Shallow Embedding of DSLs via Online Partial Evaluation

Roland Leißa Klaas Boesche Sebastian Hack
Saarland University, Germany
{leissa, boesche, hack}@cs.uni-saarland.de

Richard Membarth Philipp Slusallek
German Research Center for Artificial Intelligence, Germany
{richard.membarth, philipp.slusallek}@dfki.de

Abstract

This paper investigates shallow embedding of DSLs by means of online partial evaluation. To this end, we present a novel online partial evaluator for continuation-passing style languages. We argue that it has, in contrast to prior work, a predictable termination policy that works well in practice. We present our approach formally using a continuation-passing variant of PCF and prove its termination properties. We evaluate our technique experimentally in the field of visual and high-performance computing and show that our evaluator produces highly specialized and efficient code for CPUs as well as GPUs that matches the performance of hand-tuned expert code.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Partial evaluation

Keywords DSL Embedding, Partial Evaluation, Continuation-Passing Style

1. Introduction

To achieve optimum performance, programs have to be transformed in a way that is beyond the scope of ordinary compiler optimizations. These transformations have two goals: First, exploit domain knowledge that is lost in the implementation and not accessible to the compiler. Second, utilize features of the target hardware architecture to improve performance (vectorization, memory hierarchy, etc.).

One way of achieving this performance is to create a domain-specific language (DSL) that provides language constructs to express domain knowledge, and a compiler that leverages this knowledge to produce highly optimized code for a specific architecture. A popular approach to implement a DSL is to *embed* the DSL into a host language H. One typically distinguishes between two different styles of embedding [15]:

Deep. The DSL program is represented as a data structure in the host program.

Consider Figure 1a. In the host language H, the programmer writes a program `pgen` that constructs the embedded program `e_spec`. Because `pgen` constructs the embedded program, it can also construct a version of the embedded program that is partially evaluated with respect to the inputs `s`. Then, an optimizer `opt` transforms `e_spec` to `e_opt` which is finally emitted to target code by `compile`. Note that `opt` as well as `compile` are written in H.

Deep embeddings allow for powerful, domain-specific optimizations [7, 31, 39] because the embedded program is available as a data structure. For the same reason, deep embeddings can accommodate *any* embedded language. In terms of programming experience, one drawback of deep embeddings is that the programmer actually writes a program generator instead of a program. Modern deep embedding frameworks alleviate this problem by "virtualizing" the host language [7]: Overloading reinterprets a part of the language constructs to not perform the actual computation but to construct a representation of that computation. This virtualization is often not entirely faithful and compromises the illusion of writing the embedded program in H in several ways: First, the overloading is not powerful enough to hide this construction entirely and leak implementation details of the embedding into the host language [27]. Second, the host language can usually not be virtualized entirely. Third, to reason about the embedded program (the result of `pgen`), the programmer ultimately has to understand how the generator works.

```
// code in host language
e_spec = pgen(s);
e_opt  = opt(e_spec);
compile(e_opt);
```
(a) Deep embedding

```
// code in the host language's compiler
e_spec = mix(e, s);
e_opt  = opt(e_spec);
compile(e_opt);
```
(b) Shallow embedding

Figure 1. Deep and shallow embeddings. The underlined term constitutes the embedded domain-specific program.

Shallow. The DSL constructs are defined by implementing their semantics in the host language directly.

Consider Figure 1b. The programmer *directly* writes the embedded program `e` in language H. To perform partial evaluation, shallow embedding needs a partial evaluator `mix` to be available in the compiler of H. Both functions, `opt` and `compile` are part of *the compiler* of H.

Like virtualization but unlike deep embedding, shallow embedding can accommodate only one embedded language: H itself. However, unlike virtualization, shallow embedding uses the entire language H. In contrast to deep embedding, shallow embedding cannot manipulate the embedded program because it is not available as a data structure. However, shallow embedding does not suffer from the programming experience problems that deep embedding does, because the programmer writes the embedded program directly and not a program generator. Nevertheless, only a few shallowly embedded high-performance DSLs (e.g. HIPA^{cc} [35] and SYCL [30]) exist. One reason is that, if no partial evaluator is available for H (which is usually the case), shallow embedding involves the unpleasant task of modifying an existing compiler.

1.1 Our Approach

In this paper, we present the continuation-passing style (CPS)-based language Impala together with a novel online partial evaluator. Impala enables shallow embedding without having to modify its compiler (usually). Embedding a DSL into Impala typically means that the domain-specific constructs are implemented as (higher-order) functions. These implementations essentially constitute a *tagless interpreter* [6]. We obtain the compiled DSL program by partially evaluating this interpreter with that program. The following

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3687-1/15/10...$15.00
http://dx.doi.org/10.1145/2814204.2814208

function constitutes such an "interpreter" to iterate several loop bodies in a fused manner:

```
fn fused_iterate(iterate: fn(fn(int)->()) -> (),
                 bodies: [fn(int)]) -> () {
    for i in iterate()
        for body in @each(bodies) { body(i) }
}
```

Specializing a call (via `@`) to `fused_iterate` with a function `range` and an array of bodies yields the desired fused loop[1]:

```
@fused_iterate(|body| range(a, b, body), bodies);
```

To generate hardware-specific code, the Impala compiler exposes hardware-specific paradigms through higher-order functions. These functions can be used to implement a certain DSL construct with respect to a specific kind of hardware. This way, we enable hardware-specific code generation without having to dig into Impala's compiler. In the example above, we could replace the argument `range` with the compiler-known function `vectorize` that strip-mines the loop and vectorizes the resulting innermost loop:

```
@fused_iterate(|body| vectorize(length, a, b, body), bodies);
```

Using partial evaluation (PE), we are limited to optimizations that can be expressed by *specializing* code. Optimizations that analyze and rewrite programs are not possible without modifying Impala's compiler. We argue that many of such optimizations can be expressed by proper abstractions in the style of `fused_iterate`. Other optimizations that cannot be expressed this way have to be implemented in Impala's compiler which has a sufficiently high-level IR [32] to facilitate this. For the DSLs we present in this paper, we did not have to modify Impala's compiler.

1.2 Continuations

To enable embedded DSLs to use non-trivial control flow (see Section 5.1 for an example), Impala features continuations as first-class citizens. Impala represents all control flow (including functions) as continuations. The following example shows a simple for-loop with unstructured control flow and its internal representation using CPS.

```
for i in range(a, b) {
    if i == 23
      continue()
    else if i == 42
      break()
    else
      /*...*/
}
/*next*/
```

⇒

```
let break: fn() = || /*next*/;
range(a, b,
  |i: int, continue: fn()| {
    if i == 23
      continue()
    else if i == 42
      break()
    else
      /*...*/
  }, break)
```

One important aspect of a partial evaluator is to determine where to resume partial evaluation after skipping code under a dynamic condition. Assume the partial evaluator wants to evaluate the call to `range` above. Furthermore, assume that `a` and `b` are dynamic, i.e. their values are not known at partial evaluation time. Because the partial evaluator cannot evaluate the condition of the `if i == 23`, it should skip the call to `range`. However, at which continuation shall partial evaluation be resumed?

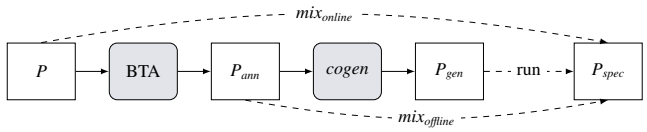
In a direct-style language a suitable resume point can be derived from the syntactic structure of the language, i.e. the statement after the skipped statement. In CPS, the "next instruction" is passed as a parameter and can be an arbitrary continuation (closure) that might not be syntactically "close". In the example above, the resume point is the **`break`** continuation passed to `range`. In this paper, we extend the notion of a post-dominator (a well-known concept in first-order control flow) to higher-order programs to derive suitable resume points.

1.3 Contributions

In summary, this paper makes the following contributions:

- We present a novel, pragmatic algorithm for online PE. We discuss our algorithm on a CPS-based variant of Plotkin's Programming Computable Functions (PCF) that captures the semantics of full as well as partial evaluation. We formally describe a termination property of our partial evaluator and prove our partial evaluator correct in that respect (Section 3).
- A crucial aspect of our PE algorithm is the computation of post-dominators in higher-order programs to designate resume points for PE. We present a novel control flow analysis (CFA) that locally computes useful post-dominators during PE (Section 4).
- We show how mapping to different hardware accelerators can be nicely expressed by higher-order functions. Our approach allows to weave in platform-specific mapping strategies such as executing code on a GPU or vectorizing code for a CPU by compiler-known higher-order functions. We demonstrate that our PE approach enables an efficient shallow embedding of high-performance DSLs for visual and high-performance computing in Impala (Section 5).

[1] Impala's syntax borrows from Rust. `|a| x` means $\lambda a.x$.




```
fn Pann(x:int, n:int) -> int {
  if n == 0 {
    1
  } else if n % 2 == 0 {
    let r = Pann(x, n/2)
    r * r
  } else
    x * Pann(x, n-1)
}
```

(a) P_{ann}

```
fn Pspec(x:int) -> int {
  let r = x*x; r*r
}
```

(b) P_{spec} with `Pgen(4)`

```
fn Pgen'(n:int) -> code {
  if n == 0 {
    "1"
  } else if n % 2 == 0 {
    let r:code = Pgen'(n/2);
    r + "*" + r
  } else
    "x*" + Pgen'(n-1)
}

fn Pgen(n:int) -> () {
  emit("fn Pspec(x:int) -> int {");
  emit(Pgen'(n));
  emit("}");
}
```

(c) P_{gen}

Figure 2. Partial evaluation and metaprogramming.

2. Background and Related Work

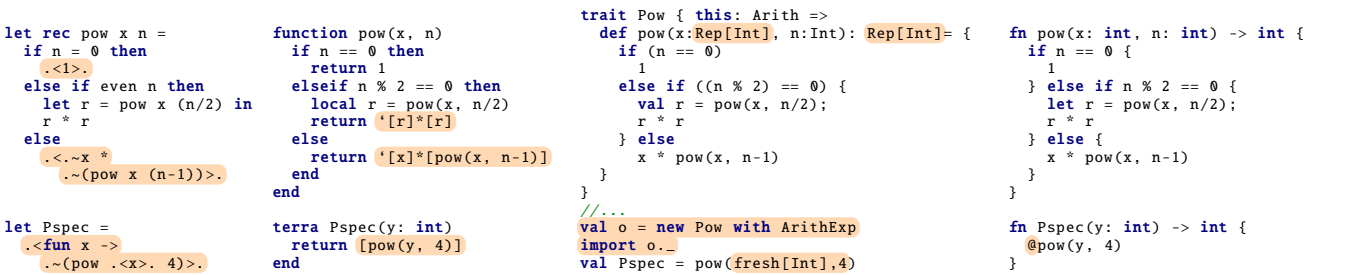
As running example to discuss prior work, we review how to specialize the power function to its exponent.

2.1 Partial Evaluation

In this paper we advocate *online* partial evaluation [9, 43, 44]. We directly specialize the source program P (Figure 3d) to the *specialized* or *residual* program P_{spec} (see Figure 2b). This corresponds to the first Futamura [14] projection: Specializing an interpreter P to an input program produces a compiled version of that program. The specializer is often called *mix* in literature.

Specializing the specializer for itself yields a compiler generator (*cogen*): a tool that converts an interpreter to a compiler (the third Futamura projection). For a long time it was unclear how *cogen* actually looks like and generating *cogen* via *mix* requires *mix* to be self-applicable which turned out to be hard in practice. Building a self-applicable evaluator becomes easier when separating the input program into static and dynamic parts: the *binding time*. In our example, a binding-time analysis (BTA) [23, 26] infers that everything which depends on `x` must stay dynamic and annotates the program accordingly (Figure 2a). Then, the specializer ($mix_{offline}$) runs on that annotated program as opposed to directly running the specializer (mix_{online}) on P. For this reason, this technique is called *offline* partial evaluation [3, 21, 23]. Glück [16] discusses a self-applicable *online* partial evaluator.

Birkedal and Welinder [2] discovered that hand-writing *cogen* is actually not more difficult than writing *mix*. In particular, a hand-written *cogen* does not require a bootstrapping process. Thus, *cogen* does not necessarily need to be written in the same language as P. Given the annotated program P_{ann}, *cogen* produces its *generating extension* P_{gen} (Figure 2c): All static parts of P_{ann} are copied over to P_{gen}. Dynamic parts are converted into a program that generates the specialized program P_{spec}. Thus, P_{gen} is parametric in P_{ann}'s



```
let rec pow x n =
  if n = 0 then
    .<1>.
  else if even n then
    let r = pow x (n/2) in
    r * r
  else
    .<.~x *
      .~(pow x (n-1))>.

let Pspec =
  .<fun x ->
    .~(pow .<x>. 4)>.
```

(a) MetaOCaml

```
function pow(x, n)
  if n == 0 then
    return 1
  elseif n % 2 == 0 then
    local r = pow(x, n/2)
    return `[r]*[r]
  else
    return `[x]*[pow(x, n-1)]
  end
end

terra Pspec(y: int)
  return [pow(y, 4)]
end
```

(b) Terra

```
trait Pow { this: Arith =>
  def pow(x:Rep[Int], n:Int): Rep[Int]= {
    if (n == 0)
      1
    else if ((n % 2) == 0) {
      val r = pow(x, n/2);
      r * r
    } else
      x * pow(x, n-1)
  }
}
//...
val o = new Pow with ArithExp
import o._
val Pspec = pow(fresh[Int],4)
```

(c) Scala/LMS

```
fn pow(x: int, n: int) -> int {
  if n == 0 {
    1
  } else if n % 2 == 0 {
    let r = pow(x, n/2);
    r * r
  } else {
    x * pow(x, n-1)
  }
}

fn Pspec(y: int) -> int {
  @pow(y, 4)
}
```

(d) Impala

Figure 3. Specializing the power function to its exponent with meta programming and partial evaluation.

static input. Running P_{gen} with a specific static input generates a program P_{spec}, which is parametric in P_{ann}'s dynamic input. For example, invoking `Pgen(4)` *generates* P_{spec} (Figure 2b). From a different point of view, *cogen* transforms the one-stage program P into a two-stage program.

The *cogen* approach is prone to the overapproximation of the static BTA. It has to produce *one* P_{gen} that must work for *every* static input of P. Thus, the BTA must always leave `d` dynamic in the following example:

```
fn f(a, b, d) { if a == b { d = 42 } /*next*/  }
```

An online evaluator, however, which specializes this function for `a` and `b` can exploit the case where `a == b` and set `d` to 42.

2.2 Metaprogramming

Metaprogramming allows the programmer to write a program that generates another program. In other words, the programmer manually implements the generating extension P_{gen}. For this reason, metaprograms conceptually look like the pseudocode in Figure 2c and the programmer can explicitly stage a DSL interpreter [10]. Many projects implement P_{gen} in a scripting language like Python. The script is invoked at build-time to generate P_{spec}—usually in a low-level language like C. As such scripts simply splice strings, the residual program may be ill-typed. Other approaches like C++ metaprogramming, Terra [13] (Figure 3b), (quasi-)quotation and macros in Scheme/Lisp or Racket [49] increase programmer productivity by incorporating metaprogramming facilities into the language, but still may construct an ill-typed residual program. MetaML [47] and MetaOCaml (Figure 3a) on the other hand, guarantee well-typedness of the residual program if the metaprogram is well-typed. With PE, well-typedness of the residual program comes for free because type checking is independently performed prior to specialization. All these metaprogramming approaches have in common that the stage is a feature of the syntax and, hence, in contrast to PE, it is not possible to write a function which is polymorphic in the binding time of its parameters. Dynamic staging [11] tackles this problem by introducing the stage as a first-class citizen to the language at the cost of an unsound type system.

2.3 DSL Embedding

Carette et al. [6] and Hudak [19] lay the foundation of embedding a typed language by ordinary functions instead of object terms. Hofer et al. [17] picked up this idea and carried it to the Scala world while emphasizing modularity and the ability to abstract over and compose semantic aspects. Rompf and Odersky [41] coined the term lightweight modular staging (LMS) and paved the way for performance-oriented embedded DSLs [7] like OptiML [46] or Liszt [12]. LMS does not rely on explicit staging capabilities of the *host* language Scala. Instead, executing the host program constructs a second domain-specific program representation like Delite [5] (*deep* embedding). Values of type `T` are wrapped into a type operator `Rep[T]` to represent values which are computed in a deferred stage. Lancet [42] is an online partial evaluator for Java bytecode that serves as a front-end for LMS, as an alternative to explicit programming with `Rep` types. Array Building Blocks [33] and Halide [39] leverage a similar staging mechanism to construct the actual program representation with C++ as host language. HIPA^{cc} [35] and SYCL [30] on the other hand, are *shallowly* embedded DSLs in C++. They rely on compiler plug-ins which directly manipulate the program representation to achieve performance.

Comparable to other explicit metaprogramming techniques, LMS essentially requires the programmer to write the generating extension. However, via overloading and type inference the staged program is somewhere between P and P_{gen} (Figure 3c). As the stage is encoded in the type system, Scala's type inference works akin to a local BTA [37]. For example, `n % 2` is of type `Int`. Thus, the expression is executed when the host program runs. But since `r` is of type `Rep[Int]`, executing `r * r` results in a residual program containing a multiplication. The implementation of *cogen* lies in LMS' library which implements `a * b` for `Rep[Int]`. The downside of this approach is that for data types unknown to LMS, the programmer must implement these overloads ("*cogen* for these types") himself. To some extent, this limitation can be alleviated via parametric polymorphism [36] at the cost of introducing type variables for each desired staging combination:

```
def f[I[_]](i: I[Int])= { /*...*/ }
```

Jovanović et al. [27] present a technique based on Scala macros to generate a deep embedding from a shallow one. Our approach on the other hand uses shallow embedding and PE to achieve the performance of a deeply embedded DSL. Finally, LMS may suffer from induced divergence while our PE technique may only induce divergence when run-annotating recursive calls (see below and Section 3).

2.4 Divergence in Partial Evaluation

Katz and Weise [29] distinguish three classes of divergence that may occur during PE:

True divergence: If the full evaluation of the program does not terminate for some inputs, PE might also not terminate.

Hidden divergence: A program may contain unreachable code that is divergent. Partially evaluating this divergent code may cause the partial evaluator to diverge.

Induced divergence: The partial evaluator diverges although neither true nor hidden divergences are present but because the evaluator is too greedy. Consider the *counting loop problem* [3]:

```
fn count(i: int, N: int) -> int {
  if i < N { count(i+1, N) } else { i }
}
```

An aggressive partial evaluator might infinitely expand `count(0, N)`, although full evaluation terminates for every $N \geq 0$.

It is easy to avoid all forms of divergence if recursive calls are not specialized [51]. Hybrid PE [44] on the other hand, does not give any termination guarantees. For this reason, Hybrid PE uses annotations similar to our approach.

A well-known technique [1, 9, 14] to avoid at least obvious endless recursions, is to memoize each specialized call site. If the evaluator is about to specialize an already cached call, it re-uses a

call to the cached version instead. This technique does not prevent the counting loop problem.

LMS has special behavior for a `while` loop: If the termination condition is of type `Boolean`, the loop will be run when the host program runs; if it is of type `Rep[Boolean]`, LMS will construct a residual loop. This approach only works since Scala does not provide `continue`. LMS also leverages the aforementioned memoization technique for recursive calls. This has the effect that a counting `while`-loop with a dynamic conditional terminates when the host program runs, whereas a recursive implementation with an `Int` counter and a `Rep[Int]` bound diverges.

Both Similix [25] and Schism [8] are offline evaluators using BTA. These evaluators will not evaluate a cycle, if the condition which breaks the cycle remains dynamic. On the one hand, this is slightly more aggressive than our approach because our approach will also jump over an acyclic, dynamic conditional. On the other hand, both evaluators suffer from the inherent imprecisions caused by the BTA (see above). Furthermore, both evaluators depend on a CFA [45] which leads to further imprecisions: As argued in Section 4, in our setting an *on-the-fly* CFA is more precise than a CFA which runs *once beforehand* because evaluating the program brings full context-sensitivity for free. Lastly, we argue that a single rule—dynamic branches are skipped—is easier to understand for the programmer.

Other more complex termination heuristics, like monitoring the argument sizes of recursive calls have been applied in the past [22, 24]. We consider such heuristics hard to understand and account for by the programmer.

3. Partial Evaluation

In this section we first formally discuss our PE technique by studying the language λ^{cps} (Figure 4). Then, we discuss how to embed and guide the partial evaluator from within a program and how this affects termination.

3.1 The CPS-Based, Simply-Typed Lambda Calculus

The syntax of λ^{cps} is similar to the simply-typed lambda calculus with an additional fixpoint operator (called *letrec*) to allow recursion. Furthermore, λ^{cps} uses CPS for function abstractions and applications. Thus, λ^{cps} is a CPS version of Plotkin's PCF [38]. CPS introduces mainly two peculiarities compared to Plotkin's original PCF:

1. As functions do not return in CPS, we do not allow functions to return a value. Instead, the actual function is formed by a *body*.
2. For the same reason, we cannot curry functions. Hence, we allow arbitrarily long parameter lists.

For the sake of presentation, we restrict the arithmetic to integer *literals* and *addition*. An *ifz* body tests the first expression—the *condition*—for zero. If the test yields true, evaluation will progress with the second expression as continuation or with the third one otherwise. Program execution ends with result e upon reaching an *exit* body `exit` e. A *skip* and *skipping* body is considered as expanded syntax, which only appears as intermediate results during evaluation. It cannot be used by the programmer directly.

We denote the expression language (using e as start symbol) by $\mathcal{L}(e)$. We sometimes refer to a body as *program* if we want to stress that a body may contain many sub-bodies. The syntactic structure of a language induces a syntax tree. We write $a \preceq b$ if we require a to be a subtree of b. Parameters and identifiers bound in *letrecs* range over x. We require all names in the program to be unique in order to circumvent name capture in the rules. We use the common notation $\overline{a}$ to denote a list $a_1, \ldots, a_n$.

In examples we often elide type annotations as the reader can easily infer them from their surrounding context. As syntactic sugar, we use `where` to bind non-recursive functions and `whererec` to bind recursive ones. Finally, we sometimes use additional features in λ^{cps} examples like boolean types or more sophisticated branch constructs.

3.1.1 Typing

As functions do not return, typing rules checking *bodies* do not yield a type. A function type $\mathtt{fn}(\overline{t})$ does not include a return type for the same reason. Apart from that, rules are standard.

Definition 1 (Expression Normal Form)**.** An expression e is in normal form iff $[\![e]\!] = e$. We write $\mathsf{nf}\ e$ if we require e to be in normal form.

Definition 2 (Exits)**.** Let $exits(b) \coloneqq \{\mathtt{exit}\ e \mid \mathtt{exit}\ e \preceq b\}$ be the set of exits in the body b.

Definition 3 (Well-Typedness)**.** We call a body b *well-typed* under Γ iff $\Gamma \vdash b$ holds.

Definition 4 (Constant)**.** We call an expression e a *constant* iff e is a normal form and $\vdash e : t$ for some type t.

Remark. These are all literals and functions without free variables.

Definition 5 (Valid Configuration)**.** Let $f \coloneqq \lambda\overline{x : t}.b$ be a function constant. We call an argument list $\overline{c}$ of constants a valid configuration for f iff the application $f(\overline{c})$ is well-typed. We denote $C(f)$ as the set of all valid configurations for f.

3.1.2 Semantics

In the classic lambda calculus each argument to a function evaluates to a constant. Then, the function is substituted by its body while replacing all occurrences of the function's parameters with its corresponding arguments. During PE however, arguments may not necessarily be constants.

Expression semantics. The function $[\![e]\!]$ evaluates an expression. If two literals occur in an *addition* they will be folded (operator $\oplus$ denotes the arithmetic addition as opposed to the syntactic terminal +). Other additions are evaluated by recursively applying evaluation. Other expressions yield identity. For example, the expression $x + (1 + 2)$ reduces to $x + 3$.

Lemma 1 (Strong Normalization Property of Expressions)**.** *Evaluating an expression $[\![e]\!]$ is strongly normalizing, that is, every $[\![e]\!]$ eventually terminates with an expression in normal form.*

Proof. Evaluation of *variables*, *literals* and *functions* is strongly normalizing by definition. By induction we show that *addition* is strongly normalizing, too. □

Body semantics. The semantics of λ^{cps} works for partial as well as for full evaluation. In contrast to expressions, body semantics is not strongly normalizing due to *letrecs*. On this account we use a small-step semantics for body evaluation rules of the form $b \Rightarrow b'$. We read: A body b evaluates in one step to body b'. We write $b \Rightarrow^n b'$ in order to specify that b evaluates to b' in n steps, and $b \Rightarrow^* b'$ to specify that b evaluates in finitely many steps to b'.

Definition 6 (Body Normal Form)**.** A body b is in normal form iff $\nexists b' : b \Rightarrow b'$. We write $\mathsf{nf}\ b$ if we require b to be in normal form.

A function application first reduces its callee e_λ and arguments to normal form (E-App). The requirement $(e_\lambda, \overline{e}) \neq (e'_\lambda, \overline{e'})$ ensures that rules are deterministic; E-App will only trigger if at least the callee of the application or one of its arguments are not yet in normal form. If the callee and the arguments are in normal form, E-App$_\lambda$ will handle applications with a known function as callee. It will reduce to the function's body while substituting all parameters with their arguments. If the callee is unknown, the application reduces to the artificial *skip* body (E-App$_{skip}$).

An *ifz* first evaluates its arguments (E-Ifz) and then selects either its true or false continuation if possible (E-Ifz$_\top$ and E-Ifz$_\bot$). Similar to E-App, rule E-Ifz only triggers if at least one of its expressions are

$$
\begin{array}{rrll}
\text{type} & t ::= & \texttt{int} \mid \texttt{fn}(\overline{t}) & \text{(integer/function type)} \\
\text{expression} & e ::= & x \mid l \in \mathbb{Z} \mid e + e & \text{(variable/literal/addition)} \\
& \mid & \lambda \overline{x : t}.b & \text{(abstraction)} \\
\text{body} & b ::= & e(\overline{e}) & \text{(application)} \\
& \mid & \texttt{ifz}(e, e, e) & \text{(zero test)} \\
& \mid & \texttt{letrec}\ x = \lambda \overline{x : t}.b\ \texttt{in}\ b & \text{(letrec)} \\
& \mid & \texttt{exit}\ e & \text{(exit)} \\
& \mid & \texttt{skip}\ b & \text{(skip)} \\
& \mid & \texttt{skipping}\ b : b : b & \text{(skipping)}
\end{array}
$$

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

$$\text{T-Var}\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{T-Add}\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\text{T-Int}\frac{}{\Gamma \vdash l : \texttt{int}} \qquad \text{T-Fun}\frac{\Gamma, \overline{x : t} \vdash b}{\Gamma \vdash \lambda \overline{x : t}.b : \texttt{fn}(\overline{t})}$$

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

$$\text{T-App}\frac{\Gamma \vdash e_\lambda : \texttt{fn}(\overline{t}) \quad \overline{\Gamma \vdash e : t}}{\Gamma \vdash e_\lambda(\overline{e})}$$

$$\text{T-Ifz}\frac{\Gamma \vdash e_c : \texttt{int} \quad \Gamma \vdash e_\top : \texttt{fn}() \quad \Gamma \vdash e_\bot : \texttt{fn}()}{\Gamma \vdash \texttt{ifz}(e_c, e_\top, e_\bot)}$$

$$\text{T-Letrec}\frac{\Gamma, x_\lambda : \texttt{fn}(\overline{t}) \vdash \lambda \overline{x : t}.b : \texttt{fn}(\overline{t}) \quad \Gamma, x_\lambda : \texttt{fn}(\overline{t}) \vdash b_{in}}{\Gamma \vdash \texttt{letrec}\ x_\lambda = \lambda \overline{x : t}.b\ \texttt{in}\ b_{in}}$$

$$\text{T-Exit}\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{exit}\ e} \qquad \text{T-Skip}\frac{\Gamma \vdash b}{\Gamma \vdash \texttt{skip}\ b}$$

$$\text{T-Skipping}\frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2 \quad \Gamma \vdash b_3}{\Gamma \vdash \texttt{skipping}\ b_1 : b_2 : b_3}$$

Expression: $[\![x]\!] \coloneqq x \qquad [\![l]\!] \coloneqq l \qquad [\![\lambda \overline{x : t}.b]\!] \coloneqq \lambda \overline{x : t}.b \qquad \boxed{[\![\cdot]\!] : \mathcal{L}(e) \to \mathcal{L}(e)}$

$$[\![e_1 + e_2]\!] \coloneqq \begin{cases} l_1 \oplus l_2 & \text{if } [\![e_1]\!] = l_1 \wedge [\![e_2]\!] = l_2 \\ [\![e_1]\!] + [\![e_2]\!] & \text{otherwise} \end{cases}$$

Body: $\boxed{b \Rightarrow b'}$

$$\text{E-App}\frac{[\![e_\lambda]\!] = e'_\lambda \quad \overline{[\![e]\!] = e'} \quad (e_\lambda, \overline{e}) \neq (e'_\lambda, \overline{e'})}{e_\lambda(\overline{e}) \Rightarrow e'_\lambda(\overline{e'})}$$

$$\text{E-App}_\lambda\frac{\overline{\mathsf{nf}\ e}}{\lambda \overline{x : t}.b\ (\overline{e}) \Rightarrow [\overline{e/x}]b} \qquad \text{E-App}_{skip}\frac{\mathsf{nf}\ e_\lambda \quad \overline{\mathsf{nf}\ e} \quad e_\lambda \neq \lambda \overline{x : t}.b}{e_\lambda(\overline{e}) \Rightarrow \texttt{skip}\ e_\lambda(\overline{e})}$$

$$\text{E-Ifz}\frac{[\![e_c]\!] = e'_c \quad [\![e_\top]\!] = e'_\top \quad [\![e_\bot]\!] = e'_\bot \quad (e_c, e_\top, e_\bot) \neq (e'_c, e'_\top, e'_\bot)}{\texttt{ifz}(e_c, e_\top, e_\bot) \Rightarrow \texttt{ifz}(e'_c, e'_\top, e'_\bot)}$$

$$\text{E-Ifz}_\top\frac{\mathsf{nf}\ e_\top \quad \mathsf{nf}\ e_\bot}{\texttt{ifz}(0, e_\top, e_\bot) \Rightarrow e_\top()} \qquad \text{E-Ifz}_\bot\frac{l \in \mathbb{Z} \quad l \neq 0 \quad \mathsf{nf}\ e_\top \quad \mathsf{nf}\ e_\bot}{\texttt{ifz}(l, e_\top, e_\bot) \Rightarrow e_\bot()}$$

$$\text{E-If}_{skip}\frac{\mathsf{nf}\ e_c \quad \mathsf{nf}\ e_\top \quad \mathsf{nf}\ e_\bot \quad e_c \notin \mathbb{Z}}{\texttt{ifz}(e_c, e_\top, e_\bot) \Rightarrow \texttt{skip}\ \texttt{ifz}(e_c, e_\top, e_\bot)}$$

$$\text{E-Letrec}\frac{}{\texttt{letrec}\ x_\lambda = \lambda \overline{x : t}.b\ \texttt{in}\ b_{in} \Rightarrow [\lambda \overline{x : t}.\texttt{letrec}\ x_\lambda = \lambda \overline{x : t}.b\ \texttt{in}\ b/x_\lambda]b_{in}}$$

$$\text{E-Exit}\frac{[\![e]\!] = e' \quad e \neq e'}{\texttt{exit}\ e \Rightarrow \texttt{exit}\ e'} \qquad \text{E-Skip}\frac{postdom(b) = p}{\texttt{skip}\ b \Rightarrow \texttt{skipping}\ b : p : p}$$

$$\text{E-Skipping}\frac{p' \Rightarrow p''}{\texttt{skipping}\ b : p : p' \Rightarrow \texttt{skipping}\ b : p : p''}$$

$$\text{E-Skipping}_{\text{nf}}\frac{\mathsf{nf}\ p'}{\texttt{skipping}\ b : p : p' \Rightarrow [p'/p]b}$$

Figure 4. Syntax, Typing and Semantics of λ^{cps}.

not yet in normal form. In the case that the condition's normal form is not a literal, the body reduces to the artificial *skip* body (E-Ifz$_{skip}$).

A *letrec* implements recursion (E-Letrec). An *exit* evaluates its expression to terminate evaluation (E-Exit).

Before discussing the artificial *skip* and *skipping* body, we have to define the post-dominators of a body.

Definition 7 (Lambda Lifting)**.** Let $X(b) \coloneqq \overline{x : t}$ be the list of free variables in a well-typed body b under $X(b)$. Then $\lambda X(b).b$ is the lambda-lifted function [20] of b. As shorthand, we write f_b.

Definition 8 (Post-Dominator)**.** Let b be a well-typed body under $X(b)$. We call p a *post-dominator* of b iff $p = \bot$ or all possible finite evaluations from b to any exit visits $p \preceq b$:

$$\forall x \in exits(b) \colon \forall \overline{c} \in C(f_b) \colon \quad f_b(\overline{c}) \Rightarrow^* x \quad \to \quad f_b(\overline{c}) \Rightarrow^* p\ .$$

Remark. Hence, $\bot$ designates the point in execution after all *exits*(b) and is always a valid post-dominator—even for diverging programs. This is a common trick in compilers to implement a sane post-dominance analysis for programs with no or multiple exits.

Rule E-Skip uses a function *postdom*(b) to get a post-dominator p of b. At this point, we are not interested in *how* post-dominators are computed. We discuss this in Section 4. We merely require *postdom*(b) to exist and to compute a *valid* but not necessarily the *immediate* post-dominator of b. Using p, *skip* evaluates to a *skipping* body while memorizing b and p. The third body is also set to p. Evaluation progresses there (E-Skipping) until a normal form is found. Then, the original post-dominator p is replaced by that normal form in the original body b (E-Skipping$_{\text{nf}}$).

Lemma 2 (Type-Safety)**.**

Progress: *If* $\Gamma \vdash b$ *then either* $b = \texttt{exit}\ l$ *or* $\neg \mathsf{nf}\ b$.
Preservation: *If* $\Gamma \vdash b$ *and* $b \Rightarrow b'$ *then* $\Gamma \vdash b'$.

Proof. By induction on the rules and Lemma 1. □

Remark. Note that $\vdash b$ implies that b does not have free variables.

3.2 Full and Partial Evaluation

If we evaluate a program with constants as arguments, we will never have to handle the case that an expression does not reduce to a constant. In particular, we will never trigger E-App$_{skip}$ nor E-Ifz$_{skip}$. This means, we will always know a body to continue evaluation.

Lemma 3 (Full Evaluation)**.** *Let f be a function constant. Evaluating $f(\overline{c})$ where $\overline{c} \in C(f)$ never triggers E-App$_{skip}$ nor E-Ifz$_{skip}$.*

Proof. For the proof of Lemma 2 we have shown that all expressions collapse to constants during evaluation. Thus, the preconditions for E-App$_{skip}$ or E-Ifz$_{skip}$ will never trigger. □

If a program does not contain any *letrecs*, evaluation will always terminate.

Lemma 4 (Termination without Letrec)**.** *A function constant not containing* letrecs *terminates for all inputs.*

Proof sketch. Tait [48] proved that the simply-typed lambda calculus is strongly normalizing. If we remove *letrecs* from λ^{cps}, the resulting language is analogously strongly normalizing. □

Remark. In particular, it is not possible to create a fixed-point operator like the Y combinator as in the untyped lambda calculus without relying on *letrec*. This also means, that all potential loops in a λ^{cps} program are syntactically recognizable by its *letrecs*. A λ^{cps} program cannot have other causes for divergence.

Now, we study the evaluation of a program b which still contains free variables. If we lambda-lift all free variables into a function f_b, all variables in f_b will be bound and we can perform a full evaluation (Lemma 3). If we perform n steps on b we will partially evaluate b and obtain b'. If we lambda-lift b' we can perform a full evaluation on that program. The following theorem states that both functions still compute the same result:

Theorem 1 (Correctness)**.** *Let b be a well-typed body under $X(b)$. We define $f' := \lambda X(b).b'$ with $b \Rightarrow^n b'$ for some $n \in \mathbb{N}$. If $f_b(\overline{c})$ terminates with result l so does $f'(\overline{c})$:*

$$\forall \overline{c} \in C(f): \quad f_b(\overline{c}) \Rightarrow^* \texttt{exit}\ l \quad \rightarrow \quad f'(\overline{c}) \Rightarrow^* \texttt{exit}\ l\,.$$

Remark. Note that f' uses the free variables of b and not b'. This is because PE might eliminate some free variables but we would like both f_b and f' to have the same signature.

Proof sketch. By PCF being confluent [18]. □

In order to prove that PE does not induce divergence, we have to show that PE will always obtain a normal form in finitely many steps, if the original program terminates for at least one configuration.

Theorem 2 (Termination Guarantee)**.** *Let b be a well-typed body under $X(b)$. If $f_b(\overline{c})$ terminates for some valid configuration $\overline{c}$ of f_b, partially evaluating b terminates to a normal form b' in finitely many steps:*

$$(\exists \overline{c} \in C(f_b): \ f_b(\overline{c}) \Rightarrow^* \texttt{exit}\ l) \quad \rightarrow \quad b \Rightarrow^* \mathsf{nf}\ b'\,.$$

Proof. Let i be the number of times rules E-App$_{skip}$ or E-Ifz$_{skip}$ are triggered during evaluation. We prove the theorem by induction on i.

By Lemma 3 evaluating $f_b(\overline{c}) \Rightarrow^*$ `exit` l never triggers E-App$_{skip}$ nor E-Ifz$_{skip}$. As an inductive base case, we presume that $b \Rightarrow^n b'$ does neither trigger E-App$_{skip}$ nor E-Ifz$_{skip}$ for any n. This implies that both evaluations $f_b(\overline{c}) \Rightarrow^*$ `exit` l and $b \Rightarrow^n b'$ will trace through the same functions till b'. Merely, expressions which do not contribute to control flow might not be constants. Hence, there exists an n such that $b' =$ `exit` e.

For the inductive step we have to show that triggering E-App$_{skip}$ or E-Ifz$_{skip}$ $i+1$ times still leads to a normal form. Both E-App$_{skip}$ and E-Ifz$_{skip}$ trigger E-Skip. By Definition 8 *postdom*(b) either retrieves $\bot$ or a body $p \preceq b$ which preserves b's termination behavior. By the induction hypothesis evaluating p along E-Skipping will finally trigger E-Skipping$_{\mathrm{nf}}$ if the original program terminates. □

3.3 Run and Halt Annotations

Until now, we studied partial evaluation of a whole program. In practice, the programmer only wants to specialize certain parts of the program while other parts should be excluded from specialization (Section 5). The compiler searches the program for annotations from top to bottom. Assume the compiler reaches an annotation as in the following *run*-annotated Impala snippet:

```
@f(args);          ⇒    λ().  @f(args, k) where
/*next*/;                k = λ(). /*next*/
```

Its λ^{cps} translation is presented on the right-hand side. During the search for annotations, all *letrecs* are substituted in—as in rule E-Letrec. In practice, we perform the substitution on demand when a free variable bound by a *letrec* is encountered. The *run* annotation `@` then causes `f(args, k)` to be specialized by triggering rule E-App.

As PE should only run until a continuation (e.g. `k`) outside the body is called, all remaining free variables, together with any **`exit`**, must be considered local *exits* for the purpose of evaluation and the definition of post-dominators in particular. Similarly, a *halt* annotation `$g(args, l)` causes the evaluator to stop specialization at that point and resume at `l`.

Run annotations impact the termination of PE. If a *run*-annotated code block is unreachable, the partial evaluator might be subject to hidden divergence:

```
if i_am_always_false { @i_will_not_terminate(42) }
```

Under certain circumstances, run annotations might induce divergence. Reconsider the function `count` from Section 2.4 where the recursive call is annotated with `@`. First, let us convert the function to λ^{cps}:

```
fn count(i: int, N: int) -> int {
  if i < N {
    @count(i+1, N)
  } else {
    i
  }
}
```

⇒

```
letrec count = λ(i, N, ret).
  if(i<N, T, F) where
    T = λ().
      @count(i+1, N, ret)
    F = λ(). ret(i)
in /*...*/
```

After substitution of `count` and application of E-App the following program results:

```
letrec count = λ(i, N, ret).
  if(i<N, T, F) where
    T = λ(). if(i+1<N, T', F') where
        T' = λ(). @count(i+2, N, ret)
        F' = λ(). ret(i)
    F = λ(). ret(i)
in /*...*/
```

This program contains a new call `@count`. After PE has terminated, the compiler looks for the next run annotation and stumbles upon that `@count`. Consequently, rule E-App is triggered again causing the compiler to diverge.

This situation only emerged because a *recursive call* has been annotated. In particular, an annotated call to a function bound via **`letrec`** *within* its definition like `@count` in the example above. A call to a recursive function outside its definition is *not* recursive and, thus, does *not* induce divergence. Hence, annotating all other calls to `count`, like `@count(0, N, RET)`, is not problematic. We want to stress that a single PE run is already Turing-complete. For this reason, we can completely fold `@ackermann(3, 4)` to `125`. As outlined above, this call is *not* recursive.

By Lemma 4 we can statically over-approximate *all* recursions. Thus, the compiler can warn the programmer about potentially dangerous run annotations. Compilers for languages with implicit recursive bindings (like Impala) must perform a loop analysis [40] to infer that information.

4. Local Control-Flow Analysis

So far, we assumed that the *postdom* function in rule E-Skip will obey Definition 8. The prerequisite for computing post-dominance is a control flow graph (CFG). If the input program is of first order, functions cannot be passed as arguments to other functions. Hence, all callees are statically known and we can directly construct a CFG from that program.

Higher-order programs may contain calls to parameters. In order to compute a CFG in this setting, we need to statically know which functions may actually reach a parameter at runtime. A k-CFA [45] computes this information for calling contexts of length k. With increasing k the analysis becomes more costly but also more precise. Hence, a 0-CFA is context-insensitive, leading to imprecisions as function arguments are merged from all calling contexts.

One possibility would be to apply a k-CFA prior to partial evaluation in order to determine an appropriate post-dominator in the context of a dynamic branch. However, for calls deeper than k contexts, the analysis information becomes imprecise. The computed post-dominator might then lie closer to the *exits*, causing the evaluator to skip more code than necessary. The programmer would have a hard time to understand, track and work around such imprecisions.

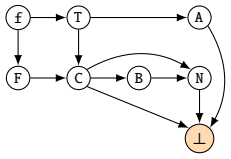
For this reason, our partial evaluator follows a different approach. As long as the evaluator does not hit a dynamic branch (E-Ifz$_{skip}$) or call (E-App$_{skip}$), the evaluator does not need any post-dominance information. Until such a point, evaluation has expanded all calls, resulting in full context-sensitivity. Whenever the evaluator needs to apply rule E-Skip it runs a *local*, *symbolic*, and *partially context-sensitive* CFA. This enables construction of a CFG to compute a post-dominator.

The CFA we employ is *local* in the sense that it starts at the run-annotation being partially evaluated and only analyzes functions currently declared within that scope. We call these functions *inside functions*. We call functions declared outside the scope, in particular functions defined in other translation units or intrinsic higher-order functions such as `nvvm` (Section 5), *outside functions*. Suppose PE of program `start` to `end` in Figure 5a meets a dynamic branch in `f`. Function `out` and higher-order parameter `end` are not defined within `start`'s scope and are thus outside functions. All other functions are inside.

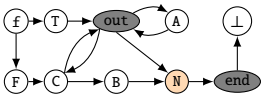
To precisely deal with references to outside functions or free variables, i.e. parameters of an outside function, the CFA tracks

```
start = λ(/*...*/, end).
  /* program already evaluated */
  f = λ(). ifz(/*dynamic*/, T, F) where
    F = λ(). C(B)
    T = λ(). out(A, C) where
      A = λ(pa: fn(int)). pa(42)
      B = λ(pb: fn()). pb()
      C = λ(pc: fn(fn()∗)). pc(N) where
        N = λ(). end()
```

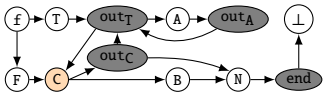
(a) Higher-order λ^{cps} program



(b) Non-symbolic CFG



(c) context-insensitive CFG



(d) Partially context-sensitive CFG

Figure 5. PE from `start` to `end` hits a dynamic branch in `f`.

these *symbolically* (see below). The CFA handles inside functions context-insensitively. This mirrors the intuition a programmer has about control flow and the bodies skipped by PE when encountering a dynamic branch. For example, PE would skip to the point after a dynamic `if` or `while` construct. For outside functions, the CFA uses one level of context (see below).

If the CFA did not handle free variables and outside functions symbolically, it would at best construct the CFG in Figure 5b. Propagation from `F` through `C` yields the value `N` for parameter `pb` which is precise. However, the CFA did not propagate outside functions and had to propagate $\bot$ to the parameters of the higher-order functions `A` and `C`. This would result in the post-dominator $\bot$, the unique local exit node in the CFG (see Definition 8). Evaluation would then skip the remainder of the scope.

Symbolic CFA. The call to `out` may pass other functions to the inside higher-order function `C` it receives as argument. For this reason, we let the call to `out` symbolically represent any function reachable from it. The CFA then determines the values `B` and `out` for parameter `pc`. Similarly, the CFA obtains `out` for `pa`.

The CFA never tracks values for the parameters of an outside function and does not propagate arguments of a call to it. Instead, the CFA represents control flowing through an outside function with a symbolic value. In the example, `T` calls `out`, which in turn may call `C`. The analysis propagates the symbolic value `out` to `C`'s parameter `pc`, and thus, `C` may jump back to `out`.

Merging control flow over all contexts for `out` leads to an unacceptable loss in precision. The CFG in Figure 5c has only a single node for `out` with edges to nodes `A`,`C` and `N` as these are the arguments to any call to `out`. This yields the imprecise post-dominator `N`.

Partially context-sensitive CFA. Instead of merging the arguments over all calls when constructing the CFG, we give one level of context to outside functions. The CFA creates a new symbolic value for each outside function within the calling context of an inside function. This allows the CFG to distinguish control flow from multiple calls to an outside function to their function arguments. The CFG in Figure 5d thus separates the calls to `out` in `T`, `A` and `C` and their different arguments. The edge from $\texttt{out}_\texttt{C}$ to `N` and its absence from other `out` shows the increase in precision. The post-dominator computed for `f` is then `C`.

Figure 5d also shows that we need to add edges from context-sensitive nodes $\texttt{out}_\texttt{C}$ and $\texttt{out}_\texttt{A}$ to the node $\texttt{out}_\texttt{T}$. These edges are required because any function passed to an inside function from outside, might also call any other function leaked to the outside function previously.

The CFG approximation is safe for computing post-dominators because the only way for an outside function `out` to return to the scope is via functions passed to it. The CFG does not model control-flow for diverging or non-returning `out`. In these cases, however, the evaluator cannot guarantee termination anyway (see Theorem 2) and continuing evaluation from the computed post-dominator remains sound for returning cases.

```
fn iterate(fld: Field, body: fn(int, int) -> ()) -> () {
  let vector_length = 8;
  for y in range(0, fld.rows)
    vectorize(vector_length, 0, fld.cols, |x| body(x, y));
}
```

(a) Iterator implementation for SIMD hardware

```
fn iterate(fld: Field, body: fn(int, int) -> ()) -> () {
  let grid  = (fld.cols, fld.rows, 1);
  let block = (128, 1, 1);
  nvvm(grid, block, || {
    let x = tid_x()+ntid_x()*ctaid_x();
    let y = tid_y()+ntid_y()*ctaid_y();
    body(x, y);
  });
}
```

(b) Iterator implementation for NVIDIA GPUs

Figure 6. Different implementations for `iterate`.

5. Applications and Evaluation

The Impala[2] compiler translates the source program into Thorin [32]: a functional intermediate representation (IR) similar to λ^{cps}. Partial evaluation and other optimizations are performed at that level. Finally, the compiler either translates to C/CUDA/OpenCL or LLVM/SPIR/NVVM (see below).

Mapping Algorithms to Different Architectures. In order to abstract from specific target platforms, Impala provides intrinsic higher-order functions. For example, invoking the following function vectorizes [28] `body` for SIMD width `L` and creates an appropriate loop from `a` to `b`:

```
fn vectorize(L: int, a: int, b: int, body: fn(int) -> ()) -> ();
```

Likewise, invoking the following function causes `body` to be executed via NVVM on an NVIDIA GPU:

```
fn nvvm(grid: (int, int, int), block: (int, int, int),
        body: fn() -> ()) -> ();
```

The execution runs in parallel by the threads defined by `grid` with the given `blocking`. Similarly, Impala supports code generation for CUDA, OpenCL, and SPIR. In contrast to pragma-based solutions like OpenACC or OpenMP, Impala's intrinsics integrate seamlessly into Impala's type system. This allows the programmer to hide the use of these functions behind other functions. Consider a higher-order function `iterate` in order to iterate over a field:

```
iterate(fld, |x, y| { /* some loop body */ });
```

Figure 6 depicts one implementation of `iterate` which vectorizes the loop body and one which schedules the loop body on the GPU. Other iterator implementations may use other intrinsics and/or more sophisticated blocking schemes.

DSL Embedding. We demonstrate the effectiveness of our partial evaluator on two examples. First, we present a small framework for stencil computations in image processing: The framework essentially is an "interpreter" that applies a stencil to an image. The aspects of boundary handling, application of the stencil, and the stencil itself are cleanly separated. PE composes those aspects together and produces high-performance code that we specialize for execution on CPU and GPU targets. Second, we present a DSL for the V-cycle multigrid iteration; a multigrid method widely-used in high-performance computing. The V-cycle employs different stencils to smooth the error on different resolutions of the same data. Passing the V-cycle components as functions to the DSL allows us to merge multiple components in order to reduce high latency memory accesses.

5.1 Stencil Computations

A linear filter convolves an image with a filter mask by applying the filter mask to each pixel. Examples of linear filters are the Gaussian blur filter, the Laplace operator, or the Sobel operator. Since the filter mask for linear filters like the Gaussian blur or the Sobel operator

[2] `http://anydsl.github.io`

	GTX 680				Radeon R9 290X		Iris 5100		Core i5-4288U	
	NVVM		OpenCL		OpenCL		OpenCL		CPU	AVX
LU	–	✓	–	✓	–	✓	–	✓	–	–
SS	2.84	2.77	2.92	2.78	1.02	0.97	27.26	25.63	94.00	124.84
+ BH	2.26	**1.90**	2.88	**2.81**	1.04	0.99	26.15	18.17	28.00	23.60
+ SM	2.23	1.98	3.42	2.85	0.82	**0.75**	32.01	**15.55**	42.87	**18.31**
OpenCV	2.25		4.08		0.89		23.31		24.31	

Table 1. Execution times in *ms* for the Gaussian blur of size 5×5 on an image of 4096×4096 pixels.

are separable, we split a filter mask of $N \times M$ in a row and column component of size $N \times 1$ and $1 \times M$, respectively. This reduces the number of required memory accesses.

Stencil Specialization (SS). We describe linear filters using the `apply_stencil` function of our stencil framework. This function applies a filter mask passed to that function to a field. Using the *run* annotation we create a specialization for a given linear filter where the filter values are propagated into the code instead of reading them from memory:

```
let stencil: Stencil = { /* ... */ }; // Gaussian blur
let mut out: Field   = { /* ... */ };
for x, y in iterate(out)
  out(y)(x) = @apply_stencil(x, y, arr, stencil, /*...*/);
```

All variants in Table 1 are specialized in this way for the Gaussian blur.

Loop Unrolling (LU). The iterator functions in Figure 6 abstract the iteration order. On a GPU, for example, it is beneficial to process multiple pixels by the same thread. To achieve this, we call `body`—passed to `iterate` as function—multiple times for different iteration points. This unrolls the iteration space by `unroll_factor` (`4` in the example below). In order to keep the implementation parametric in its unroll factor, we use PE for this job:

```
fn iterate(out: Field, body: fn(int, int) -> ()) -> () {
  let unroll_factor = 4;
  let grid  = (out.cols, out.rows/unroll_factor, 1);
  let block = (128, 1, 1);
  nvvm(grid, block, || {
    let x = tid_x() + ntid_x()*ctaid_x();
    let y = tid_y() + ntid_y()*ctaid_y()*unroll_factor;
    for i in @range(0, unroll_factor)
      body(x, y + i * ntid_y());
  });
}
```

Variants in Table 1 checked in line *LU*, use this technique.

Boundary Handling (BH). To handle array boundaries, we clamp the index to the last valid element within the array in the `apply_stencil` function. If we consider the row component, we need only to apply boundary handling at the left and right border of the image. Therefore, we introduce a `region` parameter to `apply_stencil`:

```
fn apply_stencil(region: int, /*...*/,
        bh_lower: fn(int, int, int, fn(float)) -> int),
        bh_upper: fn(int, int, int, fn(float)) -> int)) -> float {
  // ...
  if region==0 { x = bh_lower(x, 0, arr.cols, return); } // left
  if region==2 { x = bh_upper(x, 0, arr.cols, return); } // right
  // ...
}

fn iterate(/*...*/) -> () {
  let limits = /* lower and upper limits for each region */;
  for y in $range(0, out.rows)
    for region in @range(0, 3)   // left, center, right
      let bounds = limits(region);
      for x in $range(bounds(0), bounds(1))
        @body(x, y, region);
}
```

Now, the `iterate` function iterates over different regions of each line instead of naïvely iterating over the whole image. Due to the *run* annotation on the loop which iterates over the regions, boundary checks will only appear in the residual code for the left and right regions and will be specialized into `apply_stencil`. The programmer passes the functions for boundary handling as higher-order arguments to the stencil DSL, for example:

```
fn clamp_low(idx: int, low: int, up: int, out: fn(float)) -> int {
    if idx < low { low } else { idx }
}
fn const_low(idx: int, low: int, up: int, out: fn(float)) -> int {
    if idx < low { out(1.0f) } else { idx }
}
```

The function `const_low` always skips further computations in the case that `idx` lies outside the given range by passing the constant `1.0f` to the continuation `out`. This is the **`return`** continuation in `apply_stencil`. During PE it is important to infer that the function passed to `out` is the proper post-dominator (see Section 4).

Scratchpad Memory (SM). The stencil operation of the filter has high spatial locality and neighboring elements are read by multiple threads. Therefore, we can first load data for a group of threads to fast scratchpad memory (shared or local memory) and read then the neighboring elements from this scratchpad. This allows also to fuse the row and column component of the filter into a single kernel using scratchpad memory as output memory for the first component and as input memory for the second component. Fusing multiple components is outlined in Section 5.2.

Evaluation. For the measurements we use a separated version of the Gaussian blur filter with a 5×5 filter mask and an image of 4096×4096 pixels. All specialized versions are generated from the same generic description using PE. Table 1 shows the median execution time in ms on the GTX 680 using the CUDA 6.5 drivers and toolkit, on the R9 290X using the Catalyst 15.7 drivers, on the Iris 5100, and on the Intel Core i5-4288U. On the GPU, the median of seven runs is used while 27 runs are used on the CPU.

The last line shows the execution for hand-tuned CUDA, OpenCL, and CPU (vectorized C++) implementations from OpenCV (version 2.4.10), a state-of-the-art image processing toolbox. The CUDA implementation in OpenCV is provided by NVIDIA experts and uses similar optimizations: the filter is separated, the iteration space is unrolled, border handling is limited to thread blocks at the image border, and fast on-chip scratchpad memory is used to stage data. The OpenCL implementation in OpenCV is provided by AMD experts and merges the row and column component in a single kernel: First, the data is loaded to fast on-chip scratchpad memory and the column component is executed, storing its results again to the scratchpad memory. Then, the row component is executed, loading its input from scratchpad and storing the result back to device memory. On the CPU, the row component is manually vectorized 8-fold using double-pumped SSE. The column component uses superword-level parallelism (SLP), unrolling multipe loop iterations so that the compiler can merge them easily into vector operations. The schedule applies always first the row component and then the column component. This allows to hold the intermediate results in cache. It can be seen from Table 1 that the specialized versions we obtain through PE even outperform the hand-tuned implementations in OpenCV.

At the same time, our implementation is more concise: The hand-tuned CUDA version from OpenCV consists of 251 lines of CUDA code plus 386 lines of kernel instantiations for different filter mask sizes and boundary handling modes. The kernel implementation in OpenCL requires 142 lines of OpenCL code; boundary handling is realized via macros that wrap memory accesses. For the CPU implementations, more than 1500 lines of code are required, providing specialized implementations for different data types, kernel sizes, and target instruction sets (SSE, Neon). Our Impala implementation requires 88 lines of code.

We use the *run* annotations highlighted in the sample codes to trigger PE; none of them annotates a "dangerous" recursive call (see Section 3.3). We need *halt* annotations only to prevent loops iterating over the field to be evaluated at compile time.

5.2 The V-Cycle Multigrid Solver

The basic idea of the multigrid method is to smooth the error (e. g., using an iterative method like Jacobi or Gauss-Seidel) on different resolutions of the same data. The *V-cycle* describes one possible multigrid iteration [4, 50]. To transform data between different resolutions of the multigrid the algorithm uses the *restrict* and *interpolate* methods. On each level, the error is smoothed (*smoother*) and estimated (*residual*). This process is recursive and starts at the finest resolution.

For a V-cycle DSL we would like to have the different methods pluggable. Using Impala (see Figure 7), we pass the multigrid components as higher-order functions to `vcycle`. Furthermore, we use the `iterate` function introduced in Section 5.1. The partial evaluator inlines the multigrid components, unrolls the recursion and propagates other inputs (stencils, etc.). Furthermore, the evaluator weaves in the special higher-order functions for hardware-specific code generation. By providing specialized `iterate` implementations for CPU and GPU, we map the same algorithm to different target platforms.

This is a naïve implementation as each multigrid component is run after another. However, hand-tuned implementations of the V-cycle might merge multiple multigrid components in order to save unnecessary memory accesses. In Impala we achieve the same optimization by custom `iterate` functions that compute multiple components at once. As an example, consider the computation of the residual component followed by the restrict component: Instead of computing the residual for the whole field first and then restrict the field produced by the residual, we compute the residual only for two rows and restrict the residual before the next rows are processed. This pipelined processing allows to hold the result of the restrict component in cache on the CPU and allows to merge compute kernels on the GPU when using scratchpad (local or shared) memory. On the GPU, this has the same effect as loop fusion. Figure 8 illustrates this for the CPU. The index passed to the `residual` and `restrict` component refer to the temporary field. The offset to the current row of the other fields are tracked in the `Field` object and are used when accessing field elements. Merging the two components is only valid if the operation of the multigrid components is known: in our example, the restrict component is allowed to access two rows only. Otherwise, a larger temporary array has to be allocated and pre-computed before applying `restrict`.

Results. While we have shown in Section 5.1 that we achieve competitive performance for stencil codes, the multigrid iteration offers optimization opportunities when components are scheduled in a clever way. Figure 9 shows the speedup we get by merging the residual and restrict components for the first level of the V-cycle (smooth, residual, restrict, interpolate). The speedup is between 11 % on the CPU and up to 20 % on the GPU. Considering only the residual and restrict component, the computation is 25 % (27 %) faster on the CPU (AVX) and 42 % (45 %) faster on the GPU when using NVVM (SPIR). For AVX, we vectorize only the smooth and residual component since restrict and interpolate are otherwise slower due to their memory access pattern. On the Iris 5100, the execution takes 16 % longer when the two components are merged. Note that this is expected since the scratchpad memory is mapped to slow global memory in the Iris 5100 architecture. Consequently, the specialization for the Iris 5100 would not make use of scratchpad memory.

Furthermore, we compare the performance of our specialized V-cycle implementation against the performance of generated implementations by HIPA^{cc}, a DSL framework for stencil computations [35]. HIPA^{cc} provides CUDA and OpenCL back ends for execution on GPUs. We use the HIPA^{cc} implementation from [34], which uses the same V-cycle components as our implementation. For the first level of the V-cycle, our normal implementation has the same performance on the Iris 5100 (32.54 ms vs. 32.81 ms), is 8 % faster on the Radeon R9 290X (2.26 ms vs. 2.43 ms), and is 9 % slower on the GTX 680 (4.78 ms vs. 4.35 ms). Our merged implementation (see Figure 9) outperforms the HIPA^{cc} implementations on the Radeon R9 290X by 34 % and on the GTX 680 by 12 %.

```
fn vcycle(field: Field, lvls: int, vsteps: int, ssteps: int,
          smoother:    fn(/* ... */) -> (),
          residual:    fn(/* ... */) -> (),
          restrict:    fn(/* ... */) -> (),
          interpolate: fn(/* ... */) -> ()) -> Field {
  // allocate memory for all lvls: Sol, RHS, Res, Tmp
  fn vcycle_rec(lvl: int) -> () {
    if lvl == lvls-1 {
      for i in range(0, ssteps)   // solve by ssteps smooths
        if i>0 { swap(Sol(lvl), Tmp(lvl)); }
        for x, y in iterate(Sol(lvl))
          solver(x, y, /*fields*/);
    } else {
      for i in range(0, ssteps)                        pre-smoothing
        if i>0 { swap(Sol(lvl), Tmp(lvl)); }
        for x, y in iterate(Sol(lvl))
          solver(x, y, /*fields*/);
      for x, y in iterate(Res(lvl))                  compute residual
        residual(x, y, /*fields*/);
      for x, y in iterate(RHS(lvl+1))              restrict residual
        restrict(x, y, /*fields*/);
      vcycle_rec(lvl+1);                                    recurse
      for x, y in iterate(Sol(lvl))             interpolate error and
        interpolate(x, y, /*fields*/);          coarse grid correction
      for i in range(0, ssteps)                       post-smoothing
        if i>0 { swap(Sol(lvl), Tmp(lvl)); }
        for x, y in iterate(Sol(lvl))
          solver(x, y, /*fields*/);
    }
  }
  for i in range(0, vsteps) { vcycle_rec(0); }
}

let res = @vcycle(field, lvls, vsteps, ssteps,
                  jacobi, residual, restrict, interpolate);
```

Figure 7. V-cycle implementation in Impala.

```
fn iterate_rr(Sol: Field, Res: Field, RHSF: Field, RHSC: Field,
              residual: fn(/* ... */) -> (),
              restrict: fn(/* ... */) -> ()) -> () {
  let mut tmp: Field = { /* ... */ }; // temp array for 2 rows

  for y in $range_step(0, Res.rows, 2)
    for yi in @range(0, 2)
      for x in $range(0, Res.cols)  // residual for two rows
        @residual(x, yi /* ... */ Sol, tmp, RHSF);
    for x in $range(0, RHSC.cols)   // restrict the residual
      @restrict(x, 0 /* ... */ tmp, RHSC);
}
```

Figure 8. Merging residual and restrict on the CPU.

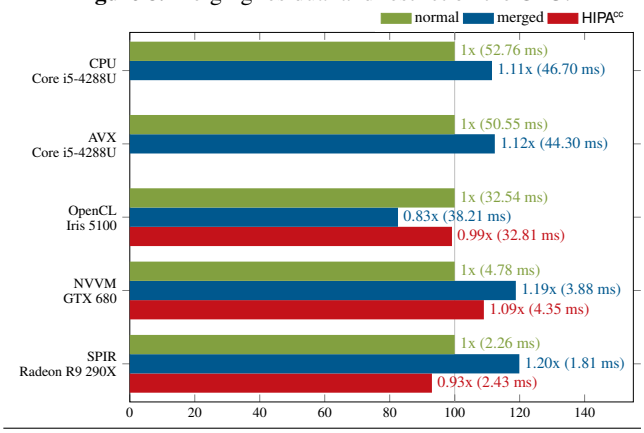


Figure 9. Speedup from merging the residual and restrict computation for the first level (4096×4096) of the V-cycle (smooth, residual, restrict, interpolate). The speedup over HIPA^{cc} implementations is also given where available.

Discussion. Our implementation can be easily extended to express different multigrid iterations. It is actually sufficient to change the recursion in the V-cycle implementation in order to get the schedule for the W-cycle multigrid iteration.

The evaluation has shown that we can map the same high-level description to different target platforms by providing target-specific mappings. Moreover, we merge multiple components as shown

exemplarily for the residual and restrict components. This yields specialized implementations that outperform the implementations generated by HIPA^{cc}, which has no support for kernel fusion.

6. Conclusions

In this paper, we present a novel partial evaluation strategy and its application to shallow embedding of DSLs. To this end, we formally define full as well as partial evaluation and prove its correctness and termination property on a CPS-based variant of PCF. Every invocation of the partial evaluator will terminate if full evaluation of that program terminates. The termination behavior of the program is retained in a predictable way: The partial evaluator skips dynamic branches and continues at the post-dominator. To efficiently compute post-dominators on parts of higher-order programs with free variables we presented a local, partially context-sensitive CFA. In order to steer the evaluator from within the language, we introduce run- and halt-annotations. As long as run annotations are not placed on recursive calls, PE will not induce divergence. Annotating a call to a recursive function is not problematic.

We apply our partial evaluator on higher-order functions to embed high-performance DSLs and generate optimized code for different hardware architectures. We evaluate our technique experimentally in the field of visual and high-performance computing and show that our evaluator produces highly specialized and efficient code for CPUs as well as GPUs that matches the performance of hand-tuned expert code.

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. It is also co-funded by the European Union (EU), as part of the Dreamspace project. Furthermore, the authors would like to thank Marcel Köster for helping with many parts of the compiler.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] L. Birkedal and M. Welinder. Hand-writing program generator generators. In *PLILP*, 1994.

[3] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *JFP*, 1993.

[4] W. L. Briggs, H. Van Emden, and S. F. McCormick. *A Multigrid Tutorial*, volume 2. SIAM, June 2000.

[5] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

[6] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. In *APLAS*, 2007.

[7] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA*, 2010.

[8] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *PEPM*, 1993.

[9] W. R. Cook and R. Lämmel. Tutorial on online partial evaluation. In *DSL*, 2011.

[10] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, 2004.

[11] P. Danilewski, M. Köster, R. Leißa, R. Membarth, and P. Slusallek. Specialization through dynamic staging. In *GPCE*, 2014.

[12] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *SC*, 2011.

[13] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *PLDI*, 2013.

[14] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 1999. Reproduction of the 1971 paper.

[15] J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings. In *ICFP*, 2014.

[16] R. Glück. A self-applicable online partial evaluator for recursive flowchart languages. *Softw., Pract. Exper.*, 42, 2012.

[17] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *GPCE*, 2008.

[18] B. T. Howard and J. C. Mitchell. Operational and axiomatic semantics of pcf. In *LFP*, 1990.

[19] P. Hudak. Modular domain specific languages and tools. In *Software Reuse*, 1998.

[20] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, 1985.

[21] N. D. Jones. Mix ten years later. In *PEPM*, 1995.

[22] N. D. Jones and A. J. Glenstrup. Program generation, termination, and binding-time analysis. In *GPCE*, 2002.

[23] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation (extended abstract). In *MFPS*, 1987.

[24] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.

[25] J. Jørgensen. Similix: a self-applicable partial evaluator for Scheme. In *Partial Evaluation*, 1998.

[26] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *POPL*, 1986.

[27] V. Jovanović, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-Yang: Concealing the deep embedding of DSLs. In *GPCE*, 2014.

[28] R. Karrenberg and S. Hack. Whole-function vectorization. In *CGO*, 2011.

[29] M. Katz and D. Weise. Towards a new perspective on partial evaluation. In *PEPM*, 1992.

[30] *SYCL™ Specification*. Khronos, 2015. Version 1.2.

[31] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[32] R. Leißa, M. Köster, and S. Hack. A graph-based higher-order intermediate representation. In *CGO*, 2015.

[33] M. McCool. A retargetable, dynamic compiler and embedded language. In *CGO*, 2011.

[34] R. Membarth, O. Reiche, C. Schmitt, F. Hannig, J. Teich, M. Stürmer, and H. Köstler. Towards a performance-portable description of geometric multigrid algorithms using a domain-specific language. *JPDC*, 74(12):3191–3201, 2014.

[35] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPA^{cc}: A domain-specific language and compiler for image processing. *TPDS*, 2015.

[36] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in Scala: towards the systematic construction of generators for performance libraries. In *GPCE*, 2013.

[37] J. Palsberg and M. I. Schwartzbach. Binding-time analysis: Abstract interpretation versus type inference. In *ICCL*, 1994.

[38] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 1977.

[39] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[40] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 2002.

[41] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2010.

[42] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In *PLDI*, 2014.

[43] E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, 1993.

[44] A. Shali and W. R. Cook. Hybrid partial evaluation. In *OOPSLA*, 2011.

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

[46] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. OptiML: an implicitly parallel domain-specific language for machine learning. In *ICML*, 2011.

[47] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, 1999.

[48] W. Tait. Intensional interpretations of functionals of finite type i. *J. Symb. Log.*, 1967.

[49] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, 2011.

[50] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, Dec. 2000.

[51] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *SPLASH*, 2013.