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. 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. However, shallow embeddings do 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.

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 that is partially 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. HIPAA [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.
Impala’s syntax borrows from Rust. 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 dive 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:

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

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 dive 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:

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

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:

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

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

1 Impala’s syntax borrows from Rust. `|a| x` means `let x = a.`

---

**Figure 2.** Partial evaluation and metaprogramming.

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).

**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 `Pspec` (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 self 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 (mixoffline) runs on that annotated program as opposed to directly running the specializer (mixonline) 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 `Pann`, `cogen` produces its generating extension `Pspec` (Figure 2c): All static parts of `Pann` are copied over to `Pspec`. Dynamic parts are converted into a program that generates the specialized program `Pspec`. Thus, `Pspec` is parametric in `Pann`'s
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 \( P_{gen} \)’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 deduce dynamic in the following example:

\[
\text{fn \( f(a, b, d) \) \{ if \( a == d \) \{ /*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 List [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 \( \text{Rep}[T] \) 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 \( \text{Rep} \) types. Array Building Blocks [33] and Halide [39] leverage a similar staging mechanism to construct the actual program representation with C++ as host language. HIPAC [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 \( \text{Int} \). Thus, the expression is executed when the host program runs. But since \( r \) is of type \( \text{Rep}[\text{Int}] \), executing \( r * r \) results in a residual program containing a multiplication. The implementation of \( \text{cogen} \) lies in LMS’ library which implements a \( * \) for \( \text{Rep}[\text{Int}] \). The downside of this approach is that for data types unknown to LMS, the programmer must implement these overloads ("\( \text{cogen} \) 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:

\[
\text{def } f[\text{Int}](i: \text{Int})= \{ /*...*/ \}
\]

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]:

\[
\text{fn count(i: \text{Int}, n: \text{Int}) \{ /*...*/ \}}
\]

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 \texttt{Bool}, the loop will be run when the host program runs; if it is of type \texttt{Rep(Bool)}, LMS will construct a residual loop. This approach only works since Scala does not provide \texttt{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 \texttt{Rep[Int]} bound diverges.

Both Simulix [25] and Schism [8] are offline evaluators using BTA. These evaluators do 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 $\lambda^{pp}$ (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 $\lambda^{pp}$ is similar to the simply-typed lambda calculus with an additional fixpoint operator called \texttt{letrec} to allow recursion. Furthermore, $\lambda^{pp}$ uses CPS for function abstractions and applications. Thus, $\lambda^{pp}$ 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 $a \leq b$ 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 $\bar{a}$ 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 \texttt{where} to bind non-recursive functions and \texttt{whererec} to bind recursive ones. Finally, we sometimes use additional features in $\lambda^{pp}$ 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 $\text{Fn}(e)$ 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 $\llbracket e \rrbracket = e$. We write $\neg f e$ if we require $e$ to be in normal form.

Definition 2 (Exits). Let $\text{exit}(e) := [\text{exit}] e \mid e \leq b$ be the set of exits in the body $b$.

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

Definition 4 (Constant). We call an expression $a$ a constant iff $a$ 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 $f := \lambda x : t. b$ be a function constant. We call an argument list $\Gamma$ of constants a valid configuration for $f$ iff the application $f(\Gamma)$ 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 $\llbracket e \rrbracket$ evaluates an expression. If two literals occur in an additior they will be folded (operator @ 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 $\llbracket e \rrbracket$ is strongly normalizing, that is, every $\llbracket e \rrbracket$ 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 $\lambda^{pp}$ 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 $b \Rightarrow^e b'$ 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 $\not \exists b' : b \Rightarrow^e b'$. We write $\neg b f$ if we require $b$ to be in normal form.

A function application first reduces its callee $e_1$ and arguments to normal form (E-App). The requirement $(e_1, \bar{e}) \neq (e_1', \bar{e}')$ 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 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$_{\text{skip}}$).

An ifz first evaluates its arguments (E-Ifz) and then selects either its true or false continuation if possible (E-Ifz$_1$ and E-Ifz$_2$). Similar to E-App, rule E-Ifz only triggers if at least one of its expressions are
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-Skip).

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

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

Definition 7 (Lambda Lifting). Let \( x \mapsto t \) be the list of free variables in a well-typed body \( b \) under \( \lambda x.b \). Then \( \lambda x.b \) is the lambda-lifted function [20] of \( b \). As shorthand, we write \( f_k \).

Definition 8 (Post-Dominator). Let \( b \) be a well-typed body under \( \lambda x.b \). We call \( p \) a post-dominator of \( b \) if \( p = \bot \) or all possible finite evaluations from \( b \) to any exit points \( p \) is \( \bot 

\[
\forall x \in \text{exit}(b) : \forall e \in C(f_k) : f_k(\overline{e}) \Rightarrow^* x \rightarrow f_k(\overline{e}) \Rightarrow^* p .
\]

Remark. Hence, \( \bot \) designates the point in execution after all \( \text{exit}(b) \) 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 \text{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 \( \text{postdom}(b) \) to exist and to compute a valid but not necessarily the immediate post-dominator of \( b \). Using \( p \), \text{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).

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

**Progress:** If \( \Gamma \vdash b \) then either \( b = \text{exit } l \) or \( \text{not } b \).

**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-Skip nor E-Skip. This means, we will always know a body to continue evaluation.

**Lemma 3 (Full Evaluation).** Let \( f \) be a function constant. Evaluating \( f(\overline{e}) \) where \( \overline{e} \in C(f) \) never triggers E-Skip.

Proof. For the proof of Lemma 2 we have shown that all expressions collapse to constants during evaluation. Thus, the preconditions for E-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] showed the simply-typed lambda calculus is strongly normalizing. If we remove letrecs from \( \lambda^{ps} \), the resulting language is analogously strongly normalizing.
Theorem 1 (Correctness). Let \( b \) be a well-typed body under \( X(b) \). We define \( f' \) to be \( \lambda X(b) \) \( b \) with \( b \Rightarrow b' \) for some \( n \in \mathbb{N} \). If \( f_X(b) \) terminates with result \( l \) so does \( f'(\overline{c}) \):

\[
\forall c \in C(f)_n: f_X(b) \Rightarrow^* \text{exit } l \Rightarrow f'(\overline{c}) \Rightarrow^* \text{exit } l.
\]

Proof. Let \( \overline{c} \) be the number of times rules E-App, or E-HzApp are triggered during evaluation. We prove the theorem by induction on \( \overline{c} \).

By Lemma 4 evaluating \( f_X(b) \Rightarrow^* \text{exit } l \) never triggers E-App, nor E-HzApp. As an inductive base case, we presume that \( b \Rightarrow b' \) does not trigger E-App, nor E-HzApp for any \( n \). This implies that both evaluations \( f_X(b) \Rightarrow^* \text{exit } l \) and \( b \Rightarrow b' \) will trace through the same functions till \( b' \). Clearly, expressions which do not contribute to control flow might not be constants. Hence, there exists an \( n \) such that \( b' \equiv \text{exit } e \).

For the inductive step we have to show that triggering E-App, or E-HzApp, \( n+1 \) times still leads to a normal form. Both E-App, and E-HzApp trigger E-Skip. By Definition 4 postdom(b) either retrieves \( \lambda \) or a body \( b \) which preserves \( b' \)’s termination behavior. By the induction hypothesis evaluating \( b \) along E-Skipping will finally trigger E-Skipping if the original program terminates. □

3.3 Run and Halts 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:

\[
\begin{align*}
\text{letrec } &\text{count} = (i, N, \text{ret}). \\
&\text{if}(i = N) \quad \text{letrec } \text{count} = (i+1, N, \text{ret}) \\
&\text{else} \quad \text{if}(i < N) \quad \text{letrec } \text{count} = (i+1, N, \text{ret}) \\
&\quad \text{ret}(i) \\
&\quad \text{in } /\ldots/ 
\end{align*}
\]

Its \( X^{X^n} \) translation is presented on the right-hand side. During the search for annotations, all \textit{letrec}s are substituted in—as in rule E-Letrec. In practice, we perform the substitution on demand when a free variable bound by a \textit{letrec} is encountered. The \textit{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. \( \lambda \)) outside the body is called, all remaining free variables, together with any \textit{exit}, must be considered local \textit{exits} for the purpose of evaluation and the definition of post-dominators in particular. Similarly, a \textit{halt} annotation \( \#g(args, 3) \) causes the evaluator to stop specialization at that point and resume at \( 1 \).

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

\[
\text{if } l \text{ _always } \text{false } \quad \{ \# \text{_will } \text{not } \text{terminate } (42) \}
\]

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 \( X \)-App:

\[
\begin{align*}
\text{fn count}(i: \text{int}, N: \text{int}) &\rightarrow \text{int} \quad \text{letrec } \text{count} = (i, N, \text{ret}). \\
&\text{if}(i < N) \quad \text{letrec } \text{count} = (i+1, N, \text{ret}) \\
&\text{else} \quad \text{if}(i = N) \quad \text{letrec } \text{count} = (i+1, N, \text{ret}) \\
&\quad \text{ret}(i) \\
&\quad \text{in } /\ldots/ 
\end{align*}
\]

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

\[
\begin{align*}
\text{letrec } &\text{count} = (i, N, \text{ret}). \\
&\text{if}(i = N) \quad \text{letrec } \text{count} = (i+1, N, \text{ret}) \\
&\text{else} \quad \text{if}(i < N) \\
&\quad \text{letrec } \text{count} = (i+1, N, \text{ret}) \\
&\quad \text{ret}(i) \\
&\quad \text{in } /\ldots/ 
\end{align*}
\]

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

This situation only emerged because a \textit{recursive} \textit{call} has been annotated. In particular, an annotated call to a function bound via \textit{letrec} within its definition like \#count in the example above. A call to a recursive function outside its definition is \textit{not} recursive and, thus, does \textit{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 \textit{ackermann}(3, 4) to 125. As outlined above, this call is \textit{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 \textit{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 \textit{k}-\textit{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 \textit{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 \textit{exit}, 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-HzApp) or call (E-App), 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 \textit{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 \textit{inside functions}. We call functions declared outside the scope, in particular functions defined in other translation units or intrinsic higher-order functions such as \textit{map} (Section 5), \textit{outside functions}. Suppose PE of program \textit{start} to \textit{end} in Figure 5a meets a dynamic branch in \textit{f}. Function out and higher-order parameter end are not defined within \textit{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...
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 pb.

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 CFA to distinguish control flow from multiple calls to an outside function to their function arguments. The CFA in Figure 5d thus separates the calls to out in T, A and C and their different arguments. The edge from outC 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 outC to N and outA to the node outT. 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.

5. Applications and Evaluation

The Impala\(^2\) compiler translates the source program into Thorin [32]: a functional intermediate representation (IR) similar to \( \lambda \). 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:

```javascript
fn vectorize(L: int, a: int, b: int, body: fn(int) -> () -> () { let vector_length = L; for i in range(0, vector_length) { body(i); }});  
```

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

```javascript
fn nvvm(grid: (int, int), block: (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:

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

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...
All variants in Table 1 are specialized in this way for the Gaussian blur. Execution times in apply_stencil and right regions and will be specialized into boundary checks will only appear in the residual code for the left run.

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 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 it reduces the number of required memory accesses.

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 in the example below. In order to keep the implementation parametric in its unroll factor, we use PE for this job:

```rust
fn iterate(out: Field, body: fn(int, int) -> ()) -> () {
    let unroll_factor = 4;
    let block = (1, 1);
    fn map(grid, block) {
        for x in range(0, block.x) {
            for y in range(0, block.y) {
                body(x, y);
            }
        }
    }
    map(grid, block);
}
```

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

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:

```rust
fn apply_stencil(region: int, out: Field, body: fn(int, int) -> ()) {
    let limits = 
        if region == 0 { x = bh_lower(x, 0, arr.cols, return); } // left
        if region == 1 { x = bh_upper(x, 0, arr.cols, return); } // right
        if region == 2 { x = bh_upper(x, arr.cols, return); } // right
        ...}
    limits = 
        if region == 0 { x = bh_lower(x, 0, arr.cols, return); y = bh_upper(y, 0, arr.cols, return); } // left
        if region == 1 { x = bh_upper(x, 0, arr.cols, return); y = bh_upper(y, 0, arr.cols, return); } // right
    ...}
}
```

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

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 x 5 filter mask and an image of 4096 x 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 time 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 multiple 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 cyclic. 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 naive 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 HIPAC®. A DSL framework for stencil computations [35], HIPAC® provides CUDA and OpenCL back ends for execution on GPUs. We use the HIPAC® 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 imple-

```haskell
fn vcycle(field: Field, lvls: int, vsteps: int, ssteps: int, 
    smoother: fn(fld: Field) -> Field, 
    residual: fn(fld: Field) -> Field, 
    restrict: fn(fld: Field) -> Field, 
    interpolate: fn(fld: Field) -> 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 i in range(0, vsteps) // post-smoothing
            if i>0 { swap(Sol(lvl), Tmp(lvl)); }
            for x, y in iterate(Sol(lvl))
                solver(x, y, /*fields*/);
        }
    }

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

fn iterate_rec(Sol: Field, Res: Field, RHSF: Field, RHSC: Field, 
    residual: fn(fld: Field) -> Field, 
    restrict: fn(fld: Field) -> Field, 
    pre-smoothing: fn(fld: Field) -> Field, 
    post-smoothing: fn(fld: Field) -> Field, 
    // temp array for 2 rows
    pre-smoothing = fn(fld: Field) -> () {
        let mut tmp = Field = (/* ... */); // temp array for 2 rows
        for y in range(0, 2) {
            for x in range(0, 2) {
                tmp = residual(x, y, /*fields*/);
                for x in range(0, 2) {
                    tmp = residual(x, y, /*fields*/);
                    for x in range(0, 2) {
                        tmp = residual(x, y, /*fields*/);
                    }
                    for y in range(0, 2) {
                        tmp = residual(x, y, /*fields*/);
                        for x in range(0, 2) {
                            tmp = residual(x, y, /*fields*/);
                            for y in range(0, 2) {
                                tmp = residual(x, y, /*fields*/);
                                for x in range(0, 2) {
                                    tmp = residual(x, y, /*fields*/);
                                    for y in range(0, 2) {
                                        tmp = residual(x, y, /*fields*/);
                                        for x in range(0, 2) {
                                            tmp = residual(x, y, /*fields*/);
                                            for y in range(0, 2) {
                                                tmp = residual(x, y, /*fields*/);
                                                for x in range(0, 2) {
                                                    tmp = residual(x, y, /*fields*/);
                                                    for y in range(0, 2) {
                                                        tmp = residual(x, y, /*fields*/);
                                                        for x in range(0, 2) {
                                                            tmp = residual(x, y, /*fields*/);
                                                            for y in range(0, 2) {
                                                                tmp = residual(x, y, /*fields*/);
                                                                for x in range(0, 2) {
                                                                    tmp = residual(x, y, /*fields*/);
                                                                    for y in range(0, 2) {
                                                                        tmp = residual(x, y, /*fields*/);
                                                                        for x in range(0, 2) {
                                                                            tmp = residual(x, y, /*fields*/);
                                                                            for y in range(0, 2) {
                                                                                tmp = residual(x, y, /*fields*/);
                                                                                for x in range(0, 2) {
                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                    for y in range(0, 2) {
                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                        for x in range(0, 2) {
                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                            for y in range(0, 2) {
                                                                                                tmp = residual(x, y, /*fields*/);
                                                                                                for x in range(0, 2) {
                                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                                    for y in range(0, 2) {
                                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                                        for x in range(0, 2) {
                                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                                            for y in range(0, 2) {
                                                                                                                tmp = residual(x, y, /*fields*/);
                                                                                                                for x in range(0, 2) {
                                                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                                                    for y in range(0, 2) {
                                                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                                                        for x in range(0, 2) {
                                                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                                                            for y in range(0, 2) {
                                                                                                                                tmp = residual(x, y, /*fields*/);
                                                                                                                                for x in range(0, 2) {
                                                                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                                                                    for y in range(0, 2) {
                                                                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                                                                        for x in range(0, 2) {
                                                                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                                                                            for y in range(0, 2) {
                                                                                                                                                tmp = residual(x, y, /*fields*/);
                                                                                                                                                for x in range(0, 2) {
                                                                                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                                                                                    for y in range(0, 2) {
                                                                                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                                                                                        for x in range(0, 2) {
                                                                                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                                                                                            for y in range(0, 2) {
                                                                                                                                                                tmp = residual(x, y, /*fields*/);
                                                                                                                                                                for x in range(0, 2) {
                                                                                                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                                                                                                    for y in range(0, 2) {
                                                                                                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                                                                                                        for x in range(0, 2) {
                                                                                                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                                                                                                            for y in range(0, 2) {
                                                                                                                                                                                tmp = residual(x, y, /*fields*/);
                                                                                                                                                                                for x in range(0, 2) {
                                                                                                                                                                                    tmp = residual(x, y, /*fields*/);
                                                                                                                                                                                    for y in range(0, 2) {
                                                                                                                                                                                        tmp = residual(x, y, /*fields*/);
                                                                                                                                                                                        for x in range(0, 2) {
                                                                                                                                                                                            tmp = residual(x, y, /*fields*/);
                                                                                                                            } }
                                                        } }
                                                    } }
                                                } }
                                            } }
                                        } }
                                    } }
                                } }
                            } }
                        } }
                    } }
                } }
            } }
        } }
    }

Figure 7. V-cycle implementation in Impala.

Figure 8. Merging residual and restrict on the CPU.

Figure 9. Speedup from merging the residual and restrict computation for the first level (4096 x 4096) of the V-cycle (smooth, residual, restrict, interpolate). The speedup over HIPAC® implementations is also given where available.

Discussion. Our implementation can be easily extended to express different levels of the 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...
variables we presented a local, partially context-sensitive CFA. In this work, we have defined run- and halt-annotations. As long as run annotations are not placed on recursive calls, PE will not induce divergence. Annotating a call 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 steer the evaluator from within the language, we introduce run- and halt-annotations.

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