leftarrow \mathtt{fn}(\bar{t}) : b'$ ◃ *insert new ℓ' where $M, b \triangleright b'$*
    end
    $\ell_e' \leftarrow$ *new label* ◃ *now deal with entry: create new label*
    $_ : b_e \leftarrow p[\ell_e]$ ◃ *get ℓ_e's body*
    $p[\ell_e'] \leftarrow t : b_e'$ ◃ *insert new entry where $M, b_e \triangleright b_e'$*
    return ℓ_e' ◃ *return entry to new mangled region*
end

Algorithm 1. Lambda mangling expects the program p to work on, the entry label ℓ_e of the scope to mangle, the new signature t and a map M which maps ℓ_e's signature to t.

The scope of function `pow` in Figure 6a uses `f`'s parameter `ret` as free variable. Assume we would like to eliminate `pow`'s dependency on `f`. To this end, we want to lift `pow` by introducing a new parameter `ret_l` (Figure 6c).

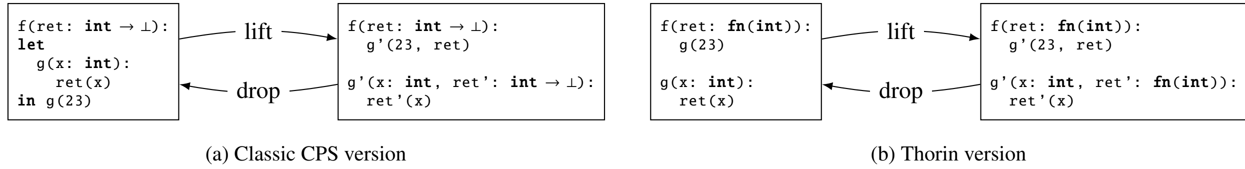
If we want to apply both transformations, we could either first drop and then lift, or first lift and then drop (Figure 6d). Or more elegantly: apply both transformations simultaneously. Both, dropping and lifting, extend the program p by a clone of ℓ's scope. In this clone all expressions are rewritten with respect to a map M (Figure 7).

Reconsider the example in Figure 6. We substitute `b` with `3` and keep the parameter `a` called `a_d` in the dropped version. We specify this mapping as follows: $M := \{\mathtt{a} \mapsto \mathtt{a_d}, \mathtt{b} \mapsto \mathtt{3}\}$. We substitute `ret` with `pow_l`'s new parameter `ret_l` and keep parameters `a` and `b` called `a_l` and `b_l` in the lifted version: $M := \{\mathtt{a} \mapsto \mathtt{a_l}, \mathtt{b} \mapsto \mathtt{b_l}, \mathtt{ret} \mapsto \mathtt{ret_l}\}$. We can simultaneously drop `b` with `3` and lift `pow`'s free variable `ret` to a new parameter `ret_m` while obeying the mapping $M := \{\mathtt{a} \mapsto \mathtt{a_m}, \mathtt{b} \mapsto \mathtt{3}, \mathtt{ret} \mapsto \mathtt{ret_m}\}$.

Algorithm 1 performs this reconstruction for the whole scope of an entry label ℓ_e, with the new signature t_e (in the example `fn(int, fn(int))`) and a corresponding mapping M. The new region is added to the program p.

4.2 Recursion

As all function calls in CPS occur in tail position, it is tempting to think that in a CPS program only tail-recursion happens. This is not the case as can be seen in Figure 8. The function `else` passes `cont` to `fac`. Reconsider Thorin



(a) Classic CPS version (b) Thorin version

Figure 5. Lambda lifting/dropping

```
f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
pow(a: int, b: int):
  branch(b = 0, then, else)
then():
  ret(1)
else():
  head(0, a)
head(i: int, r: int):
  branch(i < b, body, next)
body():
  head(i+1, r*a)
next():
  ret(r)
calcx():
  pow(x, 3)
calcy():
  pow(y, 3)
```

(a) The nested pow computes a^b.

```
f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
pow_d(a_d: int):
  head(0, a_d)

head(i: int, r: int):
  branch(i < 3, body, next)
body():
  head(i+1, r*a_d)
next():
  ret(r)
calcx():
  pow_d(x)
calcy():
  pow_d(y)
```

(b) Dropped pow_d computes a_d^3.

```
pow_l(a_l: int, b_l: int,
      ret_l: fn(int)):
  branch(b_l = 0, then, else)
then():
  ret_l(1)
else():
  head(0, a_l)
head(i: int, r: int):
  branch(i < b_l, body, next)
body():
  head(i+1, r*a_l)
next():
  ret_l(r)

f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
calcx():
  pow_l(x, 3, ret)
calcy():
  pow_l(y, 3, ret)
```

(c) Lifted pow_l doesn't use free variables.

```
pow_m(a_m: int,
      ret_m: fn(int)):
  head(0, a_m)

head(i: int, r: int):
  branch(i < 3, body, next)
body():
  head(i+1, r*a_m)
next():
  ret_m(r)

f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
calcx():
  pow_m(x, ret)
calcy():
  pow_m(y, ret)
```

(d) Dropped and lifted pow_m.

Figure 6. The function pow in (a) is always invoked with 3 as second argument. Hence, we want to specialize pow. Also, we would like to get rid of the free variable ret. Lambda dropping performs the specialization, lambda lifting eliminates pow's dependency on f and lambda mangling applies both transformations simultaneously.

```
fn fac(n: int) → int {
    if (n ≤ 1)
        return 1;
    return n * fac(n-1);
}
```

(a) Original program

```
fac(n: int, ret: fn(int)):
  branch(n ≤ 1, then, else)
then():
  ret(1)
else():
  fac(n-1, cont)
cont(res: int):
  ret(n*res)
```

(b) Thorin version

Figure 8. Naïve, recursive implementation of factorial

semantics (Figure 4): cont is a function abstraction and is evaluated to a closure. This closure captures the state of this iteration. For this reason, fac itself is not tail-recursive albeit the recursive call of fac occurs in else's tail position.

Compare this to the tail-recursive implementation in Figure 9a. It consists of a base case fac and a helper function help which implements the loop. The recursive call in help passes h_ret to help. Parameter h_ret is not a function abstraction but a parameter that is just passed around. Therefore, no closure is captured.

Concluding, it is important to distinguish tail-recursion and recursive tail-calls. Therefore, standard text book definitions for tail-recursion [e.g., 14, chap. 17.4] do not apply to CPS programs. Thus, we introduce our own nomenclature that is based on the following higher-order call graph:

1. Create a node for each Thorin function
2. For each function ℓ, draw an edge to all functions which occur in ℓ's body.

A strongly connected component (SCC) in this graph forms a recursion. We classify:

recursive call: Any call within an SCC.
simple recursive call: A recursive call within the scope of the callee.
mutual recursive call: All other recursive calls.
static parameter: A parameter which does not change its value within an SCC.
first-order recursive call: A recursive call which only uses static parameters as higher-order arguments.
loop: An SCC formed by recursive calls that are only of zeroth order.

Example. Reconsider (Figure 9a). The call of help in then2 is simple recursive since then2 belong's to help's scope. The call of help in else is *not* recursive as it is not part of the SCC formed by help and then2. The parameter h_ret is static in that SCC. This makes the call of help in then2 first-order recursive. The call of fac in else is not first-order recursive as the call passes the higher-order argument cont (Figure 8b). Note how the definition of first-order recursion reflects non-CPS tail-recursion.

4.2.1 Mangling Simple Recursion

Let us drop help into fac by dropping help's parameters h_n and h_ret with fac's parameters n and f_ret. However, our algorithm would not touch the recursive call within help's scope because we do not put the entry ℓ_e = help into the map M. We would obtain a dropped version of help, say help_d, which still calls the original function help. Just substituting ℓ_e with the new mangled function would create an ill-typed call site: help_d(i+1, r*i, h_n, h_ret). The recursive call help(i+1, r*i, h_n, h_ret) uses as third

argument h_n and fourth argument h_ret. These are help's static parameters. For this reason, we replace this call with help_d(i+1, r*i) (Figure 9b). Moreover, we replace the call site help(1, 2, n, f_ret) in else nested inside fac with help_d(1, 2) as holds:

```
help(1, 2, n, f_ret) = help(1, 2, h_n, h_ret) = help_d(1, 2) .
```

Note that the resulting program is identical to the iterative implementation (Figure 2d). In other words, with lambda mangling we performed *tail-recursion elimination* by transforming help to a loop help_d.

When we drop all parameters of the resulting function help_d with 1 and 2, we perform *loop peeling* (see Figure 9c). In this case we cannot substitute the recursive call help_d(i+1, i*r) by help_p() as the arguments are not static parameters.

Based on the program in Figure 9b, we can also drop all parameters of help_d with i+1 and r*i. This performs *loop unrolling* (Figure 9d). For the same reason as above, we cannot substitute the recursive call help_d(i+1, i*r) by calling the new function help_u(), either.

In general, we substitute calls to the entry function by calls to the mangled one if all dropped parameters are static. In the case of simple recursion, we can take this insight into account by checking for this pattern when rewriting the body in Algorithm 1.

4.2.2 Mangling Mutual Recursion

In Figure 10a functions iseven and isodd invoke each other in a mutual recursive way. There exists only a sole user of this construct: Function foo calls iseven. For performance reasons, we would like to drop iseven and isodd into foo by substituting eret/oret with foo's parameter ret. Currently, our mangling algorithm does not support this transformation: When dropping iseven to iseven' we do not know that we are going to analogously drop isodd, too, and thus, cannot substitute the recursive call isodd(ei-1, eret) with isodd'(ei'-1). Likewise, we cannot replace the call iseven(oi-1, oret) with iseven'(oi'-1) when dropping isodd.

However, we know beforehand the mapping M_e := {ei ↦ ei', eret ↦ ret} when dropping iseven to iseven' and the mapping M_o := {oi ↦ oi', oret ↦ ret} when dropping isodd to isoddd. When considering this during mangling, we can directly substitute any call of the form iseven(X, eret) with iseven'(X) and any call of the form isodd(Y, oret) with isodd'(Y) as both eret and oret are static parameters in the SCC formed by iseven, ethen, isodd and othen. Finally, the program in Figure 10b emerges. Note that the recursive calls now form a loop. Thus, we performed *mutual tail-recursion elimination*.

5. Code Generation

In order to translate Thorin programs to a lower-level program representation like machine code or an SSA-based representation (like LLVM) we classify Thorin functions in the following way:

```
foo(i: int, ret: fn(bool)):
  iseven(i, ret)

iseven(ei: int, eret: fn(bool)):
  branch(ei>0, ethen, eelse)
ethen():
  isodd(ei-1, eret)
eelse():
  eret(true)

isodd(oi: int, oret: fn(bool)):
  branch(oi>0, othen, oelse)
othen():
  iseven(oi-1, oret)
oelse():
  oret(false)
```

(a) Functions iseven and isodd are first-order recursive.

```
foo(i: int, ret: fn(bool)):
  iseven'(i)

iseven'(ei': int):
  branch(ei'>0, ethen', eelse')
ethen'():
  isodd'(ei'-1)
eelse'():
  ret(true)

isodd'(oi': int):
  branch(oi'>0, ethen, oelse)
othen'():
  iseven'(oi'-1)
oelse'():
  ret(false)
```

(b) The optimized version consists of a loop.

Figure 10. Lambda mangling to eliminate mutual tail-recursion formed by iseven, ethen, isodd and othen.

basic block (BB)-like function: A first-order function.
returning function: A second-order function with exactly one first-order parameter.
top-level function: A function whose scope does not contain free variables.
bad function: A function that is neither BB-like, nor top-level and returning.
control flow form: A scope that does not use bad functions.

Example. Reconsider Figure 2: All basic blocks in the SSA-form version are BB-like functions in Thorin. The function fac is returning and top-level. As fac's scope does not contain any bad functions the scope is in CFF.

5.1 Converting Thorin to SSA Form

It is straightforward to generate code from a CFF program. For example, we can use Kelsey's algorithm [16] to translate the program to SSA form:

- All returning functions become ordinary functions in the SSA-form program. This first-order parameter acts as "return".
- All BB-like functions become basic blocks. For each parameter we introduce a ϕ-function. The corresponding arguments of the BB's predecessors determine the ϕ-function's arguments.
- Calls to returning functions become "normal" calls. The parameter of the call's continuation forms the result value in the SSA-form program.

Example. Using this algorithm we obtain Figure 2b from Figure 2d.

5.2 Beyond Control Flow Form

The remaining top-level functions that are not in CFF, can be translated with standard code generation techniques for higher-order CPS programs [1, 3, 26]. An alternative is to eliminate bad functions. Algorithm 2 sketches how to keep

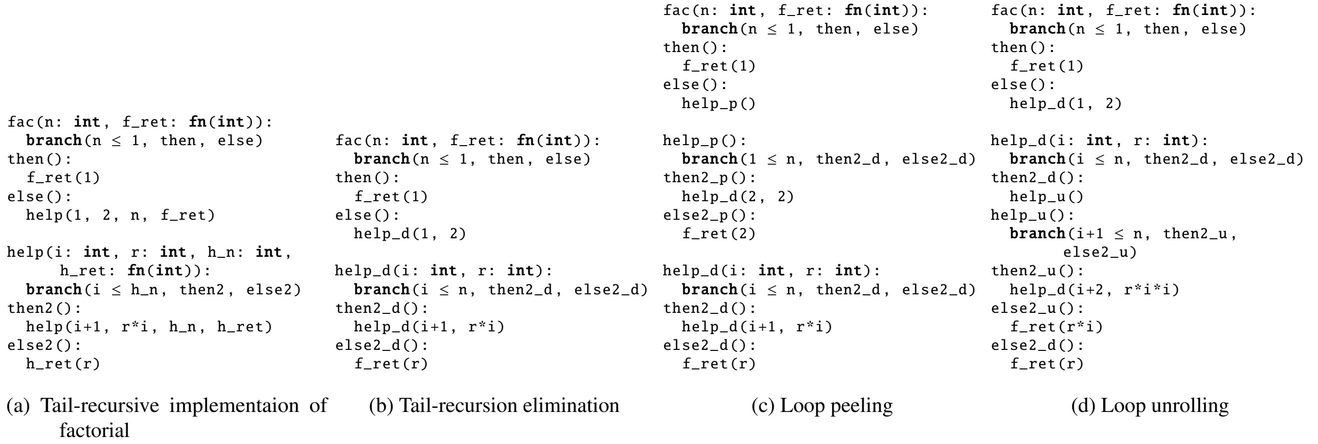
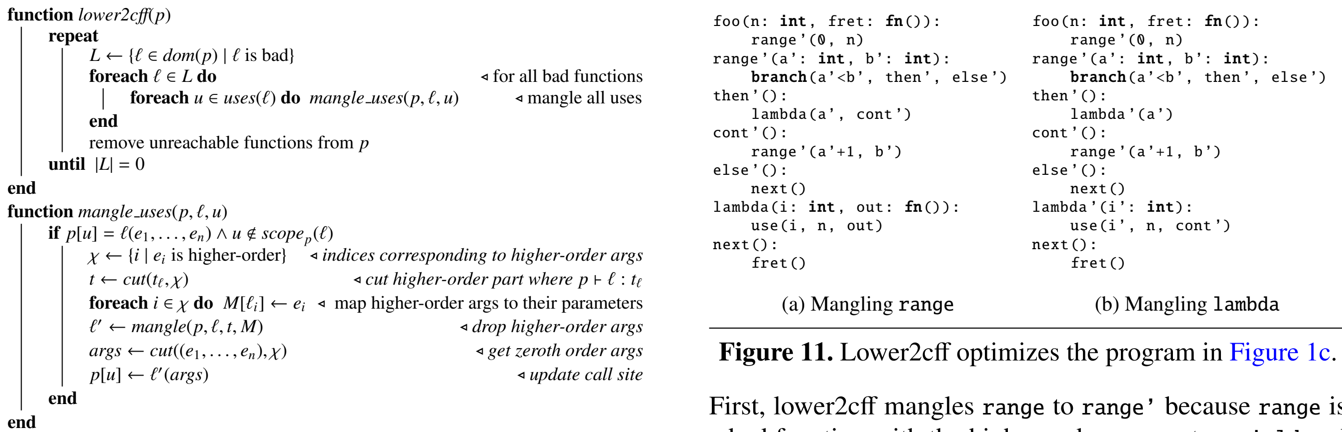


Figure 9. Tail-recursion elimination, loop peeling and loop unrolling by using lambda mangling



Algorithm 2. This algorithm elminates bad functions in program p.

Figure 11. Lower2cff optimizes the program in Figure 1c.

specializing higher-order arguments to bad functions. This routine will only terminate if all bad functions eventually disappear. In general, it is undecidable whether this will be the case for an arbitrary program p: Basically, each lambda dropping performs a function specialization. This in turn partially evaluates the program. Due to the halting problem, we cannot statically prove that all bad functions will eventually be lowered.

However, the presented lowering algorithm always reduces non-recursive calls of bad functions. Each specialized call removes one use of the bad function. This property still holds when mangling first-order recursive functions because the mangled version does not reference the original function anymore (see Section 4.2). We call such programs *CFF-convertible*. When mangling recursive but not first-order recursive functions, those references usually remain. In this case, specializing the call does not decrease the number of uses of the bad function.

Example. The program in Figure 1c uses the HOF `range` to iterate over an interval $[a, b[$. We now apply lower2cff. First, lower2cff mangles `range` to `range'` because `range` is a bad function with the higher-order parameters `yield` and `ret`. Note how mangling eliminates the static parameters (Figure 11a). The returning function `lambda` with the higher-order parameter `out` is bad since it is nested inside of `f` due to its dependency on `f`'s parameter `n`. For this reason, lower2cff mangles `lambda` and the program in Figure 11b emerges which is in CFF and the algorithm terminates. Using lambda mangling, we also inline `lambda'` and specialize `range'`'s parmeter `b'` to `n` to finally obtain the program in Figure 1d.

Enhancements. The presented lowering algorithm always uses lambda dropping. In certain cases it is worthwhile to use lambda lifting instead as lifting will not increase the code size. However, lambda lifting is only reasonable in the case of returning, non-top-level functions which only contain free variables of zeroth order (`lambda` in Figure 11b, for example). Then, the lifted function will be a returning, top-level function. Otherwise, when lifting free variables of order one or higher, the resulting function is not a returning function anymore.

Moreover, lower2cff always specializes *all* higher-order arguments. However, in order to use Kelsey's algorithm it suffices to have top-level, returning functions. This means, that it is fine to keep *one* first-order, potentially non-static parameter (the "return") of top-level bad functions. The property of first-order recursion must then only hold for the

	C	Impala	Rust	GHC
`aobench`	1.220	1.357	n/a	22.540
`fannkuch-redux`	27.137	28.070	n/a	34.670
`fasta`	2.313	1.517	n/a	1.443
`mandelbrot`	2.143	2.113	n/a	2.013
`meteor-contest`	0.047	0.043	0.050	0.327
`n-body`	5.497	6.130	5.163	6.867
`pidigits`	0.710	0.763	4.940	0.903
`regex`	6.477	6.470	18.020	7.720
`reverse-complement`	1.090	1.220	n/a	1.300
`spectral-norm`	4.423	4.480	n/a	19.347

Table 1. Median execution times of eleven runs in seconds

remaining parameters. This essentially allows to transform a greater class of programs to CFF.

Summary. Compilation works as follows:

1. Identify all bad functions.
2. Use lower2cff to eliminate all non-recursive and first-order recursive uses. For such uses lower2cff will terminate. If all uses are of these kinds, the original bad function will become unreachable and can be removed.
3. Translate all functions which are not bad with Kelsey's algorithm.
4. Translate remaining functions with conservative code generation techniques.

6. Implementation and Evaluation

6.1 Performance

We implemented a language called *Impala*[2]—a dialect of Rust[3]. Besides semantic analyses the Impala compiler does not perform any further analyses or transformations on the abstract syntax tree (AST). As Rust directly compiles to LLVM, the compiler implements many complicated analyses and transformations before translating to LLVM as pointed out in Section 1. The Impala compiler directly translates the AST—including higher-order, nested and/or polymorphic functions—to Thorin[4]. Thorin then performs its own set of optimizations including the presented lower2cff phase to eliminate closures before translating the Thorin program with Kelsey's algorithm to LLVM.

In order to evaluate the effectiveness of our approach, we ported *The Computer Benchmark Game*[5] to Impala. We elided some programs which focus on measuring API or runtime overhead. Additionally, we ported `aobench`[6]. Table 1 reports the median execution times of eleven runs for C, Impala, Rust (if available) and GHC in seconds. Our test ran on an Intel® Ivy Bridge Core™ i7-3770K CPU. We consider the C implementations as baseline. We included Rust in our measurements as Impala has similar syntax. We also included Haskell versions of the benchmarks in order to see how well GHC's Core IR performs. From the original benchmark suite we selected programs which were neither hand-vectorized nor hand-parallelized. We used clang 3.4.2, rustc 0.11 and GHC 7.8.3. The exact compile flags and benchmarks are apparent in our benchmark suite.[7]

Although we used the C versions of the benchmarks as a template for the Impala version, it is important to note that Impala does not offer C-style `for`-loops. Instead, we use HOFs to write appropriate generators and rely on Thorin's lower2cff phase. For example, in order to iterate over an interval we use a HOF `range` as presented in Section 5.2.

The performance of the Impala programs is mostly on a par with the C implementations except for `fasta` where Impala is about 1.5 times faster than C. We are still investigating this slowdown for C. Rust's performance stands or falls by the quality of the used libraries. GHC is for the most part roughly on a par with C/Impala. However, some benchmarks—in particular `aobench`—run significantly slower.

6.2 Engineering Effort

In order to estimate the engineering effort to develop code transformations we compare the Halstead metric [13] of Thorin's lambda mangling implementation versus LLVM 3.4.2 (Table 2).[8] On the one hand, LLVM is a full featured compiler suite which makes these metrics biased towards Thorin. On the other hand, lambda mangling is much more versatile: For LLVM we did not include any source code to eliminate tail-recursion (Section 4.2). Furthermore, LLVM completely lacks functionality for *partial* inlining, *partial* outlining or to represent HOFs at all. Still, LLVM's pendants are in total roughly 2.8 times more difficult to implement while it takes about 50 times more time to program.

	SLoC	Volume	Difficulty	Effort
`CloneFunction.cpp`	359	21298	107	2269693
`CodeExtractor.cpp`	523	34599	124	4287983
`InlineFunction.cpp`	526	32288	109	3511359
`LoopUnroll.cpp`	279	15393	80	1229623
total	1687	120421	207	24968883
`mangle.cpp`	132	6636	75	496757

Table 2. Source lines of code (SLoC) and Halstead numbers for LLVM's C++ implementations compared to Thorin's mangle implementation.

7. Discussion and Related Work

Graph-Based IRs and CPS. Graph-based IRs for imperative languages [like 5, 7, 22] are not designed for functional aspects and do not natively support HOFs and closures as Thorin does. On the other hand, higher-order CPS-based IRs used in compilers for functional languages use explicit scope nesting and an AST rather than a graph in contrast

[2] https://github.com/AnyDSL/impala
[3] http://www.rust-lang.org
[4] https://github.com/AnyDSL/thorin
[5] http://benchmarksgame.alioth.debian.org/
[6] https://code.google.com/p/aobench
[7] https://github.com/AnyDSL/benchmarks-impala
[8] SLoC were generated using David A. Wheeler's 'SLOCCount'; Halstead numbers were computed with `c3ms` [12].

to Thorin [1, 3, 26]. We discussed that graphs instead of names and implicit scope nesting simplify program transformations (Section 2 and 4). An alternative to CPS is A-normal form (ANF) [11] or a monadic language [21]. These languages do not use CPS but are in the spirit of CPS by binding each new temporary to a new name. However, as Kennedy [17] points out, such languages introduce problems not present in a faithful CPS presentation: For example, ANF needs a re-normalization phase after β-reduction.

SSA Form. SSA form was invented by Rosen et al. [24] and became popular after Cytron et al. [8] described an efficient algorithm for computing SSA form. As already outlined, SSA form is a restricted form of CPS. In fact, Impala exploits an SSA construction algorithm [6] to directly construct a Thorin program from the AST without further analyses like the computation of a dominance tree. The scope of a top-level function in CFF is akin to an SSA-form program because Kelsey's algorithm is only of syntactic nature (Section 5.1). However, in contrast to classic SSA programs Thorin programs can be higher-order. Even classic imperative languages would profit from a higher-order IR. It is often important to annotate code regions (for example, to mark a loop for parallelization). LLVM uses metadata to annotate code. This introduces many subtle problems. For instance, transformations must pay attention to not mistakenly destroy metadata designated for a different pass. HOFs solve this problem in a clean and type-safe way by wrapping the code region to be annotated in a HOF. Köster et al. [18] use this technique to annotate code regions for SIMD vectorization and GPU execution.

Lambda Lifting and Dropping. Lambda lifting was invented by Johnsson [15]. His algorithm uses currying in order to abstract free variables:

$$\lambda x.\ \ldots y \ldots \quad \Rightarrow \quad \lambda y.\lambda x.\ \ldots\ .$$

Danvy and Schultz [9] observe that we can directly append y to the parameter list in the case of first-order programs with n-ary functions:

$$\lambda(x).\ \ldots y \ldots \quad \Rightarrow \quad \lambda(x, y).\ \ldots\ .$$

They invented lambda dropping as reverse transformation to Johnsson's algorithm. When dealing with HOFs, their algorithm uses Johnsson's currying approach since in general a compiler cannot rewrite call sites of a function passed as argument to another function. As we want to eliminate closures, currying is not an option for us. After all, a curried function is not a returning function anymore (Section 5). For this reason, lambda mangling does not use currying to introduce or eliminate parameters. It rather produces one new generalized and/or specialized function with an updated signature which is at that point not connected to the rest of the program. It is in the responsibility of other passes to orchestrate mangling in a reasonable way and to connect a mangled function to the rest of the program properly. Due to Thorin's blockless representation we can fuse both algorithms into one simple recursive reconstruction algorithm which does not need the block floating/sinking pass of the original algorithms.

Static Argument Transformation. We are not the first to note that an argument/parameter pair of a recursive call is superfluous if the argument is just the parameter. The *static argument transformation* [10] identifies such recursive calls and eliminates them in the following way:

```
                              f: (a, b)
f: (a, b)                         letrec f': (b)
    ... use(a) ...     ⇒              ... use(a) ...
    f(a, x)                           f'(x)
                                  in  f'(b)
```

Note that `f` itself is not recursive anymore and, hence, is a potential candidate for inlining. For mutual recursive functions, we must find out which parameters do not change their values within an SCC. For this reason, we rather speak of static *parameters* than of *arguments*. However, applying the static argument transformation on a mutual recursive function still leaves the function recursive whereby further inlining becomes problematic. Our algorithm on the other hand considers all functions within one SCC simultaneously and, thus, can eliminate static parameters (Section 4.2).

Super-β inlining. Super-β inlining enables inlining of a closure call c with a function literal f. This transformation is only valid if (1) all applications of c are closures over f and (2) the environment at c is always equivalent to the one where the closure is captured. A control flow analysis (CFA) [19, 25] addresses the first condition. Δ-CFA [20] also takes the environment where a closure is captured into account. This allows aggressive inlining and, thus, aggressive closure elimination. The focus in our work is different. The presented lower2cff algorithm specializes (in contrast to inlining) HOFs even more aggressively. Consider the function `range` from Figure 1. Suppose, there are two call-sites of `range` which pass different function literals `g1` and `g2` to `range`'s parameter `f`:

```
range(..., g1, ...)
range(..., g2, ...)
```

A Δ-CFA discovers that inlining the closure call of `f` within `range`'s scope is not possible since condition (1) is violated. Thorin's lower2cff on the other hand, specializes each call-site and, hence, gets rid of the HOF `range` (Section 5.2).

Other Closure-Elimination Strategies. The SML/NJ compiler [3] allocates garbage-collected closures on the heap. An additional phase tries to find closures whose lifetime can be statically proven to be nested in a simple way. These closures can be allocated on the stack instead. Additionally, the compiler lambda-lifts functions that are not passed as argument to other functions. Then, these so-called *known* functions (as opposed to *escaping* functions) do not require a closure at all. An additional η-split phase splits functions that are used in a known as well as in an escaping context. Finally, SML/NJ's

inliner tries to decrease *escaping* contexts. Other compilers like Scala[9] also rely on inlining to eliminate explicit closures.

In summary, state-of-the-art functional compilers rely on *inline heuristics* to eliminate closures. Thorin *guarantees* to eliminate all closures in programs that are CFF-convertible. As future work we want to design a type system that captures this property in order to inform the programmer whether his higher-order program will need explicit closures.

8. Conclusions

We presented the functional, CPS-based IR Thorin. Thorin is equally suited to represent imperative as well as functional programs. Its novelty is that it does not use scope nesting to associate the use of a value with its definition, but is graph-based. We built on this property to devise *lambda mangling,* which generalizes partial in- and outlining. Lambda mangling serves as a building block for other important control-flow transformations, especially closure elimination. Finally, we presented a novel closure elimination algorithm on the basis of lambda mangling.

We evaluate Thorin experimentally using our research language Impala, a dialect of Rust, on *The Computer Benchmark Game*, a program suite to benchmark performance across several languages (and their implementations). Impala uses HOFs and closures to represent loop constructs in a generic way. Nevertheless, Thorin's closure elimination algorithm successfully removes all overhead of closure allocation on this benchmark set and produces programs that match the performance of C.

Acknowledgments

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the ECOUSS project as well as by the Intel Visual Computing Institute Saarbrücken.

References

[1] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. Orbit: an optimizing compiler for scheme. In *Symposium on Compiler Construction*, SIGPLAN, 1986.

[2] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 1998.

[3] A. W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier Science, 1985.

[5] M. Braun, S. Buchwald, and A. Zwinkau. Firm—a graph-based intermediate representation. Technical report, 2011.

[6] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In *CC*, 2013.

[7] C. N. Click and Jr. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 1991.

[9] O. Danvy and U. P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure, 2001. This is an extended version of Danvy's and Schultz' original paper of the same title which appeared at PEPM'97.

[10] A. L. de M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[12] C. H. Gonzalez and B. B. Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. In *HPCCC*, 2010.

[13] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977.

[14] M. R. Hansen and H. Rischel. *Introduction to Programming using SML*. Addison Wesley Longman, 1999.

[15] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, 1985.

[16] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. *SIGPLAN Not.*, 1995.

[17] A. Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.

[18] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek. Code refinement of stencil codes. *PPL*, 24, 2014.

[19] J. Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 2012.

[20] M. Might and O. Shivers. Environment analysis via Δ-CFA. In *POPL*, 2006.

[21] E. Moggi. Notions of computation and monads. *IC*, 1991.

[22] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *JVM*, 2001.

[23] S. Peyton Jones and S. Marlow. Secrets of the glasgow Haskell compiler inliner. *Journal of Functional Programming*, 2002.

[24] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. POPL, 1988.

[25] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[26] G. L. Steele, Jr. Rabbit: A compiler for scheme. Technical report, 1978.

[27] B. Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations*, pages 62–70, 1995.

[28] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 2000.

[9] http://magarciaepfl.github.io/scala