Verifying Peephole Rewriting In SSA Compiler IRs

Siddharth Bhat
Cambridge University, United Kingdom

Alex Keizer
Cambridge University, United Kingdom

Chris Hughes
University of Edinburgh, United Kingdom

Andrés Goens
University of Amsterdam, Netherlands

Tobias Grosser
Cambridge University, United Kingdom

Abstract

There is an increasing need for domain-specific reasoning in modern compilers. This has fueled the use of tailored intermediate representations (IRs) based on static single assignment (SSA), like in the MLIR compiler framework. Interactive theorem provers (ITPs) provide strong guarantees for the end-to-end verification of compilers (e.g., CompCert). However, modern compilers and their IRs evolve at a rate that makes proof engineering alongside them prohibitively expensive. Nevertheless, well-scoped push-button automated verification tools such as the Alive peephole verifier for LLVM-IR gained recognition in domains where SMT solvers offer efficient (semi) decision procedures. In this paper, we aim to combine the convenience of automation with the versatility of ITPs for verifying peephole rewrites across domain-specific IRs. We formalize a core calculus for SSA-based IRs that is generic over the IR and covers so-called regions (nested scoping used by many domain-specific IRs in the MLIR ecosystem). Our mechanization in the Lean proof assistant provides a user-friendly frontend for translating MLIR syntax into our calculus. We provide scaffolding for defining and verifying peephole rewrites, offering tactics to eliminate the abstraction overhead of our SSA calculus. We prove correctness theorems about peephole rewriting, as well as two classical program transformations. To evaluate our framework, we consider three use cases from the MLIR ecosystem that cover different levels of abstractions: (1) bitvector rewrites from LLVM, (2) structured control flow, and (3) fully homomorphic encryption. We envision that our mechanization provides a foundation for formally verified rewrites on new domain-specific IRs.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Semantics; Computing methodologies → Theorem proving algorithms; Theory of computation → Rewrite systems

Keywords and phrases compilers, semantics, mechanization, MLIR, SSA, regions, peephole rewrites

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.23

Supplementary Material *Software*: `https://zenodo.org/records/11519739`

Funding This project has received funding from the European Union's Horizon EUROPE research and innovation program under grant agreement no. 101070374 (CONVOLVE).

Acknowledgements We thank Sébastien Michelland and Sebastian Ullrich for their early help in this project and feedback, as well as Anton Lorenzen for his helpful feedback.

1 Introduction

Static single assignment (SSA) [30] is the workhorse of modern compilers such as LLVM [16]. A key optimization that is enabled by SSA is to syntactically match a program pattern, and replace the matched pattern with an optimized, semantically-equivalent program fragment. Despite their simplicity, these local "peephole optimizations" [22] remain important in compiler development. A quick overview of the program transformation libraries of LLVM shows that more than 10% of its code[1] is dedicated to LLVM's peephole optimizer InstCombine, which is beyond the size of the LLVM loop optimizer. Despite the large size and scope of the LLVM project, Alive [20] is regularly referenced in LLVM commits. This is evidence that SMT-based, low-effort tooling for peephole rewrites can enable the use of verification in day-to-day compiler development.

Peephole rewriting has been formalized in its simpler, classical form of straight-line assembly code [24]. To our knowledge, peephole rewriting along def-use chains [1] has not yet been formalized. As an example, consider the rewrite $(y = x + 1; z = y - 1) \mapsto (z = x)$. This pattern does *not* match the program $(y = x + 1; \mathbf{p} = \mathbf{y}; z = y - 1)$ in straight-line rewriting, due to the interleaved instruction $p = y$. On the other hand, by concentrating on the dataflow, we rewrite any subprogram of the form $(y = x + 1;\ \bigcirc\ ; z = y - 1)$ to $(y = x + 1;\ \bigcirc\ ; z = x)$, regardless of what fills the hole $\bigcirc$. This is known as rewriting on the "def-use" chain, where the pattern matching is extended to semantic subexpressions in the program. Observe that the fact that addition and subtraction are pure, and that SSA does not allow mutating the value of y is critical for the correctness of this optimization. Domain-specific peephole rewrites within the MLIR compiler framework [17] rely on purity and referential transparency to enable this class of optimizations.

MLIR is a compiler framework for multi-level, domain-specific compiler IRs. It is widely used in the context of machine learning [34], quantum computing [28], and even as an alternative compiler for Lean [6], among other domains. MLIR lowers the cost of instantiating domain-specific IRs and encourages local transformations that exploit the value semantics (i.e., referential transparency) of specialized high-level IRs over global reasoning at a lower abstraction level. MLIR also introduces the concept of regions, which can model control flow and other structured IR operations as nested IRs that replace complex unstructured control. Existing formalizations of SSA do not cover domain-specific SSA-based IRs or regions.

In this paper, we propose a framework that is aimed at prototyping and verifying peephole optimizations for domain-specific SSA-based IRs. We formalize a core calculus for SSA-based IRs and mechanize it in the Lean [8] proof assistant to enable verification of peephole rewriting over SSA IRs based on value semantics with regions. Our framework is deliberately built to be interoperable with MLIR. This aims to streamline the verification of peephole rewrites for MLIR. Concretely, we contribute:

- A formalization of SSA with regions parametrized over a user-defined IR X and its mechanization in our framework[2] `LeanMLIR`(X) that exploits denotational-style value semantics for optimizing along the SSA use-def chain of an MLIR-style IR (Sections 2, 3)
- Evidence that our formalization of SSA allows for effective meta-theoretic reasoning:
  - A verified peephole rewriter, for which we prove that lifting a peephole rewrite to a rewrite on the entire program preserves semantics (Section 4.1)
  - Two verified implementations of generic SSA-based optimizations: dead code elimination and common subexpression elimination (Section 4.2)
  - Proof automation for eliminating the abstraction overhead of our SSA calculus and exposing clean mathematical proof obligations for each rewrite (Section 4.3)
- An extension of our pure optimizations in a context with side effects (Section 5)

[1] Non-blank and non-comment lines of `.cpp` files in `llvm/lib/Transforms` on commit `f4f1cf6c3`.
[2] Our framework is open-source and available at `https://github.com/opencompl/ssa`.

```
inductive Ty
| r
| nat
```

```
inductive Op
| arith_const (x : Nat) -- with compile-time data `x`
| monomial -- build equivalence class of monomial
| add -- add op.
```

(a) User definitions for `QuotRing` in our framework. `Op` has three constructors, `add`, `monomial` and `(const x)`, for `x` an element of ℕ, matching the three operations of the IR. `Ty` has two constructors, `r` and `nat`.

```
instance : OpSignature Op Ty where signature
  | .arith_const _ => { sig := [], outTy := .nat } -- takes no args, returns an `r`.
  | .add      => { sig := [.r, .r], outTy := .r } -- takes two `r`s, returns an `r`.
  | .monomial => { sig := [.nat, .nat], outTy := .r } -- takes two`.nat`s
```

(b) User-defined signatures of each `QuotRing` operation.

```
noncomputable def generator : (ZMod q)[X] := X^(2^n) + 1
abbrev R := (ZMod q)[X] / (span {generator q n})

instance : TyDenote Ty where
  toType
  | .r => R -- the denotation of `r` is an element of the ring `R`
  | .nat => Nat

instance : OpDenote Op Ty where
  denote
  | .arith_const (x : Nat), _, _ => x -- Denotation of `(arith_const x)` is  `x`
  | .add, [(x : R), (y : R)]ₕ, _ => x + y
  | .monomial, [(c : Nat), (i : Nat)]ₕ, _ =>
      Quotient.mk (span {generator q n}) (monomial i c)
```

(c) User-defined semantics of `QuotRing`. The `instance` syntax is used to define a typeclass instance, by specifying the corresponding members, which in this case are the denotation functions. The `noncomputable` annotation in Lean tells the compiler not to generate executable code for this function, since `mathlib` uses a noncomputable definition for quotients of polynomial rings. Note that our framework ensures that values are well-typed according to `OpSignature` and `TyDenote`.

Figure 1 User definitions for `QuotRing`, which declares the operations and types of the IR, the type signatures of the operations, and the denotations of the types and operations into Lean types.

- Syntax, semantics, and local rewrites for three MLIR-based IRs: (1) arithmetic over bitvectors, (2) structured control flow, and (3) fully homomorphic encryption (Section 6)

2 Motivation: Verfying Optimizations for High-Level IRs

Effective domain-specific optimizations are almost impossible when reasoning on traditional LLVM-style compiler IRs. These offer a "universal" low-level abstraction, originally designed to represent C-style imperative code. Such LLVM-style IRs are built around the concepts of load/store/arithmetic/branching, which is ideal when optimizing at the level of scalar arithmetic, instruction scheduling, and certain kinds of loop optimizations. However, this level of abstraction is unsuitable for reasoning about high-level mathematical abstractions.

Consider a compiler for Fully Homomorphic Encryption (FHE) [9], a cryptographic technique that uses algebraic structures to allow an untrusted third party to do computation on encrypted data. In such a compiler, we might have a rewrite like $(a + X^{2^n} + 1 \mapsto a)$, which is a simple identity on the corresponding quotient ring.[3] Expressed in LLVM, the computation of this simple operation consists of multiple basic blocks forming a loop, each containing memory loads, pointer arithmetic, scalar operations, and branches. As a result, the algebraic structure is completely lost and exploiting simple algebraic identities turns into a heroic effort of reasoning about side effects and stateful program behavior. State-of-the-art compilers for FHE consequently use domain-specific IRs (often expressed with MLIR [35, 26]) when generating optimized code for FHE, where algebraic optimizations can take place at an FHE-specific IR that has value-semantics (e.g., is referentially transparent) and is overall closer to the mathematical structure of the problem.

2.1 Defining `LeanMLIR`(QuotRing): Syntax and Semantics

As an example, we model an IR aimed at FHE that manipulates objects in the algebraic structure $R \equiv (\mathbb{Z}/q\mathbb{Z})[X]/(X^{2^n}+1)$. To model it, we instantiate an IR `LeanMLIR`(QuotRing) in our framework. It has three simple operations: `arith_const` and `monomial`, to construct values in R, and `add` to add two values of R. To define the syntax and semantics of `LeanMLIR`(QuotRing), we first declare the types and operations in the IR (Figure 1a). `QuotRing` has two types: `r`, which represents the ring R, and `nat` for naturals. Terms in `Op` represent the operations `arith_const`, `monomial` and `add`, and associated compile-time data. We then define the operation signatures by giving an instance of the `OpSignature` typeclass, which is offered by our framework to instantiate custom IRs (Figure 1b). That is, for each operation we specify: (1) the arity and types of arguments (`sig`), and (2) the type of the return value (`outTy`). The operation `arith_const` takes no arguments and returns a `nat`, `monomial` and `add` take two `nat`/`r`-valued arguments respectively, and both return an `r`.

The type denotation is also simple to express with the `TyDenote` typeclass (Figure 1c). `Ty` thus represents the embedded type in the IR and has only two inhabitants `r` and `nat`, whose denotation are `R` and `Nat`, the Lean (host) type that represents the mathematical objects R and $\mathbb{N}$ respectively. The denotation of operations is a Lean function from the denotation of the input types (as recorded in the signature of that operation), to the denotation of the output type. Concretely, a `(arith_const n)` operation takes no arguments, so its denotation is an `nat`, while `add` takes two `r` arguments, so its denotation is a function from the product[4] of its arguments to its output, i.e., `R × R → R`. The same is true for `monomial` for $\mathbb{Z} \times \mathbb{Z} \to R$. We define the denotation of `(arith_const n)` to evaluate to `n`, `add(x, y)` to evaluate to `(x + y)` and `monomial(a, i)` to `Quotient.mk (span generator p q (monomial a i))`, the equivalence class of aX^i. We also require a few lines of specific code to translate the MLIR abstract syntax tree (AST) to `Ty` (e.g., mapping `index` into `nat` or `R` to `r`) and `Op`, not shown here (details in Section 3.2). Together, these definitions instantiate `LeanMLIR`(QuotRing). The `QuotRing` IR does not use regions. We will see examples of regions in (Section 6.2).

2.2 Defining and Executing Peephole Rewrites for `QuotRing`

We now verify the peephole rewrite $(a + X^{2^n} + 1 \mapsto a)$, where a is a variable and X^{2^n} is a constant in the ring. In $(\mathbb{Z}/q\mathbb{Z})[X]/(X^{2^n}+1)$ this rewrite is simple to prove and, unsurprisingly, our custom `LeanMLIR`(QuotRing) IR enables us to rewrite at exactly this level. Any given peephole rewrite (of which Figure 2 is an example) consists of a context Γ of free variables in the search pattern of the peephole rewrite. The search pattern is called `lhs`,

[3] We will discuss the underlying mathematical structure in more detail in Section 6.3
[4] The mechanization uses a heterogeneous vector type `HVector`, which is coerced into the product type.

```
def a_plus_generator_eq_a : PeepholeRewrite Op [.r] .r := {
  lhs /- a + X^(2^n) + 1 -/ := [quotring_com q, n| {
        ^bb0(%a : !R):
          %one_int = arith.const 1 : i16
          %two_to_the_n = arith.const ${2**n} : index
          %x2n = poly.monomial %one_int, %two_to_the_n : (i16, index) -> !R
          %oner = poly.const 1 : !R
          %p = poly.add %x2n, %oner : !R
          %v1 = poly.add %a, %p : !R
          return %v1 : !R
  }],
  rhs /- a -/ := [quotring_com q, n| {
    ^bb0(%a : !R):
      return %a : !R
  }],
  correct := by
    funext Γv; simp_peephole [Nat.cast_one, Int.cast_one] at Γv (1)
    /- ⊢ a + ((Quotient.mk (span {f q n})) ((monomial (2**n)) 1) + 1) = a -/
    ... /- simple proof with only definitions and theorems from Mathlib -/
}
```

Figure 2 A peephole rewrite in `LeanMLIR`(QuotRing) asserts the semantic equivalence of two SSA programs given in MLIR syntax. Our proof automation through `simp_peephole` eliminates the framework overhead, such that closing a clean mathematical goal suffices to prove correctness.

and the replacement is `rhs`. The user has a proof obligation that the denotations of the left and right-hand sides are equal, which is given by the field `correct` of the peephole rewrite.

We declare our desired peephole rewrite (in Figure 2) by defining `a_plus_generator_eq_a`. Its type is `(PeepholeRewrite Op [.r] r)`, where the `Op` specifies the IR the rewrite belongs to and `[.r]` is the list of types of free variables in the program. For $(a + X^{2^n} + 1 \mapsto a)$, this is $(a : \texttt{r})$. The final instruction we are matching yields a value of type `r`. The `lhs` is the program fragment we want to match on, with the free variable `%a` interpreted as being allowed to match any variable of type `r`. Observe that the type encapsulates exactly what is necessary for a well-typed match: the types of free variables (`r`) and the type of the instruction whose return value we are replacing (also `r` in this case). The rewritten program is the `rhs` field.

Both the left- and right-hand sides of the rewrite are written in MLIR syntax. (We use MLIR's concise IR-specific syntax for readability here; our parser currently implements the slightly-more-verbose generic MLIR syntax). Note that we also include a custom quasiquotation `${2**n}`, to inline the symbolic (universally quantified) value n, even though the IR would require 2^n to be a concrete constant. Using MLIR syntax matches the LLVM community's use of automation tooling, such as Alive: copy a code snippet and get a response. Our goal is to make the use of an interactive theorem prover part of the day-to-day workflow of compiler engineers. To enable this workflow, we implement a full MLIR syntax parser, along with facilities to convert from the generic MLIR abstract syntax tree (AST) into our framework type, such that we can use MLIR syntax in Lean.

To prove the correctness of `a_plus_generator_eq_a`, we use the `simp_peephole` (1) tactic from our framework, which removes any overhead of our SSA encoding. We are left with: `⊢ a + ((Quotient.mk (span f q n)) ((monomial (2**n)) 1) + 1) = a`, a proof obligation in the underlying algebraic structure that, thanks to Lean's `mathlib`, can be closed with a few (elided) lines of algebraic reasoning.

2.3 Executing Peephole Rewrites

Given a peephole rewrite `rw` and a source program `s`, we provide `rewritePeephole` to replace the pattern `rw.lhs` in the source program `s`. If the matching succeeds, we insert the target program `rw.rhs` (with appropriate substitutions) and replace all references to the original let-binding with a reference to the newly inserted `let`. Note that the matching is based on the def-use chain. Thus, a pattern need not be *syntactically* sequential in the program `s`. As long as the pattern `rw.lhs` can be found as *subprogram* of `s`, the subprogram will be rewritten. This makes our peephole rewriter an SSA peephole rewriter, which distinguishes it from a straight-line peephole rewriter that only matches a linear sequence of instructions.

Thanks to our intrinsically well-typed encoding, we know that the result of the rewriter is always a well-typed program, under the same context and resulting in the same type as the original program. Furthermore, the framework extends the local proof of semantic equivalence to a global proof, showing that the rewriter is semantics preserving:

```
/- The denotation of the rewritten program is equal to the source program. -/
theorem denote_rewritePeephole (fuel : ℕ) (pr : PeepholeRewrite Op Γ t)
  (target : Com Op Γ₂ t₂) : (rewritePeephole fuel pr target).denote = target.denote
```

These typeclass definitions are all we need to define the `QuotRing` IR. Our framework takes care of building the intrinsically well-typed IR for `QuotRing` from this, and gives us a verified peephole rewriter, with other optimizations like CSE and DCE. We will now delve into the details of the framework and see how it achieves this.

3 `LeanMLIR`(X): A Framework for Intrinsically Well-Typed SSA

In this section, we describe the core design of the framework: the encoding of programs and their semantics in `LeanMLIR`(X) (Figure 3a). We review some dependently-typed tooling we use to define our IR. **Contexts:** Our encoding is intrinsically well-typed (i.e., each inhabitant of `Expr` or `Com` described below is, by construction, well typed). Thus, we need a *context* to track the types of variables that are allowed to occur (`Ctxt Ty`). A context is a list of types, where for example `[int, int, bool]` means that there are two variables of the (user-defined) type `int` and one variable of type `bool` we may refer to. **Variables:** The type (`Var` Γ α) encodes variables of type α in context Γ. We use De Bruijn indices [29] in the standard way, but, additionally, a variable with index i also carries a proof witness that the i-th entry of context Γ is the type α. **Heterogeneous Vectors:** To define an argument signature (`OpSignature.sig`), say, `[int, int, bool]`, we need an expression with this operation to store two variables of type `int` and one of type `bool`. We want to statically ensure that the types of these variables are correct, so we store them in a heterogeneous vector. A vector of type `HVector f` $[\alpha_1, \ldots, \alpha_n]$ is equivalent to a tuple (`f` α_1 × ... × `f` α_n).

3.1 Semantics of `LeanMLIR`(X)

The core types for programs are `Expr` and `Com`, shown in Figure 3a. The type (`Expr` Γ α) describes individual MLIR operations; we think of it as a function from values in the context Γ—also called a *valuation* for that context—to a value in the denotation of type α. Commands (`Com` Γ α) has a similar interpretation but represents sequences of operations. Each command binds a new value in the current context (the `var` constructor) until the sequence returns the value of one such variable `v` (the `ret` constructor). Thus, this encoding of SSA exploits the similarity to the ANF [2] and CPS [14] encodings. The semantics given

```
inductive Expr [OpSignature Op Ty] : Ctxt Ty → Ty → Type where
| mk (op : Op) -- op (arg1, arg2, ..., argn) : outTy op
  (args : HVector (Var Γ) (OpSignature.sig op)) : Expr Γ (OpSignature.outTy op)

inductive Com [OpSignature Op Ty] : Ctxt Ty → Ty → Type where
| ret (v : Var Γ α) : Com Γ α -- return v
| var (e : Expr Γ α) (body : Com (Γ.snoc α) β) : Com Γ β -- let v : α := e in body
```

(a) Core syntax of `LeanMLIR`(X), polymorphic over `Op`. The arguments in square brackets are assumed typeclass instances. `Type` is the base universe of Lean types.

```
variable [TyDenote Ty] [OpDenote Op Ty] [DecidableEq Ty]

def Expr.denote : {ty : Ty} → (e : Expr Op Γ ty) → (Γv : Valuation Γ) → toType ty
| _, ⟨op, args⟩, Γv => OpDenote.denote op (args.map (fun _ v => Γv v))

def Com.denote : Com Op Γ ty → (Γv : Valuation Γ) → (toType ty)
| .ret e, Γv => Γv e
| .var e body, Γv => body.denote (Γv.snoc ( e.denote Γv))
```

(b) Denotation of `Expr` and `Com` in `LeanMLIR`(X), which extends the user's `OpDenote` to entire programs. Intrinsic well-typing of `Com` makes its denotation a well-typed function from the context valuation to the return type. The angled brackets are used to pattern match on a structure constructor anonymously.

Figure 3 Definitions in `LeanMLIR`(X) for `Expr` and `Com`, and their associated denotations.

by the user in `OpDenote` are extended to semantics for `Expr` and `Com` (Figure 3b) by the framework. An `Expr` evaluates its arguments by looking up their value in the valuation and then invokes the user-defined `OpDenote.denote` to evaluate the semantics of the `op`.

3.2 Writing `LeanMLIR`(X) Programs Using MLIR Syntax

An important goal for our framework is to provide easy access to formalization for the MLIR community. Toward this goal, we have a deep embedding of MLIR's AST and a corresponding parser. This is developed using Lean's syntax extensions [33]. We augment this with a generic framework to build `Expr` and `Com` terms from a raw MLIR AST. This framework allows the user to pattern-match on the MLIR AST to build intrinsically well-typed terms, as well as to throw errors on syntactically correct, but malformed MLIR input. These are used by our framework to automatically convert MLIR syntax into our SSA encoding, along with the ability to provide precise error messages in cases of translation failure. This enables us to write all our examples in MLIR syntax, as demonstrated throughout the paper.

More concretely, we have an embedded domain-specific language (EDSL), which declares the MLIR grammar as a Lean syntax extension. As part of this work, we have found several inconsistencies with the MLIR language reference and contributed patches upstream to update them.[5] Overall, this gives users the ability to write idiomatic MLIR code into our framework and receive an MLIR AST. Moreover, as we will showcase in the examples, our EDSL is idomatically embedded into Lean, which allows us to quasiquote Lean terms. This will come in handy to write programs that are generic over constants, such as parameterizing a program by 2^n for any choice of n. We build our intrinsically well-typed data structures from this MLIR AST by writing custom elaborators.

[5] reviews.llvm.org/{D122979, D122978, D122977, D119950, D117668}

```
structure OpSignature (Ty : Type) where /- (1) Extending signature. -/
  regSig : List (Ctxt Ty × Ty)
  ...

class OpDenote [TyDenote Ty] [OpSignature Op Ty] where /- (2) Extending denotation. -/
  denote : (op : Op) → (args : HVector toType (OpSignature.sig op)) →
    (regArgs : HVector (fun (ctx, t) => Valuation ctx → toType t) (OpSignature.regSig op)) →
    (toType (OpSignature.outTy op))

inductive Expr : (Γ : Ctxt Ty) → (ty : Ty) → Type where
  | mk (op : Op)
    ...
    (regArgs : HVector (fun (ctx, ty) => Com ctx ty) (OpSignature.regSig op)) :
    Expr Γ ty

mutual /- (3) extending expression denotation to recursively invoke regions. -/
def Expr.denote : {ty : Ty} → (e : Expr Op Γ ty) → (Γv : Γ.Valuation) → (toType ty)
  | _, ⟨op, args, regArgs⟩, Γv =>
    OpDenote.denote op (args.map (fun ty v => Γv v)) regArgs.denote
  ...
end
```

Figure 4 Extending `LeanMLIR`(X) with regions. New fields are `in green`. In `OpDenote`, one can now access the sub-computation represented by the region when defining the semantics of `Op`.

3.3 Modelling Control Flow in `LeanMLIR`(X) With Regions

So far, our definition of `Com` only allows straight-line programs. To be able to model control flow, we add regions to our IR. Regions add the syntactic ability to nest IR definitions, thereby allowing syntactic encoding of concepts such as structured control flow. This is in contrast with the approach of having a sea of basic blocks in a control-flow graph (CFG) that are connected by branch instructions. More specifically, structured control flow with regions allows modeling reducible control flow [1]. General CFGs allow us to represent more complex, irreducible control flow, which makes them harder to reason about. Consequently, compiler frameworks such as MLIR have moved toward directly representing structured control flow, and we follow their approach. Notably, region arguments replace phi nodes in MLIR.

Intuitively, regions allow an `Op` to receive `Com`s as arguments, and choose to execute these `Com` arguments zero, one, or multiple times. This allows us to model if conditions (by executing the regions zero or once), loops (by executing the region n times), and complex operations such as tensor contractions and convolutions by executing the region on the elements of the tensor [34]. We implement this by extending `Expr` with a new field representing region arguments (Figure 4). We also extend `OpSignature` with an extra argument for the input types and output types of the region. In parallel, we add the denotation of regions as an argument, extending `OpDenote`. Similarly, we extend the denotation of `Expr` to compute the denotation of the region `Com`s in the `Expr`, before handing off to `OpDenote`.

This extension to our core calculus gives us the ability to model structured nesting of programs. This is used pervasively in MLIR, to represent `if` conditions, `for` loops, and higher-level looping patterns such as multidimensional strided array accesses over multidimensional arrays (tensors). We show how to model control flow in Section 6.2.

4 Reasoning About `LeanMLIR`(X)

The correctness of peephole rewriting is a key aspect of the metatheory of `LeanMLIR`(X). We begin by sketching the mechanized proof of correctness of peephole rewriting. We then discuss how the infrastructure built for this proof is reused to prove two other SSA optimizations:

common subpression elimination (CSE) and dead code elimination (DCE). Finally, we discuss our proof automation, which manipulates the IR encoding at elaboration time to eliminate all references to the framework and provide a clean goal to the proof engineer.

4.1 Verified SSA Rewriting With `rewritePeephole`

We now provide a sketch of the mechanized correctness proof of `rewritePeephole`. The key idea is that to apply a rewrite at location i, we open up the `Com` at location i in terms of a zipper [11]. This zipping and rewriting at a location i is implemented by `rewritePeepholeAt`. The zipper comprises of `Lets` to the left-hand side of i, and `Com` to the right: `let` x_2 `=` x_1`; (let` x_3 `=` x_2`; (let` x_4 `=` x_3`; (return` x_3`))): Com` $[x_1]$ α =

`((let` $x_2 = x_1$`); let` $x_3 = x_2$`); : Lets` $[x_1]$ $[x_1, x_2, x_3]$

`(let` x_4 `=` x_3`; (return` x_3`)) : Com` $[x_1, x_2, x_3]$ α

The use of a zipper enables us to easily traverse the sequence of let-bindings during transformation and exposes the current `let` binding being analyzed. This exposing is performed by `Lets`, which unzips a `Com` such that the outermost binding of a `Lets` is the innermost binding of a `Com`. This forms the zipper, which splices the `Com` into a `Com` and a `Lets`. Also, while `Com` tracks only the return type α in the type index, `Lets` tracks the entire resulting context Δ. That is, in `(lets : Lets` Γ Δ`)`, the first context, Γ, lists all free variables (just as in `Com` Γ `t`), but the second context, Δ, consists of all variables in Γ plus a new variable for each let-binding in the sequence `lets`. We can thus think of Δ as the context at the current position of the zipper. Another difference is the order in which these sequences grow. Recall that in `Com`, the outermost constructor represents the topmost let-binding. In `Lets`, the outermost constructor instead corresponds to the bottommost let-binding. This difference is what makes the zipper work.

We have two functions to go from a program to a zipper and back: (1) `(splitProgramAt pos prog)`, to create a zipper from a program `prog` by moving the specified number of bindings to a new `Lets` sequence, and (2) `(addComInMiddleOfLetCom top mid bot)`, to turn a zipper `top, bot` into the program, while inserting a program `mid : Com` in between. We also prove that the result of splitting a program with `splitProgramAt` is semantically equivalent to the original program. Similarly, we prove that stitching a zipper back together with `addComInMiddleOfLetCom` results in a semantically equivalent program.

Given a peephole rewrite (`matchCom`, `rewriteCom`), to rewrite at location i, we first split the target program into `top` and `bot`. We then attempt to match the def-use chain of the return variable in `matchCom` with the final variable in `top` (which is the target i, since we split the program there). This matching of variables recursively matches the entire expression tree.[6] Upon successful matching, this returns a substitution σ for the free variables in `matchCom` in terms of (free or bound) variables of `top`. Using this successful matching, we stitch the program together as `top`$;\sigma($`rewriteCom`$);\tau($`bot`$)$. Here, τ is another substitution that replaces the variable at location i with the return variable of `rewriteCom`. Since we derived a successful matching, we know that the semantics of variable i is equal to that of the return variable of `matchCom`. By assumption on the peephole rewrite, the variable i is equivalent to the return variable of `rewriteCom`. This makes it safe to replace all occurrences of the variable i in `bot` with the return variable of `rewriteCom`. This proves `denote_rewritePeephole`, which states that if a rewrite succeeds, then the semantics of

[6] We match regions in expressions for structural equality. We *do not* recurse into regions during matching, and treat regions as black-boxes.

the program remain unchanged. In this way, we use a zipper as a key inductive reasoning principle to mechanize the proof of correctness of SSA-based peephole rewriting.

4.2 DCE & CSE: Folding Over Intrinsically Well Typed SSA

The classic optimizations enabled by SSA are peephole rewriting, dead code elimination (DCE), and common subexpression elimination (CSE). We implement these optimizations in our framework as a test of its suitability for metatheoretic reasoning. Our approach is different from previous approaches [41, 5] with our use of intrinsic well-typing, which mandates proofs of the structural rules on contexts to rewrite programs. We begin by building machinery to witness that a context Δ is equal to the context Γ, minus the variable x. This is spelled as `Deleted` Γ x Δ in `LeanMLIR`(X). We then prove context-strengthening theorems to delete variables that do not occur in `Expr` and `Com` while preserving denotation.

Using this tooling, DCE is implemented in ≈ 400 LoC, which shows that our framework is well-suited to metatheoretic reasoning. The implementation is written in a proof-carrying style, interleaving function definitions with their proof of correctness. The recursive step of the dead code elimination takes a program p : `Com` Γ `t` and a variable v to be deleted, and returns a new p' : `Com` Δ `t`. The two contexts Γ and Δ are linked by a context morphism `(Hom` Γ Δ`)`, to interpret p' (with the deleted variable) which lives in a strengthened context Δ in the old context Γ. We walk p recursively to eliminate dead values at each `let` binding. This produces a new p' with dead bindings removed, a proof of semantic preservation, and a context morphism from the context of p to the strengthened context of p' with all dead variables removed.

Similarly, the CSE implementation folds over `Com` recursively, maintaining data structures necessary to map variables and expressions to their canonical form. At each (`let` $x = f(v_1, ...v_n)$ `in` b) step, we canonicalize the variables v_i to find variables c_i. We then look up the canonicalized expression $f(c_1, \ldots, c_n)$ in our data structure to find the canonical variable c_x if it exists and replace x with c_x. If such a canonical c_x does not exist, we add a new entry mapping $f(c_1, \ldots, c_n)$ to x, thereby canonicalizing any further uses of this expression.

4.3 Proof Automation for Goal State Simplification in `LeanMLIR`(X)

The proof automation tactic `simp_peephole` Γ (used to eliminate framework definitions from the goal state) takes a context Γ, reduces its type completely, and abstracts out program variables to provide a theorem statement that is universally quantified over the variables of the program, with all framework definitions eliminated. It uses a set of equation theorems to normalize the type of Γ. This is necessary to extract the types of variables during metaprogramming. Once the type of Γ is known, we simplify away all framework definitions (such as `Expr.denote`). We then replace all occurrences of a variable accesses $\Gamma[i]$ with a new (Lean, i.e., host) variable. We do this by abstracting terms of the form $\Gamma[i]$ where i is the i-th variable. This gives us a proof state that is universally quantified over variables from the context. Finally, we clear the context away to eliminate all references to the context Γ. The set of definitions we simplify away is extensible, enabling us to add domain-specific simplification rewrites for the IR.

5 Pure Rewriting in a Side-Effectful World

While `LeanMLIR`(X) streamlines the verification of higher-level IRs that use only value semantics, typical IRs may interleave islands of pure operations (with value semantics) with operations that carry side effects. An IR that is user-facing can usually be rephrased with high-level, side-effect-free semantics. Yet, each operation in such an IR is compiled through a sequence of IRs that are lower level and potentially side-effectful. For example, in the case of FHE, the pure `FHE` IR is compiled to a lower-level IR that encodes the coset representative of each ideal as an array, with control flow represented via structured control flow (`scf`). Eventually, this is compiled into LLVM which is rife with mutation and global state. In such a compilation flow, peephole rewrites are used at each intermediate IR to optimize pure fragments while leaving side-effectful fragments untouched. An effective compiler pipeline introduces the right abstractions to maximize rewrites on side effect-free fragments.

`LeanMLIR`(X) is designed to facilitate verification of peephole rewrites as they arise in such a compiler pipeline. The previous sections already presented how our framework supports the verification of peephole rewrites in a pure setting. Yet, our design also allows for the optimization of a pure fragment in a side-effectful context. We have a mechanized proof of the correctness of the extended framework with support for side effects and a rewrite theorem that performs pure rewrites in the presence of side effects. The key idea is to annotate each `Op` with an `EffectKind`, where `EffectKind.pure` changes the denotation of the `Expr` into the `Id` monad, while `EffectKind.impure` denotes into an arbitrary, user-chosen, IR-specific monad. We also introduce a new notion of monadic evaluation of `Lets`, which returns a valuation plus a proof that, for every variable v that represents a pure expression e in the sequence of let-bindings, the valuation applied to v agrees with the (pure) denotation of e. This proof-carrying definition allows us to use this invariant when reasoning inside a subexpression of a monadic bind.

With the above at hand, the overall rewriter construction and proof strategy remains unchanged, with the additional constraint of performing rewriting only on those operations marked as `EffectKind.pure`, and the surrounding monadic ceremony required to show that a pure rewrite indeed does not change the state of pure variables in various lemmas.[7]

6 Case Studies

We mechanize three IRs based on ones found in the MLIR ecosystem as case studies for `LeanMLIR`(X) and show how they benefit from the different aspects of our framework. Note that the core of our framework (definitions of `Expr`, `Com`, `PeepholeRewrite`, lemmas about these objects, and the peephole rewriting theorem) is $\approx 2.2k$ LoC. The case studies based on our framework together are $\approx 5.6k$ LoC, which stresses the framework to ensure that it scales to realistic formal verification examples.

6.1 Reasoning About Bitvectors of Arbitrary Width

We first demonstrate our ability to reason about a well-established domain of peephole rewrites: LLVM's arithmetic operations over fixed-bitwidth integers. Using the Z3 SMT solver [7], the Alive project [21, 20] can efficiently and automatically reason about these. Notably, at the time of this writing, almost 700 LLVM patches have justified their correctness by referencing Alive. In this way, accessible proof tools can find a place in production compiler development workflows. However, Alive is limited by the capabilities of the underlying SMT solvers. SMT solvers are complex, heuristic-driven, and sometimes even have soundness

[7] A limitation of our current mechanization is that we assume that all regions are potentially side-effecting. This is a simplification that shall be addressed in a newer version of the proof.

bugs [38]. They are also specialized to support very concrete theories. Among others, this means Alive can only reason about a given fixed bitwidth. Even recent work that specifically aims to generalize rewrites to arbitrary bitwidths, can only exhaustively test a concrete set of bitwidths [23]. Using our framework, we can reproduce Alive-style correctness proofs, and extend them to reason about arbitrary (universally quantified) bitwidths. This ability to handle arbitrary bitwidth is important in verification contexts that have wide bitvectors, as they can occur in real-life VLSI problems [12, 36]. MLIR itself has multiple IRs that require bitvector reasoning: `comb` for combinational logic in circuits, `arith` and `index` for integer and pointer manipulation, and `llvm` which embeds LLVM IR in MLIR. Our streamlined verification experience offers developers an Alive-style workflow for the `llvm` dialect, while allowing reasoning across bitwidths. As our framework is extensible, we believe we can also support other MLIR dialects that require bitvector reasoning, such as `comb` and `arith`.

6.1.1 Modeling a fragment of LLVM IR: Syntax and Semantics

To test our ability to reason about bitvectors in practice, we model the semantics of the arithmetic fragment of LLVM as the IR `LeanMLIR`(LLVM). We support the (scalar) operators: `not`, `and`, `or`, `xor`, `shl`, `lshr`, `ashr`, `urem`, `srem`, `add`, `mul`, `sub`, `sdiv`, `udiv`, `select` and `icmp`. We support all `icmp` comparison flags, but not the strictness flags `nsw` and `nuw`.

At the foundation of our denotational semantics is Lean's `BitVec` type, which models bitvectors of arbitrary width and offers `smtlib` [4] compatible semantics. However, when we started this work, most bitvector operations were not defined in the Lean ecosystem and the bitvector type itself was not fully fleshed out. Hence, we worked with the `mathlib` and Lean community to build and upstream a theory of bitvectors.[8] After developing the core theory in `mathlib`, Lean's mathematical library, development subsequently moved into Lean core, where we continue to evolve Lean's bitvector support.

The semantics of LLVM's arithmetic operations follow the semantics of `smtlib` (and consequently Lean's) bitvectors closely. In case of integer wrapping or large shifts, for example, LLVM can produce so-called poison values [21], which capture undefined behavior as a special value adjoined to the bitvector domain. LLVM's poison is designed not to be a side effect and, consequently, can be reasoned about in a pure setting. In contrast, `ub` is a side effect that triggers immediate undefined behaviour, and can be refined into any behavior. In LLVM, the following refinements are legal: `ub` $\sqsubseteq$ `poison` $\sqsubseteq$ `val`. Among the instructions we model, division and remainder can produce immediate undefined behavior `ub`. In our framework, we approximate these by collapsing the side-effectful undefined behavior and side-effect-free poison both into `Option.none`. We thus denote bitvectors into the type `Option (BitVec w)`. This is safe for the three kinds of rewrites we consider: (1) the left- and right-hand sides are both UB and poison free (arithmetic rewrites), (2) the left- and right-hand sides are both UB free (bitwise rewrites, where left and right shifts may produce poison), (3) the left-hand side may trigger UB, while the right hand side may only produce poison (e.g., refining division into arithemtic operations). We leave separating UB as a side effect distinct from poison, and reasoning about peephole rewrites which refine such side effects as interesting future work.

For side-effect-free programs, our semantics match the LLVM semantics. We perform

[8] github.com/leanprover-community/mathlib4/pull/{5383,5390,5400,5421,5558,5687,5838,5896,7410, 7451,8231,8241,8301,8306,8328,8345,8353}, github.com/leanprover/lean4/pull/{3487,3471,3461,3457,3445,3492,3480,3450,3436}, github.com/leanprover/std4/pull/{357,359,599,626,633-636,637,639,641,645-648,655,658-660,653}

exhaustive enumeration tests between our semantics and that of LLVM. We take advantage of the fact that an IR with computable semantics automatically defines an interpreter in our framework. We build an executable program that runs every instruction, with all possible input combinations upto bitwidth 8. We get LLVM's ground truth by using LLVM's optimizer, `opt` to transform the same instruction with constant inputs. This optimizes the program into a constant output, handling undefined behavior. By exhaustive enumeration, our tested executable semantics correspond to the LLVM semantics wherever the result is `Option.some`, and also soundly model undefined behavior whenever the result is `Option.none`. This gives us confidence our semantics correspond to LLVM's.

6.1.2 Proving Bitvector Rewrites in our Framework

Effective automation for bitvector reasoning is necessary to resolve the proof obligations that LeanMLIR(X) derives automatically from peephole rewrites expressed as MLIR program snippets. While Lean does not yet have extensive automation for bitvectors, thanks to our work we can already use a decision procedure for commutative rings [10] and an extensionality lemma that establishes the equality of bitvectors given equality on an arbitrary bit index.

We test the available automation on a dataset of peephole optimizations from Alive's test suite, consisting of theorems about addition, multiplication, division, bit-shifting and conditionals. Out of the 435 tests in Alive's test suite, we translate 93 tests which are the ones that are supported by the LLVM fragment we model and without preconditions. We prove 54 of these rewrites from the Alive test suite automatically. Some rewrites cannot be handled automatically. Of those where automation struggles, we manually prove an additional 6, selecting the ones where an SMT solver takes long to prove them even for a specific bitwidth (e.g., 64). Our proofs are over arbitrary (universally quantified) bitwidth, save for some theorems that are only true at particular bitwidths.[9] As an example, let us consider the rewrite:

```
example (w : Nat) :
  [llvm ( w )| {
    ^bb0(%X : _, %C1 : _, %C2 : _):
      %v1 = llvm.xor %X, %C1
      %v2 = llvm.and %v1, %C2
      llvm.return %v2
  }] ⊑ [llvm ( w )| {
    ^bb0(%X : _, %C1 : _, %C2 : _):
      %v1 = llvm.and %X, %C2
      %v2 = llvm.xor %X, %C1
      %v3 = llvm.and %C1, %C2
    %v4 = llvm.xor %v1, %v3
    llvm.return %v4
  }] := by simp_alive_peephole; alive_auto
```

Note that due to the support of MLIR syntax in our framework, this rewrite is specified in MLIR syntax. We use a custom extension with the placeholder syntax `_`, to stand for an arbitrary bitwidth w. After simplification of the framework code with `simp_peephole`, this yields the proof obligation:

```
(w :  Nat) (X C1 C2 :  BitVec w) ⊢ (X ^^^C1) &&& C2 = X &&& C2 ^^^C1 &&& C2
```

This proof obligation only concerns the semantics in the semantic domain of bitvectors, it does not feature MLIR and SSA anymore. This goal is automatically proven by our proof

[9] e.g., `a + b = a xor b` is true only at bitwidth 1.

```
/-- only control flow operations, parametric over another IR Op'  -/
inductive Op (Op': Type) [OpDenote Op' Ty'] : Type
| coe (o : Op') -- coerce Op' to Op
| for (ty : Ty') -- a for loop whose loop carried data is Ty'

instance [I : HasTy Op' Int] : OpSignature (Op Op') Ty' where
  signature
  | .coe o => signature o
  | .for t => ⟨[/-start-/I.ty, /-step-/I.ty, /-niters-/N.ty, /-v-/t],
     /- region arguments: -/ [([/-i-/I.ty, /-v-/t], /-v'-/t)],
     /-return-/t⟩
instance [I : HasTy Op' Int] [OpDenote Op' Ty']: OpDenote (Op Op') Ty' where
  denote
  | .coe o', args', regArgs' => OpDenote.denote o' args'regArgs' -- reuse denotation of o'
  | .for ty, [istart, istep, niter, vstart]ₕ, [f]ₕ =>
        let istart : ℤ := I.denote_eq ▶ istart -- coerce to `int`.
        ... -- coerce other arguments
        let loop_fn := ... -- build up the function that's iterated.
        (loop_fn (istart, vstart)).2
```

Figure 5 Simplified implementation of `LeanMLIR`($scf(X)$) Observe that the IR is parametrized over another IR `Op'`, and that we add control flow to the other IR in a modular fashion.

automation for bitvectors, `alive_auto`. In the longer term, we aim to also connect our work to a verified SAT checker that is under development.[10]

6.2 Structured Control Flow

The examples of IRs we have seen so far are all straight-line code. In this use case, we show how we can add control flow to existing IRs, thanks to the parametricity of our framework. We also demonstrate how encoding control flow structures as regions enable succinct proofs for transformations, by exploiting the high-level structure of these operations. To this end, we model structured control flow as a fragment of the `scf` IR in MLIR, by giving semantics to two common kinds of control flow: `if` conditions and bounded `for` loops. Note that we choose to model *bounded* for loops, since these are the loops that are used in MLIR to model high-level operations such as tensor contractions. A pleasant upshot is that these guaranteed to terminate, and can thus have a denotation as a Lean function without requiring modelling of nontermination (which is side-effectful). Our sketch of the extended framework with side effects will be used to pursue this line of research in the future. The conditionals and bounded for loop allow us to concisely express loop canonicalizations and transformations from MLIR in `LeanMLIR`(`scf`). These operations allow us to concisely express loop canonicalizations and transformations in `LeanMLIR`(`scf`).

We built this parametrically over an existing IR X to allow these constructs to be added to an existing IR X. The key idea is that the `Op` corresponding to `scf` is parametrized by the `Op` corresponding to another IR X. Since the only datatypes `scf` requires are booleans and natural numbers, we ask that the type domain of X contains these types. We then provide denotations in for booleans and integers from the type domain of X. Thus, what we encode is `LeanMLIR`($scf(X)$), which is an IR for structured control flow parametrized by another, user-defined IR X.

The `scf.for` operation (Figure 5) has three arguments: the number of times the loop is to be executed, a starting and step value for the iteration, and a seed value for the loop to

[10] https://github.com/leanprover/leansat

iterate on. Note that in the definition, the IR `Op` is defined parametrically over another IR `Op'`, and the types of `Op` are the same as the types of other IR `Ty'`. We perform a similar construction for `if` conditions.

The denotation of the `for` loop, as well as theorems about loop transformations, follow from `mathlib`'s theory for iterating functions (`Nat.iterate`). The loop body in `scf.for` has a region that receives the current value of the loop counter and the current iterated value and returns the next iterated value. We prove the inductive invariant for loops using the standard theory of iterated function compositions ($f^0 = id$, $f^k \circ f^l = f^{k+l}$, $id^k = id$). We also prove common rewrites over loops: running a `for` loop for zero iterations is the same as not running a loop at all (dead loop deletion), two adjacent loops with the same body can be fused into one when the ending index of the first loop is the first index of the second loop (loop fusion), and a loop whose loop body does not depend on the iteration count can be reversed (loop reversal). Similarly, we prove that `if` $true\ e\ e' = e$, and `if` $false\ e\ e' = e'$.

These do *not* count as peephole rewrites in our framework, as they are universally quantified over the loop body (which is a region). This is unsupported — peephole rewrites in `LeanMLIR`(X) may only have free variables, not free region arguments. Increasing the power of peephole rewrites with arbitrary regions is an interesting question for future work.

Consider the loop optimization that converts iterated addition into a single multiplication. Its proof obligation is $(\vdash \lambda x.\ x + \delta)^n(c) = n \cdot \delta + c$ (a short proof by induction on n). This transformation is challenging to perform in a low-level IR, since there is no syntactic concept of a loop. However, this transformation *is* a valid peephole rewrite in our framework since it uses a *statically* known loop body. We showcase how regions permit MLIR (and, consequently, us) to easily encode and reason with commonplace loop transformations. Importantly, the parametricity of our framework allows us to prove a set of these as local peephole rewrites that are valid on all IR extensions `scf`(X).

6.3 Fully Homomorphic Encryption

A key motivation for `LeanMLIR`(X) is to enable specifying formal semantics for high-level, mathematical IRs. These IRs require access to complex mathematical objects that are available in proof assistants, and verifying rewrites on such IRs is out of practical reach for today's SMT solvers. As a case study, we formalize the complete "Poly" IR.[11] This IR is a work in progress and is in flux, as it is part of the discussion of an upcoming open standard for homomorphic encryption, developed in collaboration by Intel and Google.[12] Contrary to what its naming implies, this IR does *not* model operations on polynomials.[13] Instead, codewords are encoded as elements in a finitely-presented commutative ring, specifically, the ring $R \equiv (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n} + 1)$, where $q, n \in \mathbb{N}$ are positive integers (q composite). The name "Poly" comes from the equivalence class representatives are polynomials, but not all IR operations are invariants of the equivalence class.

The "Poly" IR is, in fact, a superset of the `QuotRing` IR we defined in Section 2. It consists of the operations `add`, `sub`, `mul`, `mul_constant`, `leading_term`, `monomial`, `monomial_mul`, `from_tensor`, `to_tensor`, `arith.constant` and `constant`.[14]

Most of these operations are self-explanatory and derive from the (commutative) ring structure of R or are used to build elements in R, like the equivalence classes of constants

[11] as of commit 2db7701de
[12] `https://homomorphicencryption.org/`
[13] In the same way that rationals $\mathbb{Q}$ are not pairs of integers $\mathbb{Z} \times \mathbb{Z}$.
[14] It also has distinct types for integers and naturals, which we unified in Section 2 for simplicity.

or monomials. Three operations, `to_tensor` and `from_tensor` and `leading_term` do not follow directly from the algebraic properties of the polynomial ring. Instead, they depend on a (non-canonical) choice of representatives for each ideal coset in the polynomial ring. More precisely, let $\pi : (\mathbb{Z}/q\mathbb{Z})[x] \twoheadrightarrow (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n}+1)$ be the canonical surjection into the quotient, taking a polynomial to its equivalence class modulo division by $x^{2^n}+1$. Further let $\sigma : (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n}+1) \hookrightarrow \mathbb{Z}/q\mathbb{Z}[x]$ be the injection taking an equivalence class to its (unique) representative with degree $\leq 2^n$. This is a right-inverse of π, i.e. $\pi \circ \sigma = id$. Note that multiple right-inverses that could have been chosen for σ, as long as $\sigma(x)$ is a representative of the equivalence class of x for all $x \in (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n}+1)$, σ will be a right-inverse of π. The operation `to_tensor(p)` returns the vector $(\sigma(p)[i])_{i=0,\ldots,2^n}$, where $a[i]$ represents the i-th coefficient, i.e. $\sigma(p) = \sum_{i=0}^{2^n}(\sigma(p)[i])x^i$, and `to_tensor` the converse. Similarly, `leading_term(p)` returns the equivalence class of the leading term of the representative $\sigma(p)$ (which also depends on the choice of σ).

This allows us to define the semantics and prototype both the IR and rewrites in it. Rewrites like `mul(p,q)` $\rightarrow$ `mul(q,p)` follow immediately from the fact that R is a commutative ring. Other rewrites like `from_tensor(to_tensor(p))` $\rightarrow$ `p`, or even `add(p,monomial(1,`2^n`))` $\rightarrow$ `sub(p,1)`, on the other hand, are more specific to this IR and have a higher manual-proof overhead. We prove all of these.

We discussed the IR and potential semantics with the authors of the HEIR IR in the context of the upcoming open standard for homomorphic encryption. We believe that a framework like the one presented in this paper will allow standards like these to be defined with formal semantics from the ground up.

7 Related Work

Alive [21, 18] and Alive 2 [20] provide push-button verification for a subset of LLVM by leveraging SMT solvers. Alive-tv does the same for a set of concrete IRs for tensor operations in MLIR [3]. The semantics and correctness of compiling compositionally have been explored by multiple authors, like Pilsener [25] or many variants of CompCert [19]: like compositional CompCert [32], CompCertX [37], SepCompCert [13], CompCertM [31], and CompCertO [15]. A great summary of the approaches to this problem (including the ones mentioned above), with their differences and similarities, is given by Patterson et al [27]. All of these use fixed languages but are reasonable ways of giving semantics to relevant IRs in `LeanMLIR`(X). Our semantics is denotational and can be executed, like interaction trees [39, 40].

Our work differs from prior work on formalizing peephole rewrites by providing a framework for reasoning about SSA peephole rewrites. The closest similar work, Peek [24] defines peephole rewriting over an assembly instruction set. Their rewriter expects instructions to be adjacent to one another. Furthermore, their rewriter restricts source and target patterns to be of the same length, filling in the different lengths with `nop` instructions. Their patterns permit side effects, which we disallow since we are interested in higher-level, pure rewrites. Our patterns provide more flexibility since the source and target patterns are arbitrary programs, and are matched on sub-DAGs instead of a linear sequence.

8 Conclusion

Peephole rewrites represent a large and important class of compiler optimizations. We have seen how domain-specific IRs in SSA with regions greatly extend the scope of these peephole rewrites. They raise the level of abstraction both syntactically with def-use chains and nesting, and semantically, with domain-specific abstractions. We have shown how to reason effectively about such SSA-based compilers, and, specifically, local reasoning in the form of peephole rewrites. We advocate building on top of a proof assistant with a small TCB, an expressive language and a large library of mathematics. This increases the confidence in our verification and extends its applicability to many domains where more specialized methods don't exist. We also advocate proof automation and an intrinsically well-typed mechanized core that can be designed to focus on the semantics of the domain. We incarnate these principles in `LeanMLIR`(X), a framework built on Lean and `mathlib` to reason about domain-specific IRs in SSA with regions. We show how `LeanMLIR`(X) is simple to use, amenable to automation, and effective for verifying IRs over complex domains.

References

1 AV Aho, R Sethi, and JD Ullman. Compilers: Principles, techniques, and tools. 1985.

2 Andrew W Appel. SSA is functional programming. *Acm Sigplan Notices*, 33(4):17–20, 1998.

3 Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. SMT-based translation validation for machine learning compiler. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 386–407. Springer, 2022. doi:10.1007/978-3-031-13188-2_19.

4 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

5 Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):1–35, 2014.

6 Siddharth Bhat and Tobias Grosser. Lambda the ultimate ssa: optimizing functional programs in ssa. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE, 2022.

7 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

8 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.

9 Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

10 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. doi:10.1007/11541868_7.

11 Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

12 Petter Källström and Oscar Gustafsson. Fast and area efficient adder for wide data in recent Xilinx FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

13 Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, 2016.

14 Richard A Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

15 Jérémie Koenig and Zhong Shao. CompCertO: compiling certified open C components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1095–1109, 2021.

16 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

17 Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

18 Juneyoung Lee, Chung-Kil Hur, and Nuno P Lopes. AliveInLean: a verified LLVM peephole optimization verifier. In *International Conference on Computer Aided Verification*, pages 445–455. Springer, 2019.

19 Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

20 Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

21 Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2015.

22 William M McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

23 Manasij Mukherjee and John Regehr. Hydra: Generalizing peephole optimizations with program synthesis. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2024.

24 Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 448–461, 2016.

25 Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 166–178, 2015.

26 Sunjae Park, Woosung Song, Seunghyeon Nam, Hyeongyu Kim, Junbum Shin, and Juneyoung Lee. HEaaN.MLIR: An optimizing compiler for fast ring-based homomorphic encryption. *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2023.

27 Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

28 Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. QSSA: an SSA-based IR for quantum computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 2–14, 2022.

29 Benjamin C Pierce and C Benjamin. *Types and programming languages.* MIT press, 2002.

30 Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based Compiler Design.* Springer Nature, 2022.

31 Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

32 Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 275–287, 2015.

33 Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In *International Joint Conference on Automated Reasoning*, pages 167–182. Springer, 2020.

34 Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. *CoRR*, abs/2202.03293, 2022. URL: `https://arxiv.org/abs/2202.03293`, `arXiv:2202.03293`.

35 Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: Automatic code optimizations for efficient fully homomorphic encryption. *arXiv preprint arXiv:2202.01649*, 2022.

36 Wei Wang and Xinming Huang. A novel fast modular multiplier architecture for 8,192-bit RSA cryposystem. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2013.

37 Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

38 Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. `doi:10.1145/3385412.3385985`.

39 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *arXiv preprint arXiv:1906.00046*, 2019.

40 Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, 2021. URL: `https://doi.org/10.1145/3473572`.

41 Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 175–186, 2013.