Graph IRs for Impure Higher-Order Languages

Making Aggressive Optimizations Affordable with Precise Effect Dependencies

OLIVER BRAČEVAC[*], Purdue University, USA and Galois, Inc., USA
GUANNAN WEI, Purdue University, USA
SONGLIN JIA, Purdue University, USA
SUPUN ABEYSINGHE, Purdue University, USA
YUXUAN JIANG, Purdue University, USA
YUYAN BAO, Augusta University, USA
TIARK ROMPF, Purdue University, USA

Graph-based intermediate representations (IRs) are widely used for powerful compiler optimizations, either interprocedurally in pure functional languages, or intraprocedurally in imperative languages. Yet so far, no suitable graph IR exists for aggressive global optimizations in languages with both effects and higher-order functions: aliasing and indirect control transfers make it difficult to maintain sufficiently granular dependency information for optimizations to be effective. To close this long-standing gap, we propose a novel typed graph IR combining a notion of reachability types to track aliasing with an expressive effect system to compute precise and granular effect dependencies, while supporting local reasoning and separate compilation. Our high-level graph IR imposes lexical structure to represent structured control flow and nesting, enabling aggressive and yet inexpensive code motion, instruction selection, and other optimizations for impure higher-order programs. We formalize the new graph IR based on a λ-calculus with a reachability type-and-effect system along with a specification of various optimizations. We present performance case studies for CUDA tensor kernel fusion, symbolic execution of LLVM IR, and SQL query compilation in the Scala LMS compiler framework using the new graph IR. We observe significant speedups of up to $21x$.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Semantics**; **Functional languages**;

Additional Key Words and Phrases: intermediate representations, higher-order languages, effects, compilers

ACM Reference Format:
Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 236 (October 2023), 31 pages. https://doi.org/10.1145/3622813

1 INTRODUCTION

Graph-based intermediate representations (IRs) are an attractive alternative to simpler tree-based IRs (such as abstract syntax trees, ASTs) for program optimizations. Graphs are sparse in nature,

[*] Work completed while at Purdue University

Authors' addresses: Oliver Bračevac, Purdue University, West Lafayette, IN, USA and Galois, Inc., Portland, OR, USA, oliver@galois.com; Guannan Wei, Purdue University, West Lafayette, IN, USA, guannanwei@purdue.edu; Songlin Jia, Purdue University, West Lafayette, IN, USA, jia137@purdue.edu; Supun Abeysinghe, Purdue University, USA, tabeysin@purdue.edu; Yuxuan Jiang, Purdue University, USA, jiang700@purdue.edu; Yuyan Bao, Augusta University, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, USA, tiark@purdue.edu.

2475-1421/2023/10-ART236
https://doi.org/10.1145/3622813

eliminate redundancy, and exhibit greater locality of information compared to syntax trees. This avoids ad-hoc reconstruction of non-local information, (*e.g.*, def-use chains), and facilitates implementations of powerful code motion algorithms and other optimizations. In essence, many forms of non-local reasoning in the space of program terms map to local reasoning in the space of dependencies between such terms. For instance, common subexpression elimination (CSE) reduces to hash consing, and dead code elimination (DCE) to graph reachability. Most algebraic optimizations are also just a matter of local "peephole-style" rewrites on nodes and their immediate neighbors.

Impure vs. Higher-Order: Pick One? Graph IRs are widely used in implementations of pure functional languages [Elliott et al. 2003; Henriksen et al. 2017; Peyton-Jones 1987; Peyton-Jones and Salkild 1989; Steuwer et al. 2017]. Pure computation induces only data dependencies, thus granting high degrees of freedom for moving and rewriting code, and enabling aggressive global and interprocedural optimizations. Graph IRs are also widely used for intraprocedural optimizations[1] in imperative language implementations, *e.g.*, "Sea-of-Nodes" IRs [Click and Paleczny 1995] in VMs for Java [Paleczny et al. 2001] and JavaScript [Titzer 2015]. The presence of effects (*e.g.*, heap accesses) in those languages limits the scope of optimizations to single functions/methods. To achieve any kind of non-local optimization, imperative languages rely on aggressive inlining and code duplication [Stadler et al. 2014]. Thus, to reap the benefits of graph IRs, language designers/implementers currently have to choose between two extremes: (1) higher-order purely functional languages with (truly) global optimizations, or (2) imperative languages with limited local optimizations. But what about the combination? So far, no suitable graph IRs for global, interprocedural optimization exist in compilers for languages that combine effects and higher-order functions.

Indirection: The Enemy of Local Reasoning. The main reasons for this significant gap are as follows: (1) higher-order functions introduce imprecise control transfers, (2) aliasing of mutable references introduces imprecise data dependencies, and (3) combinations, such as heap-allocated function values, exacerbate these issues. Consequently, it is hard to extract the sufficiently precise local dependency information that makes graphs useful. Precise dependency extraction is inherently context sensitive, so one might consider higher-order flow analyses [Midtgaard 2012; Might 2007; Shivers 1988] as a natural choice to enhance precision. However, analyses like 0-CFA, while tractable, are imprecise and yield only coarse-grained dependencies. Analyses like k-CFA, which simulate execution up to certain context depth $k > 0$, provide improved precision, but can be prohibitively expensive in practice [Horn and Mairson 2008; Shivers 2004]. Even considering tractable versions [Might et al. 2010], these analyses are whole-program (*i.e.*, non-modular), propagating information through iterative fixpoint computations. Thus, it seems that obtaining precise local reasoning first requires costly non-local reasoning.

The Key: Typed Graphs Tracking Ownership, Separation, and Effects. Fortunately, type systems, specifically polymorphic and dependent types, provide an alternative approach to tracking non-local information, replacing propagation via fixpoint iteration with substitution. Hence, we propose using types to track dependency information, reducing analysis cost and improving modularity. Type inference algorithms are widely available (*e.g.*, [Dolan and Mycroft 2017; Dunfield and Krishnaswami 2022; Odersky et al. 1999, 2001; Parreaux 2020; Pierce and Turner 2000]), propagate non-local information effectively, and improve with increased type annotations [Odersky and Läufer 1996]. Although type inference may have exponential complexity *in the worst case*, it is efficient *in practice*. Unlike k-CFA, type inference does not require simulating execution up to a certain context depth k but relies on substitution to propagate non-local information and thus

[1] Perhaps confusingly, function-level optimizations are sometimes called "global" in the compiler literature in contrast to "local" block-level optimizations.

avoids many of the challenges associated with maintaining fine-grained dependency information for aggressive global optimizations.

Based on these insights, we propose a *typed graph IR* that strikes a balance between affordable analysis cost and precision. We build on *reachability types* [Bao et al. 2021], a dependent type system tracking sets of reachable term variables. It combines ideas from ownership types [Clarke et al. 2013; Noble et al. 1998] and separation logic [Reynolds 2002]. An additional *effect system* (inspired by Gordon [2021]) infers how variables are used (*e.g.*, reads, writes, etc.). Combining reachability and effects allows for precise, local dependency calculations between graph nodes, including true (*i.e.*, read-after-write) and anti-dependencies (*i.e.*, write-after-write, write-after-read).

Our graph IR employs (local) type inference for propagating reachability and effects, and serves as (1) a compilation target of direct-style surface languages, or (2) as part of domain-specific languages (DSLs). In the first case, user annotations guide inference, following the usual credo that some annotations are valuable as documentation, and redundant annotations can be avoided. The amount of required user annotations is reasonable and in the ballpark of Rust [Jung et al. 2021; Matsakis and Klock 2014]. In the second case, DSL authors leverage the constructive properties and the constrained nature of DSLs to determine annotations on primitives, which largely eliminates the need for reachability or effect annotations in user code.

Lexical Structure for Seamless Code Motion and Progressive Lowering. Notably, the new graph IR features variable binders within nodes, creating a high-level representation of structured statements with internal control-flow and nesting (*e.g.*, nested functions, conditionals, loops), as opposed to low-level conditional jumps, and explicit control edges found in traditional "Sea of Nodes" IRs. This approach offers two main advantages: (1) seamless code motion, allowing for flexible movement of conditionals, loops, and lambdas, and (2) progressive lowering, enabling compilers to translate high-level code into successive lower-level representations within the same IR. For example, early-stage loops can have implicitly parallel semantics without committing to a specific behavior, enabling later translations to potentially parallelized and vectorized imperative loop nests. This nesting structure aligns with contemporary IR frameworks such as MLIR [Lattner et al. 2021], as opposed to earlier platforms like LLVM [Lattner and Adve 2004].

Contributions. In summary, we show that reachability types, effects, and lexical structure are a key step toward unlocking the potential of graph IRs for impure functional languages. We present a formal model of the new graph IR, λ^*_{G}, which enjoys key soundness properties, including safety of dependencies. We have also implemented the new graph IR in the Scala LMS compiler framework [Rompf and Odersky 2010; Rompf et al. 2013], where we demonstrate its effectiveness in optimizing impure higher-order programs, yielding significant speedups. We make the following specific contributions:

- Section 2 motivates graph-based global optimizations for impure higher-order functional languages and shows how they can be realized using our graph IR.
- Section 3 formalizes the graph IR in the λ^*_{G}-calculus, a version of Bao et al.'s λ^*-calculus in monadic normal form (MNF) with reachability types and effects, and extends the calculus with a novel type-and-effect-directed synthesis of precise effect dependencies. We prove type-and-dependency safety of λ^*_{G}, *i.e.*, that synthesized dependencies respect the order of runtime effects.
- Section 4 discusses extensions, such as additional forms of structured nodes (*e.g.*, conditionals), soft dependencies which lead to improved dead-code elimination (DCE), and destructive effects which enforce policies such as affine uses.
- Section 5 covers middle-tier optimizations at the graph level, such as DCE, CSE, and λ-hoisting, all backed by the underlying equational theory of λ^*_{G}. We also showcase how the graph IR supports

restricted cases of classic higher-order optimizations (*e.g.*, lambda lifting, super-β inlining), which can optionally be complemented with fixpoint-based flow analyses.
- Section 6 discusses efficient scheduling and code generation algorithms to transform IR graphs back into trees and perform backend optimizations. We showcase various optimizations including code motion, which uses a cheap frequency-estimation heuristic instead of costly flow analyses.
- Section 7 evaluates our graph IR design through detailed case studies and experiments: (1) loop fusion and kernel fusion [Wang et al. 2019a,b] on CPUs/GPUs, (2) symbolic execution of LLVM IR [Wei et al. 2020, 2023b], and (3) SQL query compilation [Essertel et al. 2018; Rompf and Amin 2015; Tahboub et al. 2018]. We achieve significant speedups of up to $21x$.

Section 8 discusses related work and Section 9 offers concluding remarks.

2 OVERVIEW: FROM TYPES AND EFFECTS TO GRAPHS AND DEPENDENCIES

In this section, we discuss the challenges of optimizing higher-order languages with effects (such as Scala and OCaml) and motivate the design of our graph IR. Throughout the paper, we use Scala-like syntax, but the presented ideas easily map to other impure functional languages.

2.1 Motivating Examples: Optimizing with Graph IRs

Consider a program that creates an array of complex numbers on the unit circle and transforms them to their conjugates, *i.e.*, maps each complex number $a + bi$ to $a - bi$:

```
case class Complex(re: Double, im: Double)
val compNums = (0 until 100).map { th => Complex(cos(th), sin(th)) }  /* : Array[Complex] */
val conjNums = compNums.map { c => Complex(c.re, -c.im) }             /* : Array[Complex] */
```

Fusion and Other Simplifications Using Rewriting. Many compilers for functional high-performance DSLs [Henriksen et al. 2017; Roesch et al. 2018; Rompf and Odersky 2010; Rompf et al. 2013; Shaikhha et al. 2019; Steuwer et al. 2017] would apply clever optimizations (such as vertical map fusion: `map f ∘ map g = map (f ∘ g)`, to avoid creating and immediately traversing an array of heap-allocated `Complex` objects. Graph IRs represent expressions as nodes, and data dependencies between expressions as edges. This makes it easy to apply such optimizations, as the two `map` operations in the example are immediate neighbors in a dependency graph, irrespective of the intervening variable bindings.

Why Side-Effects? Clearly, it would be nice to have comparable optimizations in a full general-purpose language with side effects. But even taken in isolation, the pure functional approach of context-free rewriting has clear limitations: (1) supporting other kinds of traversals such as `map`, `reduce`, `zip` leads to an explosion of fusion rules, and (2) if both `compNums` and `conjNums` are used later, *vertical* fusion of `compNums` into `conjNums` still leaves two loops, where we need *horizontal* fusion to merge them. Through a progressive lowering approach, exposing *selected* side effects and attempting fusion in the example, *e.g.*,

```
val compNums, conjNums = new Array[Complex](100) /* explicit allocation */
for (th ← 0 until 100) {
  val re = cos(th); val im = sin(th)
  compNums(th) = Complex(re, im)               /* explicit assignment */
  conjNums(th) = Complex(re, -im)
}
```

leads to more uniform and generic handling of multiple fusion scenarios compared to pure functional simplification (cf. Section 7.1). But of course, this requires precise reasoning about effects and dependencies, while still working in a higher-order setting.

Changing Data Representation. Many other non-trivial optimizations can be performed *locally* by looking at the neighbors of a node, *e.g.*, transforming from an "array-of-struct (AoS)" to "struct-of-array (SoA)" layout that disentangles the array of complex numbers `compNums` into two arrays of the real and imaginary components. This transformation changes the data layout, yielding a better fit for SIMD or GPU execution. In the transformed code, there are no explicit `Complex` objects left:

```
val compRe, compIm, conjRe, conjIm = new Array[Double](100)
for (th ⟵ 0 until 100) {
  val re = cos(th); val im = sin(th)
  compRe(th) = re;  compIm(th) = im   /* compNums */
  conjRe(th) = re;  conjIm(th) = -im  /* conjNums */
}
```

Clearly, care must be taken here to ensure that reordered operations are noninterfering. A compiler using a graph IR can also recognize that the computation does not change the real parts of the numbers and altogether eliminate the `conjRe` array. Downstream uses of `conjNums` can share the array of real components with `compNums`.

Moving Code Around Freely. Another benefit of using a graph IR is cheap *code motion*. A code motion algorithm converts a graph into a tree and decides the hierarchy and ordering according to dependencies and heuristics (cf. Section 6). Let us add a conditional selection:

```
val result = if (cond) { (compRe, compIm) /* compNums */ }
             else      { (compRe, conjIm) /* conjNums */ }
```

If the rest of the computation only uses either branch via `result`, then the previous `conjIm` computation is wasteful. Code motion will move it directly into the else-branch:

```
val result = if (cond) { (compRe, compIm) } else {
  val conjIm = new Array[Double](100)
  for (th ⟵ 0 until 100) { conjIm(th) = -compIm(th) }
  (compRe, conjIm)
}
```

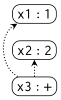
On the other hand, for blocks that may be executed multiple times (such as loops and functions), it is preferable to compute as much as possible before the block, so a code motion algorithm would attempt to place instructions as little as possible inside these blocks.

2.2 Data and Effect Dependencies

We now switch to informal but thorough explanations of our graph IR. We use A-normal form (ANF) [Flanagan et al. 1993] to represent programs as sequential blocks binding each operation to a unique variable, which permits a straightforward reading as a directed graph. Fundamental to optimizing impure higher-order programs are *dependencies* exposing local information at nodes.

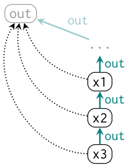
Data Dependencies. The first and most obvious kind are *data dependencies* which coincide with free-variable occurrences in terms. We draw them as directed dotted edges, connecting nodes with free occurrences to the respective nodes with binding occurrences. We draw ANF terms (*i.e.*, nodes) as white boxes labeled with their (unique) name. These nodes are also labeled with the operand of the term, but we often omit it in examples.

```
val x1 = 1
val x2 = 2
val x3 = x1 + x2
```



Effect Dependencies. We have further kinds of dependencies that induce a partial ordering over effectful operations. Such a dependency imposes a strict before-after execution/scheduling order. Consider the following program that prints a sequence of messages to standard output (`out`):

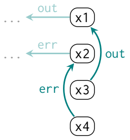
```
val x1 = println(out, "hello")   / {out ↦ last-use-out}
val x2 = println(out, "changed") / {out ↦ x1}
val x3 = println(out, "world")   / {out ↦ x2}
```



We annotate the variable bindings with effect-dependency mappings. A dependency mapping $x \mapsto y$ indicates an effectful use of the operand x by the annotated node, and y is the node where x was *last used*, or the start of a block (cf. Section 2.3). For example, `out ↦ x1` above denotes that node x2 prints to out, and the last prior print operation to out happens at node x1, therefore the execution/scheduling of x2 should happen after x1. We draw effect dependencies as directed solid edges in teal, labeled with the name of the used resource, and connect a node using the resource to the node that last used it. In the drawing on the right, all prior uses are subsumed under "…". Notably, pure expressions are recognizable by having an empty effect-dependency mapping.

Nodes can induce effects on multiple operands/resources, which will each appear in the dependency mapping's domain. Consider printing to two files out and err:

```
val x1 = println(out, "ok 1") / {out ↦ last-use-out}
val x2 = println(err, "bad")  / {err ↦ last-use-err}
val x3 = println(out, "ok 2") / {out ↦ x1}
val x4 = println(err, "bad")  / {err ↦ x2}
```



Since node x2 writes to err and only needs to depend on the last node printing to err, node x3 can depend on x1, skipping x2. Optimizations/code motion may use this fact to rearrange x2 after x3.

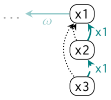
Hard and Soft Dependencies. Effect dependencies induce an unbreakable "hard" ordering requirement which is sometimes overly strict, especially in programs involving memory allocations, reads, and writes. Consider the following snippet that consecutively writes numbers to a heap-allocated cell and finally reads from that cell:

```
val x1 = new Ref(0); val x2 = (x1 := 1); val x3 = (x1 := 2); !x1
```

Note that the assignment to x1 at node x2 is immediately overwritten at node x3, and it is thus unnecessary to generate code for x2. However, the last use of x1 at x3 points to x2, and the effect dependency notion so far prevents eliminating x2. We solve this issue with *soft dependencies* (cf. Section 4.2), and by further classifying effectful uses of operands as read or writes. Nodes with only soft dependencies may be eliminated but still need to be taken into account for ordering.

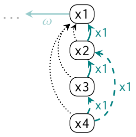
The above example exposes the "write-after-write" pattern and induces a soft dependency from x3 to x2. Another pattern inducing soft dependency is "write-after-read", for example:

```
val x1 = new Ref(0) / {ω ↦ ...}_h {}_s
val x2 = !x1        / {x1 ↦ x1}_h {}_s
val x3 = (x1 := 42) / {}_h {x1 ↦ x2}_s
```



Subscripts (h and s) distinguish the hard from soft dependencies, which are depicted by dashed teal edges. Here, x3 soft-depends on node x2, meaning that x2 can never be executed/scheduled after x3. However, for x3, it is not necessary to be executed/scheduled after x2. So we could safely eliminate x2 if it has no further use in the program. Note that reference allocations (*e.g.*, x1) induce a read effect on the node ω, which represents a global allocation capability. Once a reference is allocated, it is tracked with hard/soft dependencies, and it may have multiple soft dependencies:

```
val x1 = new Ref(0) / {ω ↦ ...}_h {}_s
val x2 = (x1 := 21) / {}_h {x1 ↦ x1}_s
val x3 = !x1        / {x1 ↦ x2}_h {}_s
val x4 = (x1 := 42) / {}_h {x1 ↦ [x2, x3]}_s
```



Read and Write Effects. A read effect (*e.g.*, at x3) results in a hard dependency on the last write. In contrast, a write effect (*e.g.*, at x2 and x4) induces soft dependencies on the last write and all reads after that write. The assignment at x2 is the first effectful use of x1, hence the soft dependency points to the node declaring x1. It is safe to eliminate x2 and x3, as long as x3 has no further occurrence.

The choice of read/write effects needs to be individually determined for each kind of operation, and our approach supports effect-specific optimizations. Consider the following examples of different kinds of "read" operations and their effects:

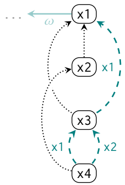
```
/* file I/O; read is also a write */            /* randomness; non-idempotent read  */
val f1 = read(fd) / {fd ↦ last-write-fd}_h {}_s  val r1 = random() / {ρ ↦ block-start}_h {}_s
val f2 = read(fd) / {fd ↦ f1}_h {}_s             val r2 = random() / {ρ ↦ block-start}_h {}_s
/* memory allocation; reads next store address */ /* references; idempotent reads, permitting CSE */
val a1 = alloc(4) / {ω ↦ block-start}_h {}_s     val x1 = !x       / {x ↦ last-write-x}_h {}_s
val a2 = alloc(4) / {ω ↦ block-start}_h {}_s     val x2 = !x       / {x ↦ last-write-x}_h {}_s
```

We categorize file I/O, memory allocation, and randomness differently from mutable references. File-system reads create hard dependencies and are considered both *read and write* effects due to the implicit change in the file cursor's state. However, soft dependency for files is unsafe, *e.g.*, removing `f1` would change the behavior of the program, even if `f1` is never used. Memory allocation involves heap reads but does not require ordering, allowing removal of unused allocations (cf. Section 7.3). Reading random numbers alters the generator ρ, but is treated as a non-idempotent read since the order is insignificant. A local CSE rule can be defined for multiple reference reads (cf. Section 5.1), which is not applicable to the other operations mentioned due to their idempotency.

Aliasing and Reachability. As shown above, the key of dependency tracking is identifying the last use of effectful resources. This is less straightforward if resources are aliased through multiple bindings, closed over by first-class functions, or other data structures. It is crucial to track aliasing to maintain fine-grained dependencies, which would otherwise become useless for optimizations.

We use reachability types [Bao et al. 2021] for tracking sets of aliased variables in types. Consider the following example of two aliased references:

```
/* x1: Ref[Int]^{x1} */
val x1 = new Ref(0)                  / {ω ↦ ...}_h {}_s
/* x2: Ref[Int]^{x1, x2} */
val x2 = x1                          / {}_h {}_s
val x3 = (x1 := 1) /* @wr(x1)    */ / {}_h {x1 ↦ x1}_s
val x4 = (x2 := 2) /* @wr(x1,x2) */ / {}_h {x1 ↦ x3, x2 ↦ x3}_s
```



The type of `x2` is an integer reference. The reachability type system annotates it with a set of variables `{x1, x2}`, *i.e.*, the aliases of `x2`. We make use of this fact at `x4`, inferring dependencies for each alias of the assignment target, informed by effect types. That is, we assign effects to reachability sets, *e.g.*, writing `x2` induces the `@wr(x1,x2)` effect on `x2` and all its aliases (dually, the `@rd(_)` effect indicates a read), and effects determine the dependency mappings (cf. Section 3). Dependency tracking thus remains precise even in the presence of aliasing.

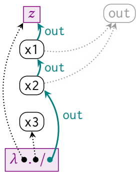
For simplicity, we conflate reachability and aliasing, but the former is actually a stronger relation, *e.g.*, `x1`'s set remains unaffected by the subsequent alias `x2`. This is a key aspect of the type system: if two entities are aliased, then their reachability sets will overlap, which is sufficient for reasoning about program properties and optimization opportunities, and avoids more complex alias analysis.

2.3 Block Structure

Instructions are implicitly grouped as blocks via their dependencies. In contrast to traditional "Sea-of-Nodes" IRs (*e.g.*, Graal [Duboscq et al. 2013] and V8), our IR's blocks have bound variables, so that we can directly represent nested blocks and function definitions without a known control-flow predecessor. "Sea-of-Nodes" IRs do not support these features.

A block ends with the return variable and summary of the effect dependencies at the block's return position, including their last effectful use. Here is an example block labeled $\lambda z.x3/\delta$:

```
{ /* z =>  block with implicit binder */
  val x1 = println(out, "str1") / {out ↦ z}
  val x2 = println(out, "str2") / {out ↦ x1}
  val x3 = 42 / {}
  x3 / {out ↦ x2}
}
```



```
 1  val c = new Ref(0)
 2
 3  /* inc : (Int=>Int @rd(c)@wr(c)){c} */
 4  def inc(x0: Int) = {
 5    val x1 = !c          / {c ↦ x0}
 6    val x2 = c := x1 + x0 / {c ↦ x1}
 7    val x3 = !c          / {c ↦ x2}
 8    x3                   / {c ↦ x3}
 9  } / {}
10
11  val u = (c := 21) / {c ↦ c}
12
13  /* function call to be inlined */
14  val r = inc(42)   / {c ↦ u}
15
16  val a = (c := 0)  / {c ↦ r}
17  val p = println(out, r) / {out ↦ ...}
```

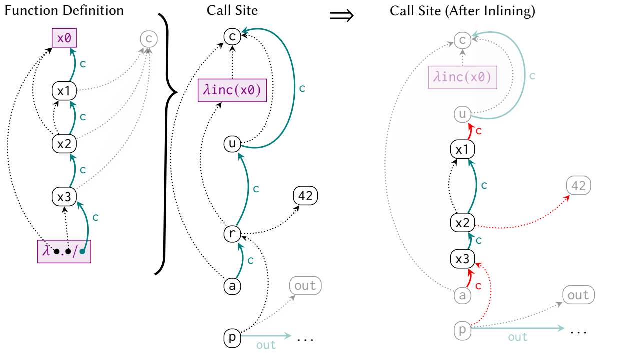


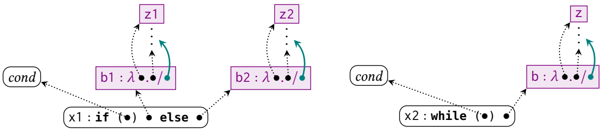
Fig. 1. Example: Function inlining and dependency rewiring. Left: Function definition using blocks with binders. Middle: Dependency of a function application. Right: Inlining the function at node r rewires the caller and callee dependencies (highlighted in red). The type of function inc indicates that it closes over the reference c and induces read/write effects on it. This effect informs the dependency on c at the call site r.

The return variable x3 of the block includes the dependency on out at node x2. Transitively, this will also include the println effect on out at node x1. The first effectful operation in a block depends on the bound variable provided by the enclosing structure, *e.g.*, a function's formal parameter.

Crucially, unlike traditional SSA/CFG IRs that use jump instructions at the terminal position to represent control transfer, our IR directly supports *nesting* and *bindings*, thus retaining a more direct representation of higher-order programs in the graph. This design choice allows us to efficiently perform frequency estimation and code motion (cf. Section 6.2).

Blocks can be used to compose larger elements, such as conditionals, loops, and function abstractions. Conditional nodes consist of a symbol referring to the conditional expression, and two nested blocks representing the two branches. Similarly, a loop node has a single symbol and a single block:

```
val x1 = if (cond) b1 else b2
val x2 = while (cond) b
```



Using the block structure, a potentially recursive function abstraction can be syntactically represented as a self-reference symbol f, a formal parameter x, and a block b, *i.e.*, $\lambda f(x).b$.

2.4 Function Abstraction

Typing Functions with Latent Effects. Free variables captured by functions are considered reachable by the function and hence occur in the function type's reachability qualifier. The following example shows a heap-allocated cell and a function that increments it and returns the new value:

```
val c = new Ref(0) /* c: Ref[Int]{c}*/
/* inc: (Int => Int @rd(c)@wr(c)){c} */
def inc(y: Int) = { val old = !c; c := old + y; val res = !c; res }
```

The comments precisely describe the type of c and inc. The function type of inc has a qualifier {c}, which is the combined reachable set of all its free variables, and @rd(c)@wr(c) indicates the function's latent effects: inc can read and write reference c.

Dependency Abstraction. Figure 1 (left) shows the ANF version of function inc with dependencies (Line 3-9). A function definition is pure and hence has no effect dependency (Line 9). However, inside the function, the formal parameter x0 is not only a placeholder for an unknown value, but

Graph IR Syntax $\boxed{\lambda^*_{\mathrm{G}}}$

$$\begin{array}{rcll@{\qquad}rcll}
g & ::= & \mathsf{x} \mid \textbf{let}\ x = b \bullet \delta\ \textbf{in}\ g & \text{Graph} & \mathsf{x},\mathsf{y},\mathsf{z} & ::= & x \mid \ell & \text{Name} \\
b & ::= & n \mid g & \text{Binding} & v & ::= & \ell & \text{Value} \\
n & ::= & \iota \mid \mathsf{x}\ \mathsf{x} \mid \textbf{ref}_{\mathsf{x}}\ \mathsf{x} \mid !\ \mathsf{x} \mid \mathsf{x} \coloneqq \mathsf{x} & \text{Graph Node} & p,q,\varepsilon,\varphi & \in & \mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \mathsf{Loc}) & \text{Qualifier/Effect} \\
\iota & ::= & c \mid \lambda x.(g \bullet \delta) \mid \textbf{ref}_{\ell}\ \ell & \text{Introduction} & S,T,U & ::= & B \mid (x : T^{q}) \to^{\varepsilon} T^{q} \mid \mathsf{Ref}\ T & \text{Type} \\
\Delta,\delta & ::= & \overline{\mathsf{x} \mapsto \mathsf{x}} & \text{Dependency} & \Gamma & ::= & \varnothing \mid \Gamma, x : T^{q} & \text{Typing Context} \\
x,y,z & \in & \mathsf{Var} & \text{Variable} & \Sigma & ::= & \varnothing \mid \Sigma, \ell : T^{q} & \text{Store Typing} \\
\ell,w & \in & \mathsf{Loc} & \text{Location} & \varepsilon_1 \rhd \varepsilon_2 & := & \varepsilon_1 \cup \varepsilon_2 & \text{Sequential Composition}
\end{array}$$

Fig. 2. Term and type syntax of the graph IR λ^*_{G}.

also a placeholder for unknown dependencies at call sites. In this example, the node that first reads c depends on the parameter x0 (Line 5), hence it will be scheduled at the block's start.

There are two abstractions here: (1) The function parameter can be a dependency for operations in the function's body; (2) functions need to return a dependency mapping that summarizes last-use information. Using them together enables a modular, context-insensitive way to specify dependency for functions. Next, we discuss how dependencies are instantiated at call sites.

Connecting and Rewiring Dependencies. Consider the call site at Line 11-17 in Figure 1, which has an application of inc in-between two operations over c (Line 14). From the type of inc (Line 3) we know that it performs effects over its captured resource c. Hence, call sites of inc should include dependencies of c, which is {c ↦ u} in this example. In the rest of the block, the next operation of c (Line 16) also depends on the application (*i.e.*, {c ↦ r}).

Figure 1 (right) shows the graphs of inc's definition, and the call site before and after inlining inc. We can observe the following difference (highlighted in red): (1) the function body replaces r, and the parameter in the body has been replaced with the call's argument 42; (2) the dependency of the body's first node pointing to the formal parameter x0 is rewired to the upstream node u at the call site; (3) any downstream dependency to the function application r is rewired with the actual last effectful operation in the function body (using the information from the function's return dependency). We call the second and third changes "dependency rewiring".

3 CORE GRAPH IR

We sketch the formal theory and metatheory of the typed graph IR λ^*_{G},[2] proving syntactic type soundness and correctness properties of effect-dependency synthesis. The core system presented here supports only "hard" dependencies, with soft dependencies being covered in Section 4.2. We also omit other non-essential features like untracked values, recursive λ-abstractions, higher-order mutable references, and type polymorphism. Those can be supported by basing the λ^*_{G} IR on polymorphic reachability types [Wei et al. 2023a].

From ANF to MNF. So far, we motivated the λ^*_{G} IR in Section 2 informally in ANF (*i.e.*, flat sequences of named atomic expressions), which is also used by our Scala LMS implementation. Now we generalize from ANF to monadic normalform (MNF) [Hatcliff and Danvy 1994], allowing nested let bindings. This permits decomposing complex transformations into more elementary steps, simplifying reasoning in λ^*_{G}'s metatheory. For instance, function inlining in LMS (Figure 1) while straightforward in its implementation is quite involved when viewed syntactically because of the need to maintain the "flat" ANF syntax, and requires simultaneous substitution, flattening, and dependency rewiring of the function body. So, using evaluation contexts, an atomic formal

[2] The full details, including soundness proofs are in the supplementary material [Bračevac et al. 2023].

Dependency Synthesis

$$\boxed{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash (n \mid g) : T^{q}\ \boldsymbol{\varepsilon} \rightsquigarrow (\boldsymbol{n} \mid \boldsymbol{g}) \bullet \delta}$$

$$\frac{c \in B}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash c : B^{\varnothing}\ \boldsymbol{\varnothing} \rightsquigarrow c \bullet \varnothing}\ (\rightsquigarrow\text{-CST})$$

$$\frac{\mathsf{x} : (\mathsf{Ref}\ B)^{q} \in [\Sigma \mid \Gamma]^{\varphi} \quad \mathsf{y} : B^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathsf{x} \coloneqq \mathsf{y} : \mathsf{Unit}^{\varnothing}\ \mathsf{x} \rightsquigarrow \mathsf{x} \coloneqq \mathsf{y} \bullet \Delta|_{\mathsf{x}*}}\ (\rightsquigarrow\text{-:=})$$

$$\frac{[\Sigma \mid \Gamma, x : T^{p}]^{q,x} \bullet \mapsto x \vdash g : U^{r}\ \boldsymbol{\varepsilon} \rightsquigarrow \boldsymbol{g} \bullet \delta \qquad q \subseteq \varphi}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \lambda x.g : ((x : T^{p}) \to^{\boldsymbol{\varepsilon}} U^{r})^{q}\ \boldsymbol{\varnothing} \rightsquigarrow (\lambda x.\boldsymbol{g} \bullet \delta) \bullet \varnothing}\ (\rightsquigarrow\text{-ABS})$$

$$\frac{\mathsf{x} : T^{q} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathsf{x} : T^{\mathsf{x}}\ \boldsymbol{\varnothing} \rightsquigarrow \mathsf{x} \bullet \varnothing}\ (\rightsquigarrow\text{-RET})$$

$$\frac{\begin{array}{c}\mathsf{x} : \big((z : T^{p*\cap q*}) \to^{\boldsymbol{\varepsilon}} U^{r}\big)^{q} \in [\Sigma \mid \Gamma]^{\varphi} \\ \mathsf{y} : T^{p} \in [\Sigma \mid \Gamma]^{\varphi} \qquad \theta = [p/z] \\ z \notin \mathrm{fv}(U) \qquad \boldsymbol{\varepsilon} \subseteq q, z \qquad r \subseteq \varphi, z\end{array}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathsf{x}\ \mathsf{y} : (U^{r}\ \boldsymbol{\varepsilon})\theta \rightsquigarrow \mathsf{x}\ \mathsf{y} \bullet \Delta|_{(\boldsymbol{\varepsilon}\theta)*}}\ (\rightsquigarrow\text{-APP})$$

$$\frac{\begin{array}{c}[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash b : S^{p}\ \boldsymbol{\varepsilon_1} \rightsquigarrow \boldsymbol{b} \bullet \delta_1 \\ [\Sigma \mid \Gamma, x : S^{p*\cap\varphi*}]^{\varphi,x} \bullet \Delta, (\boldsymbol{\varepsilon_1}*, x) \mapsto x \vdash g : T^{q}\ \boldsymbol{\varepsilon_2} \rightsquigarrow \boldsymbol{g} \bullet \delta_2 \\ \theta = [p/x] \quad x \notin \mathrm{fv}\,(T)\end{array}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathbf{let}\ x = b\ \mathbf{in}\ g : (T^{q}\ \boldsymbol{\varepsilon_1} \rhd \boldsymbol{\varepsilon_2})\theta \rightsquigarrow (\mathbf{let}\ x = \boldsymbol{b} \bullet \delta_1\ \mathbf{in}\ \boldsymbol{g}) \bullet \delta_1, \delta_2[x \rightsquigarrow \Delta|_{p*}]}\ (\rightsquigarrow\text{-LET})$$

$$\frac{\mathsf{x} : \mathsf{Alloc}^{q} \in [\Sigma \mid \Gamma]^{\varphi} \quad \mathsf{y} : B^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathbf{ref}_{\mathsf{x}}\ \mathsf{y} : (\mathsf{Ref}\ B)^{\varnothing}\ \mathsf{x} \rightsquigarrow \mathbf{ref}_{\mathsf{x}}\ \mathsf{y} \bullet \Delta|_{\mathsf{x}*}}\ (\rightsquigarrow\text{-REF})$$

$$\frac{\mathsf{x} : (\mathsf{Ref}\ B)^{q} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash\ !\,\mathsf{x} : B^{\varnothing}\ \mathsf{x} \rightsquigarrow\ !\,\mathsf{x} \bullet \Delta|_{\mathsf{x}*}}\ (\rightsquigarrow\text{-!})$$

$$\frac{\begin{array}{c}[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash b : S^{p}\ \boldsymbol{\varepsilon_1} \rightsquigarrow \boldsymbol{b} \bullet \delta_1 \\ \Sigma \mid \Gamma \vdash S^{p}\ \boldsymbol{\varepsilon_1} <: T^{q}\ \boldsymbol{\varepsilon_2} \\ q, \boldsymbol{\varepsilon_2} \subseteq \varphi\end{array}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash b : T^{q}\ \boldsymbol{\varepsilon_2} \rightsquigarrow \boldsymbol{b} \bullet \Delta|_{\boldsymbol{\varepsilon_2}}}\ (\rightsquigarrow\text{-SUB})$$

Name Lookup

$$\boxed{\mathsf{x} : T^{q} \in [\Sigma \mid \Gamma]^{\varphi}}$$

$$\frac{x : T^{q} \in \Gamma \qquad q, x \subseteq \varphi}{x : T^{q} \in [\Sigma \mid \Gamma]^{\varphi}}\ (\text{L-VAR}) \qquad \frac{\ell : T^{q} \in \Sigma \qquad q, \ell \subseteq \varphi}{\ell : T^{q} \in [\Sigma \mid \Gamma]^{\varphi}}\ (\text{L-LOC})$$

Fig. 3. Dependency synthesis and select type checking rules of the graph IR λ^{*}_{G}. The remaining type checking rules coincide with the synthesis rules when ignoring the output to the right of $\rightsquigarrow$.

inlining step on ANF would look like

$$\frac{(f = \lambda y.A_2[\,z/\delta_2\,]) \in A_1 \qquad \theta_1 = [u/y] \qquad \theta_2 = [z/x] \qquad \rho_1 = [y \rightsquigarrow \delta_1] \qquad \rho_2 = [x \rightsquigarrow \delta_2]}{A_1[\,\mathbf{let}\ x = f\ u/\delta_1\ \mathbf{in}\ a\,] \longrightarrow A_1[\,A_2[\,a\theta_2\rho_2\,]\theta_1\rho_1\,]}$$

where we are careful to replace the function call at x with the entire function body, plugging the continuation a into function block A_2's return position, before plugging the inlined result back into the calling context A_1. Plus, we need to patch up the formal parameter and block return by substitution and dependency rewiring (cf. Section 3.3).

Formally proving the soundness of such a rule within the constraints of ANF is tedious, and one is better off with more fine-grained reductions. MNF enables modeling function inlining by a form of β-reduction, simply replacing the call $f\ u$ with an instance of the function's body at the binding x above. We can further "flatten" the nested term by separate administrative liftings.

3.1 Syntax

We represent programs in the MNF syntax (Figure 2) as terms g, where let bindings denote named nodes or nested graphs, variable occurrences indicate edges for data dependencies, and δ maps indicate edges for effect dependencies. A node n is either an introduction form (*i.e.*, constants, functions, allocations), or an operation on let-bound names, *i.e.*, applications, reference allocations,

Store Terms, Graph Term Contexts, Binding Contexts

$$\sigma ::= \varnothing \mid \sigma,\ \mathbf{let}_{\mathsf{s}}\ \ell = \iota \qquad G ::= \square \bullet \delta \mid (\mathbf{let}\ x = G\ \mathbf{in}\ g) \bullet \delta \qquad B ::= (\mathbf{let}\ x = \square\ \mathbf{in}\ g) \bullet \delta \mid (\mathbf{let}\ x = B\ \mathbf{in}\ g) \bullet \delta$$

Reduction Rules $\boxed{\sigma \mid z.g \bullet \delta \longrightarrow_{\mathrm{G}} \sigma \mid z.g \bullet \delta}$

$$\begin{array}{rcll}
\sigma \mid z.B[\,\ell_1\ \ell_2 \bullet \delta_1\,] & \longrightarrow_{\mathrm{G}} & \sigma \mid z.B[\,(g \bullet \delta_2)[x \rightsquigarrow \delta_1][\ell_2/x]\,] & (\beta) \\
& & \sigma = \sigma_1,\ \mathbf{let}_{\mathsf{s}}\ \ell_1 = \lambda x.g \bullet \delta_2,\ \sigma_2 & \\
& & \mathrm{dom}(\delta_1) \subseteq \mathrm{dom}(\sigma),\ \mathrm{cod}(\delta_1) \subseteq \{z\} & \\
\sigma \mid z.G[\,\mathbf{let}\ x = \ell \bullet \delta\ \mathbf{in}\ g\,] & \longrightarrow_{\mathrm{G}} & \sigma \mid z.G[\,g[x \rightsquigarrow \delta][\ell/x]\,] & (\textsc{let}) \\
& & \mathrm{dom}(\delta) \subseteq \mathrm{dom}(\sigma),\ \mathrm{cod}(\delta) \subseteq \{z\} & \\
\sigma \mid z.G[\,\mathbf{let}\ x = \iota \bullet \delta\ \mathbf{in}\ g\,] & \longrightarrow_{\mathrm{G}} & \sigma, \mathbf{let}_{\mathsf{s}}\ \ell = \iota \mid z.G'[\,g[x \rightsquigarrow \delta][\ell/x]\,] & (\textsc{intro}) \\
& & \ell \notin \mathrm{dom}(\sigma),\ \mathrm{cod}(\delta) \subseteq \{z\},\ G' = G\langle \iota : z.\ell \rangle & \\
\sigma,\ \mathbf{let}_{\mathsf{s}}\ \ell = \mathbf{ref}_{\mathsf{w}}\ \ell',\ \sigma' \mid z.B[\,!\ell \bullet \delta\,] & \longrightarrow_{\mathrm{G}} & \sigma,\ \mathbf{let}_{\mathsf{s}}\ \ell = \mathbf{ref}_{\mathsf{w}}\ \ell',\ \sigma' \mid z.B[\,\ell' \bullet \delta\,] & (\textsc{deref}) \\
& & \mathrm{dom}(\delta) \subseteq \mathrm{dom}(\sigma),\ \mathrm{cod}(\delta) \subseteq \{z\} & \\
\sigma,\ \mathbf{let}_{\mathsf{s}}\ \ell = \mathbf{ref}_{\mathsf{w}}\ \ell',\ \sigma' \mid z.B[\,\ell := \ell'' \bullet \delta\,] & \longrightarrow_{\mathrm{G}} & \sigma,\ \mathbf{let}_{\mathsf{s}}\ \ell = \mathbf{ref}_{\mathsf{w}}\ \ell'',\ \sigma' \mid z.B[\,\mathsf{unit} \bullet \delta\,] & (\textsc{assign}) \\
& & \mathrm{dom}(\delta) \subseteq \mathrm{dom}(\sigma),\ \mathrm{cod}(\delta) \subseteq \{z\} &
\end{array}$$

Contextual Effect Propagation $\boxed{G\langle \iota : z.\ell \rangle}$

$$\begin{array}{lcl}
G\langle c : z.\ell \rangle & = & G \\
G\langle \lambda x.g \bullet \delta : z.\ell \rangle & = & G \\
(\square \bullet \delta)\langle \mathbf{ref}_{\mathsf{w}}\ \ell_1 : z.\ell_2 \rangle & = & \square \bullet \delta, \ell_2 \mapsto z \\
((\mathbf{let}\ x = G\ \mathbf{in}\ g) \bullet \delta)\langle \mathbf{ref}_{\mathsf{w}}\ \ell_1 : z.\ell_2 \rangle & = & (\mathbf{let}\ x = G\langle \mathbf{ref}_{\mathsf{w}}\ \ell_1 : z.\ell_2 \rangle\ \mathbf{in}\ g) \bullet \delta, \ell_2 \mapsto z
\end{array}$$

Fig. 4. Call-by-value reduction for λ^{*}_{G} with runtime dependency checking.

dereferences, or assignments. Operands can either be store locations or term variables, and we use the category of names to refer to either. For a uniform treatment of effect introductions, we let allocations $\mathbf{ref}_{\mathsf{x}}\ \mathsf{y}$ take an explicit capability parameter x granting permission to allocate. We consider only store locations as values, because the operational semantics (Section 3.5) retains both immutable and mutable variables, so that term substitution becomes renaming.

3.2 Reachability Types, Aliasing and Separation

The λ^{*}_{G} type system tracks reachability qualifiers q and effects $\boldsymbol{\varepsilon}$ (Figure 2), which are both sets of variables and store locations. Qualifiers track which other values are reachable/aliased from a computation's result, whereas effects indicate which variables/node names are the target of an effect. Tracking reachable variables in types enables reasoning about many useful properties of higher-order programs (cf. [Bao et al. 2021]).

Observable Separation. In the case of function types $((x : S^{p}) \to^{\boldsymbol{\varepsilon}} T^{q})^{r}$, the attached qualifier r includes the free variables. Function types are dependent in their argument x, which may appear free in the latent effect $\boldsymbol{\varepsilon}$, codomain type T, and codomain qualifier q and refers to the qualifier of arguments at call sites. The reachability type system guarantees *observable separation*, *i.e.*, the qualifier r above certifies that functions of this type do not observe any reachable names beyond r in the context. That limits variables/node names in effects, and thus potential effect dependencies. More generally, we assign a *filter* φ to typing judgments (Figure 3) which restricts access to the context, akin to substructural type systems, *i.e.*, the subject g can only observe the names in φ.

On-demand Reachability. Following Wei et al. [2023a], we choose a variant of Bao et al.'s system which assigns reachability qualifiers (and effects) in an on-demand manner. That is, type assignment assigns minimal reachability sets to expressions, *e.g.*, variables initially only reach themselves in rule ($\rightsquigarrow$-RET), as opposed to an "eager" assignment where all reachable aliases in context are immediately assigned. To resolve maximal dependencies, we compute the transitive reachability closure in the given context, written $q*$ resp. $\boldsymbol{\varepsilon}*$, when appropriate. Its formal definition is in the supplementary material ([Bračevac et al. 2023], Figure 3).

3.3 Dependencies

Effect dependencies are finite maps δ from names to names attached to let bindings and function bodies. Dependencies come with the standard "update" operator which is associative:

$$(\delta_1, \delta_2)(\mathsf{x}) = \delta_2(\mathsf{x}),\ \mathsf{x} \in \mathrm{dom}(\delta_2) \qquad\qquad (\delta_1, \delta_2)(\mathsf{x}) = \delta_1(\mathsf{x}),\ \mathsf{x} \notin \mathrm{dom}(\delta_2)$$

and a restriction operator of the domain to a given set (abusing set notation): $\delta|_\alpha := \{\mathsf{x} \mapsto \mathsf{y} \in \delta \mid \mathsf{x} \in \alpha\}$, and we also define a removal operator, *i.e.*, $\delta - \alpha := \{\mathsf{x} \mapsto \mathsf{y} \in \delta \mid \mathsf{y} \notin \alpha\}$ removing all mappings pointing into α. In symmetry with substitutions on graphs and qualifiers, there is a notion of substitution (or rewiring, rerouting), over dependencies:

$$\delta_1[\mathsf{x} \rightsquigarrow \delta_2] := \delta_1 - \{\mathsf{x}\}, \{\mathsf{y} \mapsto \mathsf{z} \in \delta_2 \mid \mathsf{y} \mapsto \mathsf{x} \in \delta_1\},$$

which is rerouting the dependency target x via δ_2. That is, we substitute some of the targets in dependency mappings, which happens when the name x is replaced, *e.g.*, when x is the formal parameter of a function at a call site.

3.4 Type-Effect-Dependency System

From reachability qualifiers and effects, we can extract precise effect dependencies, as motivated in Section 2.2, and formalized by type-directed dependency synthesis (Figure 3).

The judgment $[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash g : T^{q}\ \boldsymbol{\varepsilon} \rightsquigarrow \boldsymbol{g} \bullet \delta$ intuitively means: Given a dependency map Δ recording the ambient last uses of each variable/location bound in Γ and Σ, the graph g synthesizes to the dependency-annotated $\boldsymbol{g}$ and has local dependencies δ.

Ignoring the teal parts, we obtain a version of Bao et al.'s type system for MNF, where effects track those variables/locations and their reachable aliases which are the operand of an effect operation.

Synthesis. Dependency synthesis follows a simple idea: We track in the map Δ attached to the context (essentially a coeffect) the *last use* of each bound name as the target of an effect invocation. If the graph's effect is $\boldsymbol{\varepsilon}$, then the generated local dependency δ is always the projection $\delta = \Delta|_{\boldsymbol{\varepsilon}*}$, *i.e.*, we attach the current last uses of the variables in the effect. The last-uses map Δ is updated at let bindings ($\rightsquigarrow$-LET), *i.e.*, if the bound subgraph has effect $\boldsymbol{\varepsilon}_1$, then we synthesize the body with $\Delta, (\boldsymbol{\varepsilon}_1*, x) \longrightarrow x$, *i.e.*, the last use of the variables in $\boldsymbol{\varepsilon}_1*$ is at x, and x points to itself since it is new. In contrast, synthesizing functions ($\rightsquigarrow$-ABS) sets all last uses to the parameter x, abstracting over all possible call sites. The shorthand $\mapsto x$ denotes the map Δ that assigns x to all names in context.

Typing nodes and graphs may use subsumption ($\rightsquigarrow$-SUB) to scale (1) qualifiers (increasing aliasing), (2) effects, and (3) dependency mappings (remaining consistent with the effect scaling). The subtyping rules are defined in the supplement [Bračevac et al. 2023]. To summarize, we can prove that synthesis is essentially a function over typing derivations of Bao et al.'s system in MNF:

Lemma 3.1 (Synthesis Properties).

(1) Dependencies mirror effects: $[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash g : T^{q}\ \boldsymbol{\varepsilon} \rightsquigarrow \boldsymbol{g} \bullet \delta$*, then* $\delta = \Delta|_{\boldsymbol{\varepsilon}*}$.
(2) Synthesis is sound: If $[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash g : T^{q}\ \boldsymbol{\varepsilon} \rightsquigarrow \boldsymbol{g} \bullet \delta$*, then* $[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \boldsymbol{g} : T^{q}\ \boldsymbol{\varepsilon}$.
(3) Synthesis is total: If $[\Sigma \mid \Gamma]^{\varphi} \vdash_{\mathsf{M}} g : T^{q}\ \boldsymbol{\varepsilon}$ *then* $\forall\Delta.\ \exists\boldsymbol{g}.\ \exists\delta.\ [\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash g : T^{q}\ \boldsymbol{\varepsilon} \rightsquigarrow \boldsymbol{g} \bullet \delta$.

3.5 Dependency-Checking Operational Semantics

Figure 4 shows the call-by-value operational semantics for λ^*_{G}. Unlike standard formulations, it keeps both mutable and immutable values in the store, and static dependencies as part of the terms.

Reductions occur over runtime configurations $\sigma \mid z.g \bullet \delta$, attaching a block start variable z to the graph term g with its dependency δ. z is chosen to not be a free variable of g and to check if a dependency has been evaluated. The next operation's dependencies will point to z, ensuring all current node dependencies are resolved. Well-typed terms do not exhibit dependency violations, meaning dependencies correctly reflect the runtime execution order of effects.

As motivated in Section 2.4, function applications (β) rewire function and call-site dependencies, similar to let-binding reduction in (LET). Rule (INTRO) commits an introduction form into the store, allocating a fresh location. For reference introductions, the fresh location is included in an effect and must be inserted/propagated in dependencies along the evaluation context spine. The other rules for dereference and assignment are standard.

3.6 Metatheory

The λ^*_{G} graph IR's type system is sound with respect to the dependency-checking operational semantics. We only state the theorems here, their proofs can be found in [Bračevac et al. 2023]:

THEOREM 3.2 (PROGRESS). *If* $[\Sigma \mid \varnothing]^{\mathrm{dom}(\Sigma)} \vdash z.g \bullet \delta : T^{q}\ \varepsilon$, *then either* g *is a location* $\ell \in \mathrm{dom}(\Sigma)$, *or for any store* σ *where* $[\Sigma \mid \varnothing]^{\mathrm{dom}(\Sigma)} \vdash \sigma$, *there exists a graph term* g', *store* σ', *and dependency* δ' *such that* $\sigma \mid z.g \bullet \delta \longrightarrow_{\mathrm{G}} \sigma' \mid z.g' \bullet \delta'$.

THEOREM 3.3 (PRESERVATION).

$$\frac{[\Sigma \mid \varnothing]^{\mathrm{dom}(\Sigma)} \vdash z.g \bullet \delta : T^{q}\ \varepsilon \quad \Sigma \mid \varnothing \bullet \mapsto z.z\ \mathrm{ok} \quad [\Sigma \mid \varnothing]^{\mathrm{dom}(\Sigma)} \bullet \mapsto z \vdash \sigma \quad \sigma \mid z.g \bullet \delta \longrightarrow_{\mathrm{G}} \sigma' \mid z.g' \bullet \delta'}{\exists \Sigma' \supseteq \Sigma.\ \exists p \subseteq \mathrm{dom}(\Sigma' \setminus \Sigma).\quad [\Sigma' \mid \varnothing]^{\mathrm{dom}(\Sigma')} \vdash z.g' \bullet \delta' : T^{q,p}\ \varepsilon, p \quad \Sigma' \mid \varnothing \bullet \mapsto z.z\ \mathrm{ok} \quad [\Sigma' \mid \varnothing]^{\mathrm{dom}(\Sigma')} \bullet \mapsto z \vdash \sigma'}$$

COROLLARY 3.4 (DEPENDENCY SAFETY). *Evaluation respects the order of effect dependencies for well-typed* λ^*_{G} *terms,* i.e., *an operation is executed only if all its dependencies are resolved in the store.*

Conclusion. Dependency synthesis is determined entirely by effect typing through modular and local reasoning. Effect dependencies are term-level witnesses of type-level effects on reachable variables in the program's context. Well-typed graphs carry provably correct dependencies. They have no direct operational meaning, but induce edges when viewing λ^*_{G} terms as graphs, giving localized insights into the program's behavior exploitable by optimizations (Sections 5 and 6).

4 EXTENSIONS

4.1 Structured Expressions

Based on the treatment of λ abstractions in λ^*_{G}, conditionals, loops, comprehensions, and other structured expressions/nodes become straightforward variations, *e.g.*, we model conditionals as nodes: $n ::= \cdots \mid \mathbf{if}\ (\mathsf{x})\ g\ \mathbf{else}\ g$. Typing and effects for conditionals are standard:

$$\frac{\begin{array}{c}\mathsf{x} : \mathsf{Bool}^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi} \\ [\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash g_1 : T^{q}\ \varepsilon_1 \rightsquigarrow g_1 \bullet \delta_1 \qquad [\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash g_2 : T^{q}\ \varepsilon_2 \rightsquigarrow g_2 \bullet \delta_2\end{array}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathbf{if}\ (\mathsf{x})\ g_1\ \mathbf{else}\ g_2 : T^{q}\ \varepsilon_1 \cup \varepsilon_2 \rightsquigarrow \mathbf{if}\ (\mathsf{x})\ g_1\ \mathbf{else}\ g_2 \bullet \delta_1, \delta_2} \qquad (\textsc{if})$$

Branches should have the same type and analogously the same reachability qualifier, which can be mediated by subtyping. As with most effect systems, the branch effects are composed by a notion of join which is set union in our simple effect system, and that translates to sequential composition of the branch dependencies in the output, by the properties of dependency calculation (Lemma 3.1).

4.2 Hard and Soft Dependency

In Section 2.2, we have discussed examples and optimizations enabled by distinguishing hard and soft dependencies. During code generation, a node that is only soft-depended by other nodes is considered dead, and therefore is not scheduled (cf. Section 6).

If node A hard-depends on node B, then B must be executed (or scheduled) before A. This is the default notion for the base λ^*_{G} system (Section 3). In contrast, if A *soft-depends* on B, then B should never be scheduled after A, but B might not be scheduled even if A is scheduled. The entire formal system and reasoning principles of λ^*_{G} carry over into a system with hard and soft dependencies as presented in this section. The difference is the change in the effect and dependency structure, *i.e.*, effects are split into reads and writes, which induce hard dependencies (the previous section's notion) and soft dependencies, respectively. This is a product composition of structures:

$$\begin{array}{llll llll} \mathsf{H}, \mathsf{h} & ::= & \overline{\mathsf{x} \mapsto \mathsf{x}} & \text{Hard Dependencies} & \Delta, \delta & ::= & \mathsf{h}; \mathsf{s} & \text{Dependencies} \\ \mathsf{S}, \mathsf{s} & ::= & \overline{\mathsf{x} \mapsto \overline{\mathsf{x}}} & \text{Soft Dependencies} & \boldsymbol{\varepsilon} & ::= & \mathsf{r}{:}q; \mathsf{w}{:}q & \text{Effects} \end{array}$$

Hard dependencies h have the same structure as before (assign at most one dependency), whereas soft dependencies s map variables to sets of variables. By overloading each dependency and effect operator in the metatheory with versions acting on products, we can reuse the system in Figure 3 with minimal adjustments. Effectful primitives need to label their effects as read or write, *i.e.*,

$$\frac{\mathsf{x} : \mathsf{Alloc}^{q} \in [\Sigma \mid \Gamma]^{\varphi} \qquad \mathsf{y} : B^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathbf{ref}_{\mathsf{x}}\ \mathsf{y} : (\mathsf{Ref}\ B)^{\varnothing}\ \mathsf{r}(\mathsf{x}) \rightsquigarrow \mathbf{ref}_{\mathsf{x}}\ \mathsf{y} \bullet \Delta|_{\mathsf{r}(\mathsf{x})*}} \qquad (\rightsquigarrow\text{-REF})$$

$$\frac{\mathsf{x} : (\mathsf{Ref}\ B)^{q} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash\ !\,\mathsf{x} : B^{\varnothing}\ \mathsf{r}(\mathsf{x}) \rightsquigarrow\ !\,\mathsf{x} \bullet \Delta|_{\mathsf{r}(\mathsf{x})*}} \qquad (\rightsquigarrow\text{-!})$$

$$\frac{\mathsf{x} : (\mathsf{Ref}\ B)^{q} \in [\Sigma \mid \Gamma]^{\varphi} \qquad \mathsf{y} : B^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathsf{x} \coloneqq \mathsf{y} : \mathtt{Unit}^{\varnothing}\ \mathsf{w}(\mathsf{x}) \rightsquigarrow \mathsf{x} \coloneqq \mathsf{y} \bullet \Delta|_{\mathsf{w}(\mathsf{x})*}} \qquad (\rightsquigarrow\text{-:=})$$

where $\mathsf{r}(q) = (\mathsf{r} : q\,;\,\mathsf{w} : \varnothing)$ and $\mathsf{w}(q) = (\mathsf{r} : \varnothing\,;\,\mathsf{w} : q)$. Allocations classify as a "read" on the capability operand, dereferencing induces a read on the reference itself and its reachable aliases, and assignment induces a write. Plus, updating the last uses Δ at let bindings and projecting the local dependency from effects $\Delta|_{\varepsilon}$ need changing, to implement the informal policy on read and write effects motivated in Section 2.4. We use the hard dependency in Δ to track the last write of any variable/location in context, whereas its soft dependency tracks all the reads on a variable since it was last written.

Projections need to merge the hard dependencies of a write effect into its soft dependencies, and project the hard dependencies for reads, *i.e.*, $(\mathsf{h}; \mathsf{s})|_{(\mathsf{r}:q\,;\,\mathsf{w}:p)} \coloneqq (\mathsf{h}|_q; \mathsf{h}|_p \sqcup \mathsf{s}|_p)$.

Last-use updates at let bindings need to reset the recorded last reads for any written variable, and add the bound variable to the last reads for any written variable, *i.e.*,

$$(\mathsf{h}; \mathsf{s}) \oplus_x (\mathsf{r} : q\,;\,\mathsf{w} : p) \coloneqq (\mathsf{h}, (p, x) \mapsto x\,;\ (\mathsf{s}, (p, x) \mapsto \varnothing) \oplus_x q),$$

where x is the let-bound variable and $\mathsf{s} \oplus_x q \coloneqq \mathsf{s}, \{y \mapsto \mathsf{s}(y), x \mid y \in q\}$.

In future work, we would like to develop a generic theory of graph IRs that is parametric in such effect and dependency structures. Bao et al. [2021]'s direct style system already proposes one half of the solution by adopting Gordon [2021]'s effect quantales. While we present a specific instance, nothing prevents a graph IR definition that is parametric in such structures.

```
def Tensor(shape: Seq, f: Int => Int): Tensor = {
  val builder = mkTensorBuilder(shape)
  val lp = forLoop(shape, i => builderAdd(builder, i, f(i)))    / {builder ↦ builder}
  builderRes(builder)                                           / {builder ↦ lp}
}
```

(a) The Tensor DSL internally uses mutable builders. The `forLoop` IR node named `lp` induces a corresponding dependency on the `builder` object. Sum is defined similarly.

```
def mkTensorBuilder: Seq                   => TensorBuilder       @rd(world)
def builderAdd     : (bld:TensorBuilder)  => Int => Int => Unit  @wr(bld)
def builderRes     : (bld:TensorBuilder)  => Tensor              @kl(bld)  @rd(bld)
def mkSumBuilder   : Int                   => SumBuilder          @rd(world)
def sumBuilderAdd  : (sbld:SumBuilder)    => Int => Int => Unit  @wr(sbld)
def sumBuilderRes  : (sbld:SumBuilder)    => Sum                 @kl(sbld) @rd(sbld)
```

(b) Builder APIs annotated with read/write/kill effects. The destructive "kill" effect (`kl`, cf. Section 4.3) on `builderRes` and `sumBuilderRes` signifies that the builder objects are frozen and can no longer be mutated.

Fig. 5. Builder and Tensor APIs.

4.3 Tracking Typestate with Destructive "Kill" Effects

In addition to read/write effects, we can further add destructive "kill" effects as sketched in Bao et al. [2021]. They can model memory deallocation, ownership transfer, and move semantics. In the context of our graph IR, the type system prevents any further dependencies on a killed node. This is exemplified in a snippet using the tensor DSL in Figure 5b, where converting a mutable builder object `bld` into an immutable result tensor via `builderRes(bld)` induces the kill effect to disable any further mutation:

```
  val t1: Tensor = builderRes(bld)                  / @kl(bld) {bld ↦ ...}
  forLoop(Seq(100), i => builderAdd(bld, i, 1+2*i)) /* error */
```

The attempt to mutate the tensor builder using `builderAdd` would not type check, since the write effect on `bld` induced by `builderAdd` introduces a dependency to the killed `bld`. In practice, the DSL developer needs to properly annotate the kill effects on operations in the DSL of interest. In Section 7.1, we present a case study of loop fusion using an extended set of tensor APIs with these effects.

5 GRAPH-LEVEL OPTIMIZATIONS

In this section, we examine optimizations that rewrite or transform the dependency structure. We first study domain-agnostic equational transformation rules over the graph IR that justify the later code motion and generic optimizations. Then we discuss how to realize several optimizations for higher-order programs.

5.1 Equational Theory

Figure 6 shows equational rules for λ^*_{G} with soft and hard dependencies (Section 4.2) specifying an equivalence over graph-IR programs.

The first two rules (COMM) and (λ-HOIST) correspond to code motion. Rule (COMM) permits reordering of two consecutive let-bindings, as long as the second binding has no data/hard/soft dependency on the first. Rule (λ-HOIST) permits hoisting a node or nested graph b out of the body of a function into its outer scope. Since a function's effects are latent, hoisting is only permissible if b is pure. One can determine purity by simply checking that hard and soft dependencies are absent, since dependency synthesis always results in non-empty dependency mappings for effectful terms. Furthermore, there should be no data dependency on nodes that are locally bound in the function, *i.e.*, no locally bound variable in the context of the function body appears free in b.

Other rules allow us to shrink the program in a semantics-preserving way. The dead-code elimination rule (DCE) just needs to determine the absence of downstream data dependencies ($x \notin \mathrm{fv}(g)$), and no hard dependencies ($x \notin \mathrm{FHD}(g)$). In that case, any soft dependency of the

$$C ::= \square \mid \textbf{let } x = C \bullet \delta \textbf{ in } g \mid \textbf{let } x = (\lambda y.C \bullet \delta) \bullet \delta \textbf{ in } g \mid \textbf{let } x = b \bullet \delta \textbf{ in } C$$

(comm)

$$\frac{x \notin FV(b_2) \cup \mathrm{dom}(\mathsf{h}') \cup \mathrm{dom}(\mathsf{s}') \cup \mathrm{cod}(\mathsf{h}') \cup \mathrm{cod}(\mathsf{h}') \qquad \mathrm{dom}(\mathsf{h}) \cap \mathrm{dom}(\mathsf{h}') = \mathrm{dom}(\mathsf{s}) \cap \mathrm{dom}(\mathsf{s}') = \varnothing}{C[\, \textbf{let } x = b_1 \bullet \mathsf{h}; \mathsf{s} \textbf{ in let } y = b_2 \bullet \mathsf{h}'; \mathsf{s}' \textbf{ in } g] \equiv C[\, \textbf{let } x = b_2 \bullet \mathsf{h}'; \mathsf{s}' \textbf{ in let } y = b_1 \bullet \mathsf{h}; \mathsf{s} \textbf{ in } g]}$$

(λ-hoist)

$$\frac{\mathrm{fv}(b) \cap (\mathrm{BV}(C_2) \cup \{x\}) = \varnothing}{C_1[\, \textbf{let } f = \lambda x.C_2[\, \textbf{let } y = b \bullet \varnothing; \varnothing \textbf{ in } g_1 \,] \textbf{ in } g_2 \,] \equiv C_1[\, \textbf{let } y = b \bullet \varnothing; \varnothing \textbf{ in let } f = \lambda x.C_2[\, g_1 \,] \textbf{ in } g_2 \,]}$$

(dce)

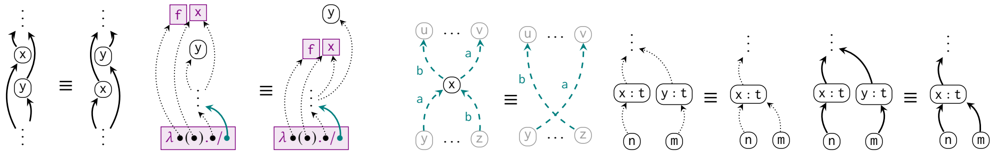
$$\frac{x \notin \mathrm{fv}(g) \qquad x \notin \mathrm{FHD}(g)}{C[\, \textbf{let } x = b \bullet \mathsf{h}; \mathsf{s} \textbf{ in } g \,] \equiv C[\, g[x \rightsquigarrow \mathsf{s}] \,]}$$

(cse)

$$C[\, \textbf{let } x = b \bullet \varnothing; \varnothing \textbf{ in let } y = b \bullet \varnothing; \varnothing \textbf{ in } g \,] \equiv C[\, \textbf{let } x = b \bullet \varnothing; \varnothing \textbf{ in } g[y/x] \,]$$

(e-cse)

$$\frac{b \text{ is idempotent}}{C[\, \textbf{let } x = b \bullet \mathsf{h}; \mathsf{s} \textbf{ in let } y = b \bullet \mathsf{h}; \mathsf{s} \textbf{ in } g \,] \equiv C[\, \textbf{let } x = b \bullet \mathsf{h}; \mathsf{s} \textbf{ in } g[y/x] \,]}$$



(a) COMM (b) λ-HOIST (c) DCE (d) CSE (e) E-CSE

Fig. 6. Equational rules and their visualizations for λ^*_{G} with hard and soft dependencies. Black thick edges represent either a data, hard, or soft dependency.

eliminated binding has to be rewired downstream to maintain consistency. This rule is powerful in combination with soft dependencies, and can optimize memory access patterns such as write-after-write (Section 2.2). In general compilers, we need a termination analysis to apply DCE, which we omit here in the context of DSLs. The common subexpression elimination rule (cse) removes *pure* expressions if the same expression has been defined previously. Additionally, *effectful* redundant expressions can be removed if the node/subgraph b is idempotent (e-cse), *e.g.*, consecutively reading a reference twice (Section 2.2).

Soundness. We apply the logical type soundness approach [Timany et al. 2022] to support relational reasoning about contextual equivalence of two programs in the direct-style type-and-effect system λ^*_{ε} ([Bračevac et al. 2023] Sec. 7). We build on a framework for modeling reachability types with logical relations by Bao et al. [2023].

A key insight is that the proof based on logical relations in the direct-style calculus is sufficient to argue for contextual equivalence in MNF (supplement Sec. 8) because: (1) MNF is a sublanguage of the direct-style language, and (2) dependencies are entirely determined by effects (Lemma 3.1). Effect dependencies have no operational meaning beyond asserting that they respect the observed call-by-value evaluation order of effects (Corollary 3.4). Therefore, we can appeal to the logical relations for λ^*_{ε} by dependency erasure and re-synthesis to derive their counterparts for λ^*_{G}.

We have proved contextual equivalence of the rules in Figure 6 restricted to only hard dependencies. This restriction does not impact most rules in a major way, except for (dce), which requires forbidding any mention of x in the rest of the program g. To prove contextual equivalence of the full (dce), the logical relation needs to be extended to support read/write effects, and consequently soft dependencies. We leave the proofs for λ^*_{G} with hard/soft dependency as future work.

5.2 Higher-Order Program Optimizations

Our graph IR enables useful and efficient higher-order optimizations purely from local reasoning with dependencies and its support for lexical structure. Even without using costly higher-order flow and environment analyses [Might 2007; Shivers 1988], our graph-based transformations are already useful for eliminating higher-order functions. Additional fixed-point based analyses could be integrated to enable more aggressive optimizations.

Lambda Lifting. Lambda lifting [Johnsson 1985] transforms nested functions into top-level functions, where free variables captured by the function are replaced with newly added function parameters. Call-sites are also transformed to pass the extra arguments. This avoids creating closures at run time, *e.g.*,

```
// before lambda lifting:
def f(x: Int): Int =
  def g(y: Int): Int = x + y // closure
  g(42)
```

```
// after lambda lifting:
def g(y: Int, x': Int): Int = x' + y // g lifted
def f(x: Int): Int =
  g(42, x) // no closure
```

There are two possible ways to realize lambda lifting in the graph IR: (1) *post-scheduling*, *i.e.*, first convert the graph back into a tree (cf. Section 6), which explicates lexical scope, and then apply classic lambda lifting algorithms. In this way, the resulting program depends on the downstream scheduling algorithm which decides the scope structure of programs. (2) *pre-scheduling*, *i.e.*, directly transform the graph representation before scheduling a tree representation, discussed below:

We compute the set of extra parameters using graph dependencies [Danvy and Schultz 2004; Leißa et al. 2015; Morazán and Schultz 2007] and express lambda lifting by graph rewriting:

- The dependencies to scope-sensitive variables are rewired to the newly introduced parameters.
- Call-sites are transformed to introduce explicit dependencies to lexically scoped variables.

After that, we must instruct the later code motion algorithm (cf. Section 6) to lift functions without local dependencies to the top level, *i.e.*, those functions should not have a parent scope. Using a graph structure to compute the set of extra parameters also exhibits better asymptotic complexity [Morazán and Schultz 2007] compared to Johnsson's equation-based algorithm.

Similar to lambda lifting, with additional support of data structures, we can also implement closure conversion as a graph transformation.

Super-Beta Inlining. Super-β inlining [Shivers 1991] is an aggressive form of higher-order function inlining across abstraction boundaries, traditionally driven by expensive control-flow analysis (CFA) [Midtgaard 2012; Shivers 1988, 1991] and environment analysis [Might 2007]. Our graph IR does not directly answer global control-flow questions or track the dynamic environment component of closure values. However, a subset of super-β inlining can be viewed as a combination of lambda dropping [Danvy and Schultz 2000] (*i.e.*, the inverse of lambda lifting) and local function inlining. We show that by taking advantage of the sparsity and locality of graph dependencies, our graph IR can already enable super-β inlining in many cases.

Consider the tail-recursive factorial function written in continuation-passing style (adapted from Might [2007]). We invoke `fact` with a continuation that prints the final result.

```
// before super-β inlining:
def fact[T](n: Int, acc: Int, k: Int => T) =
  if (n == 0) k(acc)
  else fact(n - 1, acc * n, k)
fact(x, 1, res => println(res))
```

```
// after super-β inlining:
def fact[T](n: Int, acc: Int) =
  if (n == 0) println(acc)
  else fact(n - 1, acc * n)
fact(x, 1)
```

The highlighted call of `k` in the body (left) is a super-β inlinable call-site. In our IR, we immediately know that `fact` is called with a continuation, connected by a data-dependency edge. This continuation function as a closure shares no variables with `fact` from the environment. Moreover, all recursive calls to `fact` in its body have the same continuation argument `k`. These facts are enough to establish the safety condition to perform super-β inlining in this case. Moreover, they can be identified without relying on any fix-point analysis but inspecting the graph locally. After super-β inlining, the call-site of `k` is replaced with the continuation body. Furthermore, the argument `k` becomes useless and is therefore eliminated. To summarize, the resulting program can be obtained by first dropping the continuation function into the scope of `fact` and then inlining the function.

Beyond Graph Transformation. We have explored optimizations based on local graph rewriting or transformations, which are practical and cost-effective. However, powerful (and expensive) analyses based on fixed-point iteration can also be implemented on our IR. For instance, super-β inlining typically requires sophisticated environment and flow analyses. Implementing a global CFA on the graph IR would be more efficient compared to tree-like IRs. Instead of iteratively traversing tree-like ASTs and propagating information, we can directly update the graph and leverage its sparsity and locality for information propagation.

6 FROM GRAPHS BACK TO TREES

This section covers efficient algorithms and heuristics for code generation, transforming λ^*_{G} graphs into trees. These algorithms refine the optimizations presented in Section 5. We discuss the basic scheduling algorithm with dead code elimination (Section 6.1), a lightweight frequency estimation heuristic for code motion (Section 6.2), and a compact scheduling algorithm for instruction selection and expression inlining (Section 6.3).

6.1 Basic Scheduling Algorithm with Dead Code Elimination

The block scheduling algorithm traverses a hierarchy of graph-represented blocks based on dependencies and selects the nodes to build tree-represented blocks. Figure 7 shows the vanilla block scheduling algorithm. Following the dependencies of the final result `res`, `scheduleBlock` partitions the unscheduled nodes `scope` into two groups: (1) schedule in the current block, and (2) schedule into some inner block. This process is recursively applied when encountering lambda nodes in `traverseNode`. Scheduling decisions rely on two properties over nodes, *available* and *reachable*:

Nested Scopes. A node is *available* if all its dependent bound variables have been introduced under the current `path`. This is the set of accumulated bound variables (*e.g.*, introduced by lambdas) down to the current block. We schedule a node in the current block if it is both reachable and available; otherwise, it is deferred to some inner block. Consequently, we need to transitively compute the bound variables depended upon by nodes, which is performed by `boundDeps`. Both `path` and `scope` need to be maintained through recursive calls to properly handle blocks and nested scopes.

Dead Code Elimination. A node is *reachable* if it can be traced back from the current result node through effect or data dependencies. Only reachable nodes are scheduled, which naturally eliminates dead code (cf. the DCE rule in Figure 6). In Figure 7, *reachable* is implemented as a priority queue (`reachable`) to keep track of reachable nodes in topological order, populated with data and effect dependencies during the iteration. As an extension, we discern soft dependencies (Section 4.2) and identify data and hard dependencies as `reachableHard`. This ensures nodes that are only reachable via soft dependencies can be eliminated.

Complexity. Given the total number of nodes n and the maximal depth of nested scopes k, the worst-case asymptotic time complexity is $O(kn^2)$. This is because the algorithm traverses over the reachable nodes in order ($O(n)$ each), and repeats this process for nested scopes. In practice, the complexity is $O(kn \log n)$ due to the decreasing size of nested scopes and the limited degrees of graph nodes. For example, our symbolic-execution compiler (Section 7.2) based on the graph IR schedules 548,976 nodes within 19.3 sec, which is rather efficient.

```
/* auxiliary functions to access different sorts of dependencies of a node
   boundDeps: dependencies that are bound variables
   dataDeps, effDeps: data- and effect-dependencies
   hardDeps (⊆ effDeps): hard effect-dependencies */
val boundDeps, dataDeps, effDeps, hardDeps : Node => Set[Node]
/* obtain estimated frequencies of data-/effect-dependencies of a node: */
val depFreq: Node => Map[Node, Double]

/* traverse a single node to emit a tree node */
def traverseNode(inner: Set[Node], path: Set[Node], n: Node): TreeNode = n match
 case λf(x).r =>                                       // schedule nodes into a λ scope
   TreeNode.Scope(λf(x), scheduleBlock(inner, path ∪ {f, x}, r))
 ...
 case "$sym := $op($args)" =>                          // schedule common nodes as leaves
   TreeNode.Leaf(sym, Exp(op, args))

/* schedule a block given its final result, producing a scoping tree */
def scheduleBlock(scope: Set[Node], path: Set[Node], res: Node): List[TreeNode] =
 val reachable: PriorityQueue[Node] = {res}            // reachable nodes, topologically ordered
 val reachableHard: Set[Node] = {res}                  // reachable nodes, required by data/hard deps.
 val reachableHot: Set[Node] = {res}                   // reachable nodes, frequently executed
 val current: List[Node] = ∅                           // scheduled in current block
 val inner:   Set[Node]  = ∅                           // scheduled in inner blocks
 def available(n: Node): Boolean = boundDeps(n) ⊆ path // available: bound vars. in deps. are ready

 for n ⟵ reachable do
   if reachableHard(n) then                            // reachable via data/hard dependencies
     if reachableHot(n) ∧ available(n) then            // reachable via hot paths
       current = n :: current
       for m ⟵ (dataDeps(n) ∪ effDeps(n)) ∩ scope do   // consider deps. hot if freq > 0.5
         if depFreq(n)[m] > 0.5 then reachableHot += {m}
     else                                              // only via cold path, or hot but unavailable
       inner += {n}
       if reachableHot(n) then                         // deps. of unavailable hot nodes are still hot
         reachableHot += (dataDeps(n) ∪ effDeps(n)) ∩ scope
     reachableHard += (dataDeps(n) ∪ hardDeps(n)) ∩ scope  // reachable via data and only hard dependencies
   reachable += (dataDeps(n) ∪ effDeps(n)) ∩ scope     // reachable via data/effect dependencies

 for n ⟵ current yield traverseNode(inner, path, n)    // recursively build up the scoping tree
```

Fig. 7. The pseudocode of the basic scheduling algorithm with the extensions of dead code elimination (Section 6.1) and frequency estimation (Section 6.2). Function `scheduleBlock` decides which nodes are scheduled into the `current` block and recursively schedules `inner` scopes. To generate code for a graph `g`, we invoke `scheduleBlock(g.nodes, ∅, g.result)`.

6.2 Code Motion with Frequency Estimation

Our basic scheduling algorithm eagerly schedules nodes into their outermost block, following the equational rules (COMM) and (λ-HOIST) in Figure 6. This is a form of code motion with no extra effort and generally desirable for functions and loops. For instance, consider the following code:

```
List(1, 2, 3, 4, 5).map(x => x * factorial(N))
```

Lifting the expensive `factorial` out of the lambda is feasible since it does not depend on the bound variable `x`. However, this does not guarantee generating optimal code. Consider a conditional expression that transforms an array of complex numbers only in the then-branch,

```
if (cnd) compNums.map(f) else compNums
```

Since `compNums.map(f)` has no dependency on the condition `cnd`, it would be lifted to the outer block and always be executed, imposing unnecessary runtime overhead when the else-branch is taken.

To avoid this situation, we can statically *estimate* how frequently a node will be used at runtime and move less frequent ("cold") nodes into nested scopes. The results of functions and loops are assigned frequency 100, indicating that they can be executed multiple times (definitely "hot"). The results of conditional branches are assigned 0.5 (*i.e.*, cold), assuming each branch is taken with equal probability. All other nodes are assigned 1.0 (*i.e.*, normal). Numbers above are illustrative and context-insensitive. Alternative metrics are possible, while what we present here is beneficial to most code patterns.

The teal parts in Figure 7 highlight the changes to the basic scheduling algorithm for frequency estimation. Given a node which is to be scheduled in the current block, we use the function `depFreq` to access the frequency estimation of its dependencies. Only those dependencies with frequencies greater than 0.5 are considered hot-reachable, and thus can be included in `current`. Other dependencies are classified as cold, and a node should be deferred to inner blocks if downstream nodes depend on it only via cold dependencies. For a node which is to be scheduled in some inner block, its dependencies are considered to have the same "temperature" as the node itself, ensuring consistent code motion behavior for code with nested scopes.

Our approach is easier to implement and more efficient than sophisticated analyses, such as lazy code motion [Knoop et al. 1992a], partial redundancy elimination [Kennedy et al. 1999], and whole-program dataflow analyses. The heuristic associates a constant factor to each traversed node, keeping the scheduling algorithm's complexity unchanged.

6.3 Instruction Selection with Compact Traversal

The basic scheduling algorithm binds every intermediate expression, which is not only verbose but also suboptimal without considering the target-specific primitives. Consider the following tensor computation snippet,

```
val X = Matmul(A, B); val C = Add(C, X); C
```

The unique use of `X` in `Add` enables destination-passing style using generalized matrix multiplication `GEMM`, which updates `C` in-place by $C \longleftarrow \alpha AB + \beta C$. Thus, we can match the tree structure `Add(C, Matmul(A, B))` and generate a single operation, eliminating the intermediate multiplication `X`:

```
GEMM(A, B, C, alpha=1.0, beta=1.0); C
```

This is a form of *instruction selection* as seen in optimizing compilers. However, it is non-trivial on computation graphs where all consumers of a value need consideration. While LLVM's SelectionDAG algorithm [Lattner and Adve 2004] offers a comprehensive solution, our graph IR employs a simpler, efficient alternative: *compact traversal*. It converts graph nodes into inlined trees when possible, then selects the best primitive using tree-matching algorithms (*e.g.*, maximal munch) while respecting dependencies to preserve the semantics. Consider the following code where `inc(x)` has an effect dependency on node `y`, although `y` is used only once.

```
if (cond) { val y = !x; inc(x); println(y) }
```

Therefore, inlining `y` to `println` breaks the semantics, and our algorithm correctly avoids generating inlined code for such cases.

Compact traversal works on each `current` block determined by the basic scheduling algorithm (Figure 7). Initially, all nodes in the block are viewed as individual trees. Three steps are then taken to build them into a list of inlined trees ready for pattern matching and code emission. We document the complete pseudocode in the supplementary material [Bračevac et al. 2023].

(1) **Track Node Usage.** The core concept of compact traversal is to consider inlining only for nodes that are locally defined, used exactly once, and not used in nested scopes. We identify inlining candidate nodes by tracking local definitions and node usages in the current and inner blocks.

(2) **Compute Local Successors.** After recording node usage information, we calculate local successors of a node within the current block. Node x is a local successor of node y if they are scheduled in the same block and x depends on y. A map is used to store local successors of nodes.

(3) **Check Inlining.** Finally, the algorithm runs a backward pass for all inlinable nodes, checking if all successors are emitted after the potential inlining point. If not, inlining the current node is disabled; otherwise, it is inlined and any node it uses is checked for inlinability.

Summary. We demonstrated various optimizations achievable with simple, efficient algorithms through scheduling graphs back to nested tree representations. Additional optimizations can be further incorporated, *e.g.*, to exploit instruction-level parallelism, we could use node priority values reflecting the dependency and timing to perform *instruction scheduling*. In summary, our graph IR serves as an efficient and flexible bedrock for generating high-performance code.

7 CASE STUDIES

In this section, we demonstrate optimizations enabled by our approach and evaluate its performance empirically. We implemented our graph IR in LMS [Rompf and Odersky 2010; Rompf et al. 2013], a compiler framework embedded in Scala. LMS evaluates DSL expressions into the graph IR using normalization by evaluation [Rompf 2016]. The modified LMS allows DSL authors to annotate effects for each operation (*e.g.*, Figure 5b), while dependencies are resolved automatically.

7.1 Fusing Tensor Computations on CPU/GPU

In Section 2, we discussed the interaction between loop-fusion transformations and effect dependencies. Here, we present more sophisticated case studies applying loop fusion and kernel fusion based on graph dependencies to tensor computations on CPUs and GPUs. Consider the mean and variance in a functional tensor DSL (Figure 5a):

```
def square(d: Int)      = d * d
def mean(t: Tensor)     = Sum(Seq(t.size), { i => t(i) }) / t.size
def variance(t: Tensor) = Sum(Seq(t.size), { i => square(t(i)) }) / t.size - square(mean(t))
```

The `Tensor` and `Sum` forms are constructors for creating tensors and summations, respectively. Under the hood, these types represent the computations as loops. Say we computed the mean and variance of an affine tensor, obtained by pointwise addition of the tensors `constant` and `linear`:

```
val constant = Tensor(Seq(100), { i => 1 })
val linear   = Tensor(Seq(100), { i => 2*i })
val affine   = Tensor(Seq(100), { i => constant(i) + linear(i) })

println(mean(affine))
println(variance(affine))
```

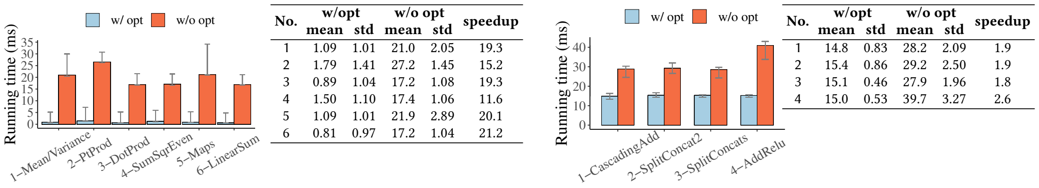
A compiler without fusion would generate six loops for this program. Utilizing the dependency information from our graph IR, we can efficiently fuse these loops. In this example, the `Tensor` and `Sum` constructors internally use mutable builder objects during construction, as shown in the definitions (Figures 5a and 5b), which greatly reduces the number of fusion rules needed for program transformations.

Dependency-Guided Fusion. Following Rompf et al. [2013], we consider two fusion types: (1) horizontal fusion, combining loops with the same range, and (2) vertical fusion, fusing data producers with consumers. To apply fusion, note that the initialization of `affine` consumes data from `constant` and `linear` at the same loop index. By applying *vertical* loop fusion, `affine` directly uses this data, eliminating the need for materializing the `constant`/`linear` tensor. The resulting loop representation for computing `affine` is:

```
forLoop(Seq(100), i => builderAdd(affBld, i, 1+2*i)) /* def. for builder affBld omitted */
```

The tensor `affine` is consumed by the downstream `println` calls taking three `Sum` loops introduced by `mean`/`variance`. Applying the same vertical fusion transformation, we can continue fusing `affine` into its three consumers (*i.e.*, summing loops) and obtain the following code (there are two loops performing the same computation since `variance` also calls `mean`):

```
forLoop(Seq(100), i => sumBuilderAdd(sumBld1, i, 1 + 2 * i))
forLoop(Seq(100), i => sumBuilderAdd(sumBld2, i, 1 + 2 * i))
forLoop(Seq(100), i => val t = 1 + 2 * i; sumBuilderAdd(sumBld3, i, t * t))
```



No.	w/opt mean	w/opt std	w/o opt mean	w/o opt std	speedup
1	1.09	1.01	21.0	2.05	19.3
2	1.79	1.41	27.2	1.45	15.2
3	0.89	1.04	17.2	1.08	19.3
4	1.50	1.10	17.4	1.06	11.6
5	1.09	1.01	21.9	2.89	20.1
6	0.81	0.97	17.2	1.04	21.2

(a) Loop fusion of tensor computations.

No.	w/opt mean	w/opt std	w/o opt mean	w/o opt std	speedup
1	14.8	0.83	28.2	2.09	1.9
2	15.4	0.86	29.2	2.50	1.9
3	15.1	0.46	27.9	1.96	1.8
4	15.0	0.53	39.7	3.27	2.6

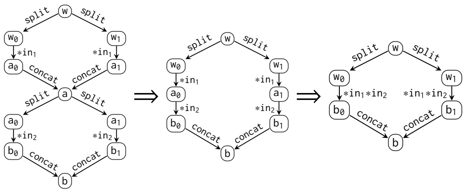
(b) CUDA kernel fusion with code motion.

Fig. 8. Empirical evaluation of (a) loop fusion and (b) kernel fusion. Bar charts show the median of running times (ms) where whiskers are max/min of samples. Tables show mean and std of the same sample set.

The identical index range of the loops suggests *horizontal fusion* is possible, but care must be taken regarding effectful operations, such as destructive mutations via `sumBuilderAdd`. By analyzing dependency information in the graph IR, we can tell that builders are separate and loops are independent, allowing fusion into a single loop. After applying additional lowering transformations, the compiler generates an optimal single-loop program.

CUDA Kernel Fusion. Kernel fusion is another case utilizing dependency information for high-level optimization. Kernels, executed by GPUs, process tensor inputs and outputs. Naive CUDA backends in ML compilers may generate excessive intermediate results in the GPU's global memory, reducing throughput and efficiency. Kernel fusion [Filipovič et al. 2015] vertically combines a sequence of kernels into one, improving performance. We extend the AIRCop compiler [Wang et al. 2019a] on top of our graph IR with kernel fusion and other optimizations (*e.g.*, DCE and CSE). Consider the following tensor program:

```
val in1 = Input(Seq(100,100), "input1.data")
val in2 = Input(Seq(100,100), "input2.data")
val w = Tensor(Seq(100,100))
val a = in1 * w
val b = in2 * a
print(b)
```



When executing a program on a two-GPU cluster, the weight tensor `w` can be split and scattered across the cluster, using model parallelism. Dependency analysis shows that the intermediate result `a`, generated by the first multiplication, is only consumed by the second multiplication. Therefore, tensor `a` can remain distributed during execution. Kernel fusion combines the two matrix multiplications, eliminating intermediate results and leading to the optimized dataflow to the right.

Kernel fusion is also used to generate low-level CUDA code for conserving memory in distributed model training, by scattering weight gradients across the cluster. The following snippet demonstrates this transformation through kernel fusion. Before `t` is accumulated, it is collected from other GPUs' transient memory using a sequence of `add` kernels:

```
recv(GPU1, w1, GPU2, w2, GPU3, w3);
cudaMalloc(t1); cudaMalloc(t2);                        recv(GPU1, w1, GPU2, w2, GPU3, w3);
add(t1, w0, w1); add(t2, w2, w3); add(t, t1, t2);  ⇒   addAccumulate(t, w0, w1, w2, w3);
accumulate(t);
```

After fusion (right), these `add` kernels and `accumulate` kernel are fused into a single one, eliminating intermediate tensors `t1` and `t2` which saves memory.

Empirical Evaluation. Figure 8a compares the performance of low-level Scala code with and without loop fusion on benchmarks, including the mean/variance example. Using arrays of 10^6 elements, each benchmark runs 20 times, reporting median/min/max in the bar chart and mean/std in the table. The large std numbers result from JVM warmup. Our fusion algorithm successfully combines multiple loops, with fused programs running up to 21x faster than those without fusion.

Figure 8b compares the execution time of CUDA kernel programs with and without fusion, using an input size of 300,000 on two NVIDIA GeForce GTX 1080 GPUs. Kernel fusion with code motion enhances tensor benchmark performance by 1.5x to 2x on average.

7.2 Symbolic-Execution Compiler

Wei et al. [2020, 2023b, 2021] introduce a symbolic-execution (SE) compiler framework built using LMS. It converts LLVM IR programs into C++ files for SE. The artifact generates purely functional, easily parallelizable code with domain-specific optimizations for LLVM's symbolic semantics. Due to purity and persistence, only data dependencies exist, simplifying the implementation of optimizations, but potentially negatively impacting single-thread execution performance.

In this case study, we refactor their artifact to generate effectful code with in-place updates, reducing intermediate structures. In-place operations are effect-annotated, enabling LMS to generate an effectful IR graph with dependencies. We then apply *effect-aware* graph rewriting to reproduce their optimizations and evaluate their effectiveness.

Effect-aware Rewriting and Optimizations. We identify interesting optimizations in the original artifact requiring effect-aware rewriting after refactoring, such as register assignment/lookup elimination and stack allocation aggregation. Local and effect-free expression simplification is also important but less relevant to this case study. We use register lookup elimination as an example.

In LLVM IR, local variables within function scope are called "virtual registers" and will be assigned only once. For example, the following code snippet is the compiled program corresponding to an LLVM program that assigns register `x` with value `v` and then later looks up this register `x`:

```
val _ = s.assign(x, v) /* write effect over s.x, where s is a symbolic state */
...                    /* other effectful operations over s */
val t = s.lookup(x)    /* read effect over s.x */
...                    /* later code uses lookup result t */
```

The compiled program operates on a symbolic state `s`. The unnecessary `lookup` can be eliminated since `x` is assigned once and its value is locally known. Rewriting the program binds `t` to `v`. In the graph representation, an effect dependency connects the `lookup` and `assign` nodes, with `lookup` inducing a read effect on `s.x` and `assign` inducing a write effect on `s.x`. The assigned value is found by locating the last write over `s.x` and performing substitution on downstream computations, *i.e.*, in the formal notations from Sections 3 and 5.1:

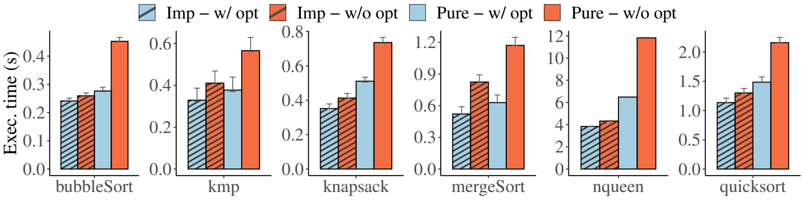
$$C_1[\textbf{let } n = s.\text{assign}(x, v) \bullet \mathsf{h}; \mathsf{s} \textbf{ in } C_2[\textbf{let } y = s.\text{lookup}(x) \bullet \{s.x \mapsto n\}; \mathsf{s} \textbf{ in } g]] \equiv$$
$$C_1[\textbf{let } n = s.\text{assign}(x, v) \bullet \mathsf{h}; \mathsf{s} \textbf{ in } C_2[g[y \rightsquigarrow \{s.x \mapsto n\}; \mathsf{s}][v/y]]]$$

After exhaustive elimination of `lookup`s, the corresponding assignments become dead code and will also be removed during code scheduling.

Experiment. We compare the refactored (Imp) and original (Pure) SE compiler with and without the optimizations. We record the sizes of generated programs and their running times, which are divided into solver time and execution time. Each task is repeated 10 times. We use a downstream C++ compiler `g++ 9.2.0` with `O3`, SMT solver `STP`, and JDK 8, running on a machine with 4 Xeon 8168 CPUs and 3TB memory.

Empirical Evaluation: Imp w/ opt vs. Imp w/o opt. Figure 9 shows the result of 6 benchmarks. We first observe that the generated C++ programs are more compact (21.49% smaller on avg.), since optimizations lead to constructing a smaller graph and program.

Symbolic execution workloads often spend most execution time on constraint solving by an external solver. Compile-time rewriting of symbolic expressions can improve solver-side performance. For example, KMP considerably benefits from eliminating unnecessary bit-vector extensions.



Benchmark			LOC	Solver Time (s)		Exec. Time (s)		
				mean	std	mean	std	speedup
Bubble Sort	Imp	w/ opt	198	21.71	0.03	**0.24**	0.003	**1.9**
		w/o opt	259	**21.42**	0.02	0.26	0.003	1.8
	Pure	w/ opt	195	21.65	0.04	0.27	0.004	1.6
		w/o opt	241	21.43	0.008	0.45	0.004	1.0
KMP	Imp	w/ opt	399	**4.11**	0.006	**0.33**	0.01	**1.7**
		w/o opt	504	8.32	0.03	0.41	0.01	1.4
	Pure	w/ opt	400	4.31	0.01	0.38	0.02	1.5
		w/o opt	486	8.37	0.01	0.57	0.02	1.0
Knapsack	Imp	w/ opt	288	**126.01**	0.06	**0.35**	0.008	**2.1**
		w/o opt	360	130.53	0.07	0.41	0.008	1.8
	Pure	w/ opt	257	134.97	2.29	0.51	0.009	1.4
		w/o opt	314	129.96	0.07	0.73	0.009	1.0
Merge Sort	Imp	w/ opt	459	47.22	0.03	**0.52**	0.02	**2.2**
		w/o opt	634	**46.93**	0.02	0.83	0.02	1.4
	Pure	w/ opt	448	47.82	0.13	0.63	0.02	1.9
		w/o opt	582	47.82	0.05	1.17	0.02	1.0
NQueen	Imp	w/ opt	564	**2.56**	0.004	**3.93**	0.01	**3.1**
		w/o opt	678	2.57	0.003	4.32	0.01	2.7
	Pure	w/ opt	563	3.04	0.02	6.48	0.02	1.8
		w/o opt	656	3.58	0.005	11.81	0.02	1.0
Quick Sort	Imp	w/ opt	237	**89.93**	0.04	**1.14**	0.02	**1.9**
		w/o opt	297	90.13	0.19	1.31	0.02	1.7
	Pure	w/ opt	229	90.82	0.06	1.49	0.02	1.5
		w/o opt	276	90.42	0.03	2.17	0.02	1.0

Fig. 9. Empirical evaluation: Graph IR optimizations on a symbolic-execution compiler. Bar charts show the execution time for each benchmark and engine configurations. Table reports the size of the generated code (LOC), solver running time, execution time, and speedups of execution times (baseline is "Pure w/o opt").

```
SELECT SUM(l_extendedprice*l_discount)
  FROM lineitem
 WHERE l_shipdate >= '1994-01-01'
   AND l_shipdate < '1995-01-01'
   AND l_discount between 0.05 and 0.07
   AND l_quantity < 24
```

(a) TPC-H Query 6.

```
class ColumnarBuffer(schema: Schema, columns: Map[String, Column], /* elided */) {
 def insert(rec: Record) =
  schema map { field  =>            // enumerate table fields
   val column = columns(field.name) // get the column for the field
   column.insert(rec(field.name))   // insert the new value
  }
}
```

(b) `ColumnarBuffer` insert implementation in LB2.

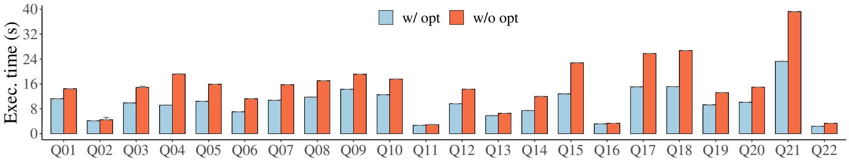
```
int main() {
 /* allocating memory for columns */
 int* l_orderkey_buf = (int *) malloc(init_size);
 int* l_partkey_buf = (int *) malloc(init_size);
 ... (10 other unused columns)
 int* l_shipdate_buf = (int *) malloc(init_size);
 double* l_discount_buf = (double *) malloc(...);
 ... (2 other used columns)

 while (/* end of source file */) {
  /* parse the data and insert to buffers */
  // parse l_orderkey (omitted)
  l_orderkey_buf[pos] = new_l_orderkey;
  // parse l_partkey (omitted)
  l_partkey_buf[pos] = new_partkey;
  ... (10 other unused columns)
  // parse l_shipdate (omitted)
  l_shipdate_buf[pos] = new_shipdate;
  // parse l_discount (omitted)
  l_discount_buf[pos] = new_discount;
  ... (other 2 used columns)

  /* resize the buffers if full */
  if (/* buffer is full */) {
    l_orderkey_buf = (int *) realloc(l_orderkey_buf, new_size);
    l_partkey_buf = (int *) realloc(l_partkey_buf, new_size);
    ... (resize other 10 unused columnar buffers)
    l_shipdate_buf = (int *) realloc(l_shipdate_buf, new_size);
    l_discount_buf = (double *) realloc(l_discount_buf, new_size);
    ... (resize other 2 used columnar buffers)
  }
 /* rest of the query */
}
```

(c) Code generated by LB2 for TPC-H Query 6.

Fig. 10. The DCE pass on our graph IR eliminates unused columns (red lines) altogether which general-purpose compilers (*e.g.*, GCC -O3) are incapable of.

Execution time (*i.e.*, total time minus solver time) is not prominent in most cases due to utilizing in-place updates and avoiding copying. Our graph-IR optimizations still yield significant improvements. On average, the optimized Imp version is 16.95% faster than the non-optimized version. These speedups demonstrate the effectiveness of graph-based domain-specific optimizations, particularly for workloads with dominant concrete execution, such as `NQueen` in Figure 9.

Empirical Evaluation: Imp vs. Pure. In Figure 9, we also assess the impact of transitioning from persistent to imperative data structures, enabling effectful in-place updates. The unoptimized imperative version is sometimes faster than the optimized pure version. When optimizations are enabled for both, the imperative version is 24.30% faster on average than the pure version, because our optimizations eliminated the creation and storage of intermediate results.



Time (s)	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
w/ opt	11.2	4.1	9.9	9.2	10.4	7.0	10.7	11.8	14.2	12.5	2.7	9.6	5.8	7.4	12.8	3.1	15.0	15.1	9.2	10.1	23.2	2.4
w/o opt	14.4	4.6	14.9	19.2	15.9	11.2	15.7	17.0	19.1	17.6	2.9	14.3	6.5	11.9	22.8	3.3	25.8	26.7	13.2	15.0	39.2	3.3
speedup	1.3	1.1	1.5	2.1	1.5	1.6	1.5	1.4	1.3	1.4	1.1	1.5	1.1	1.6	1.8	1.1	1.7	1.8	1.4	1.5	1.7	1.4

Fig. 11. Empirical evaluation of the effect of precise dependency tracking based DCE in graph IR. We run all 22 queries of the TPC-H Benchmark (Scale Factor = 10) with (and without) the DCE pass.

7.3 LB2: Relational Query Compiler

LB2 [Essertel et al. 2018; Rompf and Amin 2015; Tahboub et al. 2018] is a comprehensive SQL query compiler built with LMS that achieves competitive performance compared to highly optimized counterparts like HyPer [Neumann 2011]. One of the key optimizations enabled by our graph IR's dependency tracking is a very precise DCE during code generation that cannot be achieved with downstream general-purpose compilers (*e.g.*, GCC -O3). The precise DCE prevents redundant memory allocations, copying, etc., for attributes that are not used in a query.

In Query 6 (Figure 10a) from the TPC-H benchmark [TPC 1999], an aggregate is computed for the `lineitem` relation based on specific constraints. The `lineitem` relation has 16 attributes, but only four are used for the final result. The query plan includes data loading, filtering, and an aggregation operator. Figure 10b presents an excerpt of the `ColumnarBuffer` type, storing table data in a columnar format with separate arrays for each column. These high-level abstractions exist only in the library code and map to native C arrays in the compiled code.

The generic buffer abstraction is easy to use but lacks query-specific information, such as used columns. Without precise DCE, the generated C code for Query 6 would unnecessarily allocate, reallocate, and copy memory for 12 unused columns in `lineitem`. The downstream compiler, unaware of these high-level dependencies, cannot eliminate these unneeded operations.

Our graph IR precisely tracks dependencies, such as the creation and usage of individual columns in each relation through effect and data dependencies. Columns needed for the final output have data dependencies to the corresponding columnar arrays. Our DCE rule (Section 5.1) eliminates unreachable columnar arrays, preventing them from being scheduled for code generation, as shown in the highlighted parts of Figure 10c.

Empirical Evaluation. We evaluate the effectiveness of precise DCE on large-scale realistic queries from the TPC-H benchmark (Scale Factor 10), using the same experimental setup as Section 7.2. Figure 11 summarizes the results of running the queries with and without DCE. We observe a maximum speedup of 2.1x and an average speedup of 1.5x across all queries.

8 RELATED WORK

Graph-based IRs for Imperative Languages. Program dependence graphs (PDGs) [Ferrante et al. 1987] include explicit data and control dependencies which allow many optimizations to run in a single pass and incrementalize dataflow computations. Our work shares these qualities, *e.g.*, in the code motion and tree generation phase. Similarly, the graph-based IR by Click and Paleczny [1995] is an enhanced form of static single assignment form (SSA) [Cytron et al. 1991] that has been used in production and research compilers, such as the Java HotSpot compiler [Paleczny et al. 2001] and Truffle/Graal [Duboscq et al. 2013; Würthinger et al. 2013]. However, these works target first-order imperative languages, while our IR targets higher-order functional languages with effects. Notably our IR has a type system based on reachability and effects [Bao et al. 2021], which is novel in this area. General-purpose typed graph representations are becoming more commonplace in compilers. For instance, MLIR [Lattner et al. 2021] can be classified as an extensible DSL for compiler infrastructures.

Graph Reduction and Rewriting for λ-Calculi. Pure functional language compilers have a quite different notion of graph IR. Terms in the λ-calculus correspond to terms in combinatory logic which effectively yields simplified IRs for functional programs in point-free style, *i.e.*, devoid of variables and bindings [Turner 1979; Wadsworth 1971]. Due to referential transparency, combinatory terms can be presented as graphs with zero code duplication, and optimizations apply simultaneously to all occurrences. Graph reduction techniques later became popular for implementing non-strict functional languages [Johnsson 1984; Peyton-Jones 1987; Peyton-Jones and Salkild 1989; Turner 1979]. A closely-related notion is term-graph rewriting [Barendregt et al. 1987], and both are interderivable [Danvy and Zerny 2013; Zerny 2013]. Our work deals with impure functional languages so that referential transparency and the sharing properties of a combinator graph are lost. Moreover, our reliance on sets of variables and effects on variables is incompatible with the point-free style.

Functional Compilers and High-Performance DSLs. Elliott et al. [2003] propose a framework for compiling embedded DSLs that contains optimizations including CSE and code motion. Code motion is implemented by first converting the input program into a directed acyclic graph (DAG). After let-bindings are introduced for CSE in the DAG, expressions in loop bodies that are not dependent on the loop variable are candidates for code motion. More fine-grained constraints ensure that it moves the largest such expression that does not incur additional re-computation.

Early LMS [Rompf 2012; Rompf et al. 2011] and Delite [Sujeeth et al. 2014] systems did not allow sharing between mutable objects or nested mutable references, and rely on DSL writers to manually annotate side effects. Thus, they can only perform loop fusion in the phase of code generation. As shown in Section 7.1, this work allows us to add another abstraction layer into LMS, *i.e.*, translating a DSL into a typed imperative graph IR, instead of directly translating into low-level code. This treatment separates optimization from code generation, thus enhances the modularity of LMS by progressively lowering the level of abstraction.

Futhark [Henriksen et al. 2017] is a functional data-parallel language targeting GPUs, similar to our work, using types for local reasoning. Lift [Steuwer et al. 2017] employs a λ-calculus IL and graph IR for array computations on GPU backends. RISE and ELEVATE [Hagedorn et al. 2020], successors to Lift, feature a DSL for optimization strategies, inspired by Stratego [Visser et al. 1998]. As demonstrated in Section 7.1, our work can lower tensor computations to GPUs with kernel fusion [Wang et al. 2018, 2019b]. We could potentially support similar user-defined strategy DSLs.

Equality saturation and e-graphs [Nelson and Oppen 1980] have been adopted for compiler optimization [Tate et al. 2009; Willsey et al. 2021]. However, their scalability in impure higher-order languages is unclear. Integrating them with our graph IR is an interesting research direction.

Thorin [Leißa et al. 2015] is a graph IR for pure higher-order functional programs based on CPS that tracks data dependencies only, whereas our graph IR supports impurity/effects in direct-style functional programs, including complex usage patterns arising from mutability and higher-order functions. Thorin calculates node dependencies by a transitive reachability analysis, while our system uses reachability types to first determine transitively aliased sets as potential dependency candidates, before further narrowing these down by considering their usage effects.

Whole-program Optimization. MLton [Weeks 2006] reduces polymorphic and higher-order programs to a first-order SSA-based IR using monomorphization, inlining, and defunctionalization, enabling traditional intraprocedural optimizations but causing code duplication and longer compilation times due to being whole-program.

In contrast, our approach works directly on unmodified higher-order and potentially polymorphic program representations, enabling high-level rewriting rules (*e.g.*, map/loop fusion (Sections 2 and 7.1) and modular application to program units, resulting in benefits such as faster compilation times and late binding against libraries.

Code Motion. Classical code motion algorithms over CFGs aim to eliminate redundant computation based on expensive program analyses identifying unnecessary or loop-invariant computations [Cytron et al. 1986; Knoop et al. 1992b, 1994]. However, loop-invariant code motion is bound to the structure of a pre-existing CFG, thus cannot move code where no block exists. The lexical structure in our graph IR avoids these problems and has more degrees of freedom when moving code. Based on Click [1995]'s work, our code motion algorithms work over graph IRs and schedule instructions. In addition, we take advantage from the dependencies and effects information provided by our type system. We also consider higher-order functions as a basic ingredient of our IR and can move entire pure functional blocks, which are not considered by Click's IR and code motion algorithm. Barany and Krall [2013] extends Click's work with a spilling algorithm that performs global code motion for effectful programs in the phase of machine instruction scheduling. Our code motion algorithm is applied in the stage of code generation.

Type-and-Effect Systems. Ownership type systems [Clarke et al. 2013; Noble et al. 1998] have a long history of structuring memory resources. As a close cousin to those systems, we are the first to use reachability types [Bao et al. 2021] to extract effect dependencies in an IR and use them to perform optimizations. Benton et al. [2009, 2006] study the use of effect information to identify valid transformations over higher-order effectful programs. However, computations with write effects cannot be eliminated even if they are not observed by the rest of the program. We go a step further, and use effects and aliases to derive precise dependencies that track last uses. Therefore, we are able to soundly remove certain non-observable effectful computations.

9 CONCLUSION

This work bridges a long-standing gap in graph-based compiler optimization, enabling affordable global optimizations for impure higher-order programs using reachability types, effects, and high-level IR graphs with lexical structure. We have formalized and proved type-and-dependency safety, and our experimental evaluation demonstrates competitive performance on a variety of benchmarks.

ACKNOWLEDGMENTS

A large number of people have contributed to the design of the LMS graph IR over the years, and in particular to the system of effect dependencies: Nada Amin, Thaïs Baudon, Kevin J. Brown, James M. Decker, Grégory Essertel, Georg Ofenbeck, Alen Stojanov, Arvind K. Sujeeth, Fei Wang, and Yushuo Xiao. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Meta, Google, Microsoft, and VMware.

REFERENCES

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 139:1–139:32.

Yuyan Bao, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2023. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, and Equational Theory. arXiv:2309.05885 [cs.PL]

Gergö Barany and Andreas Krall. 2013. Optimal and Heuristic Global Code Motion for Minimal Spilling. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7791)*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer, 21–40.

Hendrik Pieter Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. 1987. Term Graph Rewriting. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 259)*, J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven (Eds.). Springer, 141–158.

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based program transformations: higher-order store. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, António Porto and Francisco Javier López-Fraguas (Eds.). ACM, 301–312.

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4279)*, Naoki Kobayashi (Ed.). Springer, 114–130.

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages (Technical Report). arXiv:2309.08118 [cs.PL]

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58.

Cliff Click. 1995. Global Code Motion / Global Value Numbering. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, David W. Wall (Ed.). ACM, 246–257.

Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, Michael D. Ernst (Ed.). ACM, 35–49.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490.

Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. 1986. Code Motion of Control Structures in High-Level Languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 70–85.

Olivier Danvy and Ulrik Pagh Schultz. 2000. Lambda-dropping: transforming recursive equations into programs with block structure. *Theor. Comput. Sci.* 248, 1-2 (2000), 243–287.

Olivier Danvy and Ulrik Pagh Schultz. 2004. Lambda-Lifting in Quadratic Time. *J. Funct. Log. Program.* 2004 (2004).

Olivier Danvy and Ian Zerny. 2013. Three syntactic theories for combinatory graph reduction. *ACM Trans. Comput. Log.* 14, 4 (2013), 29:1–29:27.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *POPL*. ACM, 60–72.

Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. 2003. Compiling embedded languages. *J. Funct. Program.* 13, 3 (2003), 455–481.

Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. USENIX Association, 799–815.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* (2015), 3934–3957.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247.

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 4 (April 2021), 79 pages.

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29.

John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *POPL*. ACM Press, 458–471.

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 556–571.

David Van Horn and Harry G. Mairson. 2008. Deciding *k*CFA is complete for EXPTIME. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 275–282.

Thomas Johnsson. 1984. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, Mary S. Van Deusen and Susan L. Graham (Eds.). ACM, 58–69.

Thomas Johnsson. 1985. Lambda Lifting: Treansforming Programs to Recursive Equations. In *FPCA (Lecture Notes in Computer Science, Vol. 201)*. Springer, 190–203.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152.

Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. 1999. Partial Redundancy Elimination in SSA Form. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 627–676.

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992a. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) *(PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 224–234.

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992b. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, Stuart I. Feldman and Richard L. Wexelblat (Eds.). ACM, 224–234.

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Optimal Code Motion: Theory and Practice. *ACM Trans. Program. Lang. Syst.* 16, 4 (1994), 1117–1155.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. IEEE, 2–14.

Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *CGO*. IEEE Computer Society, 202–212.

Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 103–104.

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3 (2012), 10:1–10:33.

Matthew Might. 2007. *Environment Analysis of Higher-Order Languages.* Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA, USA.

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the *k*-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI*. ACM, 305–315.

Marco T. Morazán and Ulrik Pagh Schultz. 2007. Optimal Lambda Lifting in Quadratic Time. In *IFL (Lecture Notes in Computer Science, Vol. 5083)*. Springer, 37–56.

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (1980), 356–364.

Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1445)*, Eric Jul (Ed.). Springer, 158–185.

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *POPL*. ACM Press, 54–67.

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999), 35–55.

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *POPL*. ACM, 41–53.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot$^{\text{tm}}$ Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, Vol. 1. 1–12.

Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 124:1–124:28.

Simon L. Peyton-Jones. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall.

Simon L. Peyton-Jones and Jon Salkild. 1989. The Spineless Tagless G-Machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 184–201.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 55–74.

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: a new IR for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL 2018*, Justin Gottschlich and Alvin Cheung (Eds.). ACM, 58–68.

Tiark Rompf. 2012. *Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming.* Ph.D. Dissertation. EPFL.

Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 318–335.

Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *ICFP.* ACM, 2–9.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 497–510.

Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-Blocks for Performance Oriented DSLs. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011 (EPTCS, Vol. 66)*, Olivier Danvy and Chung-chieh Shan (Eds.). 93–117.

Amir Shaikhha, Andrew W. Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.* 3, ICFP (2019), 97:1–97:30.

Olin Shivers. 1988. Control-Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 164–174.

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* Ph.D. Dissertation. Carnegie Mellon University, PA, USA.

Olin Shivers. 2004. Higher-Order Control-Flow Analysis in Retrospect: Lessons Learned, Lessons Abandoned. In *Best of PLDI.* ACM, 257–269.

Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, David R. Kaeli and Tipp Moseley (Eds.). ACM, 165.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 74–85.

Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s (2014), 134:1–134:25.

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference.* ACM, 307–322.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: A new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 264–276.

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. https://iris-project.org/pdfs/2022-submitted-logical-type-soundness.pdf.

Ben L. Titzer. 2015. *Digging into the TurboFan JIT.* Retrieved April 14, 2022 from http://web.archive.org/web/20220414155345/https://v8.dev/blog/turbofan-jit

TPC. 1999. *TPC Benchmark H.* Retrieved April 14, 2022 from http://web.archive.org/web/20220407033028/https://www.tpc.org/tpch/

D. A. Turner. 1979. A new implementation technique for applicative languages. *Software: Practice and Experience* 9, 1 (1979), 31–49.

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 13–26.

Christopher Peter Wadsworth. 1971. *Semantics and Pragmatics of the Lambda-Calculus.* Ph.D. Dissertation. University of Oxford.

Fei Wang, Guoyang Chen, Weifeng Zhang, and Tiark Rompf. 2019a. Parallel Training via Computation Graph Transformation. In *2019 IEEE International Conference on Big Data (Big Data).* 3430–3439.

Fei Wang, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2018. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 10201–10212.

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019b. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* 3, ICFP (2019), 96:1–96:31.

Stephen Weeks. 2006. Whole-program compilation in MLton. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, Andrew Kennedy and François Pottier (Eds.). ACM, 1.

Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 164:1–164:33.

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2023a. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. arXiv:2307.13844 [cs.PL]

Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2023b. Compiling Parallel Symbolic Execution with Continuations. In *ICSE.* IEEE, 1316–1328.

Guannan Wei, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2021. LLSC: a parallel symbolic execution compiler for LLVM IR. In *ESEC/SIGSOFT FSE.* ACM, 1495–1499.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 187–204.

Ian Zerny. 2013. On graph rewriting, reduction, and evaluation in the presence of cycles. *High. Order Symb. Comput.* 26, 1-4 (2013), 63–84.

Received 2023-04-14; accepted 2023-08-27