First-Class Verification Dialects for MLIR

MATHIEU FEHR, University of Edinburgh, United Kingdom
YUYOU FAN, University of Utah, USA
HUGO POMPOUGNAC, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG Grenoble, France
JOHN REGEHR, University of Utah, USA
TOBIAS GROSSER, University of Cambridge, United Kingdom

MLIR is a toolkit supporting the development of extensible and composable intermediate representations (IRs) called *dialects*; it was created in response to rapid changes in hardware platforms, programming languages, and application domains such as machine learning. MLIR supports development teams creating compilers and compiler-adjacent tools by factoring out common infrastructure such as parsers and printers. A major limitation of MLIR is that it is syntax-focused: it has no support for directly encoding the semantics of operations in its dialects. Thus, at present, the parts of MLIR tools that depend on semantics—optimizers, analyzers, verifiers, transformers—must all be engineered by hand.

Our work makes formal semantics a first-class citizen in the MLIR ecosystem. We designed and implemented a collection of semantics-supporting MLIR dialects for encoding the semantics of compiler IRs. These dialects support a separation of concerns between three domains of expertise when building formal-methods-based tooling for compilers. First, compiler developers define their dialect's semantics as a lowering (compilation transformation) from their dialect to one or more of ours. Second, SMT solver experts provide tools to optimize domain-specific high-level semantics and lower them to SMT queries. Third, tool builders create dialect-independent verification tools.

We validate our work by defining semantics for five key MLIR dialects, defining a state-of-the-art SMT encoding for memory-based semantics, and building three dialect-agnostic tools, which we used to find five miscompilation bugs in upstream MLIR, verify a canonicalization pass, and also formally verify transfer functions for two dataflow analyses: "known bits" (that finds individual bits that are always zero or one in all executions) and "demanded bits" (that finds don't-care bits). The transfer functions that we verify are improved versions of those in upstream MLIR; they detect on average 36.6% more known bits in real-world MLIR programs compared to the upstream implementation.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: compilers, verification, intermediate representations

ACM Reference Format:
Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proc. ACM Program. Lang.* 9, PLDI, Article 206 (June 2025), 25 pages. https://doi.org/10.1145/3729309

1 Introduction

An intermediate representation (IR) is, effectively, a specialized programming language that exists only ephemerally inside of a compiler. Modern compilers often contain multiple IRs to facilitate analyses and transformations at widely different levels of abstraction. For example, Rust [28] and

Authors' Contact Information: Mathieu Fehr, University of Edinburgh, Edinburgh, United Kingdom, mathieu.fehr@ed.ac.uk; Yuyou Fan, University of Utah, Salt Lake City, USA, yuyou.fan@utah.edu; Hugo Pompougnac, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG Grenoble, France, hugo.pompougnac@inria.fr; John Regehr, University of Utah, Salt Lake City, USA, regehr@cs.utah.edu; Tobias Grosser, University of Cambridge, Cambridge, United Kingdom, tobias.grosser@cst.cam.ac.uk.

ACM 2475-1421/2025/6-ART206
https://doi.org/10.1145/3729309

Swift [34] both have a high-level language-specific IR (MIR and SIL, respectively), which is lowered to LLVM IR, then passes through the low-level "machine IR" before reaching assembly language.

IRs have traditionally been second-class citizens regarding tooling, specification, and documentation; this made sense in the past when each IR was being used by a small, close-knit group of compiler developers. However, it makes much less sense in the modern era of open-source compilers, where the IR has become an extension point, supporting the interposition of new front-ends, backends, optimization passes, analysis tools, and more. Moreover, in an era of rapid innovation in programming languages and hardware platforms, IRs have come under intense pressure to be rapidly developed and evolved. Over the past several years, MLIR [19]—the Multi-Level Intermediate Representation—has emerged as a partial solution to these problems. The MLIR infrastructure supports the creation of *dialects*, which are partial or complete IRs. It provides substantial support for creating parsers, printers, analyses, transformations, and other tools that need to exist for every IR. Perhaps the most interesting feature of MLIR is that it supports the fine-grained composition of dialects: multiple dialects can be used even within a single expression.

MLIR, as it currently exists, is syntax-focused: there is no support for defining the semantics of MLIR-based IRs. Semantics, however, are crucial when showing that analyses, optimizations, and other transformations are behaving correctly. For example, the CompCert C compiler [25] formally defines the semantics of its IRs in Rocq to prove the correctness of its passes. Other projects, such as LLVM [18], define the semantics of their IR only informally,[1] leaving the task of formalization to third parties such as Vellvm [40] and Alive2 [26].

The late, external formalization of LLVM IR revealed numerous ambiguities and even semantics-level defects in LLVM IR, necessitating difficult engineering work such as the introduction of the new "freeze" instruction [21]. Our work is based on the premise that from this point forward, IRs should be (at least partially) formalized early in their development. From this point of view, MLIR offers an unparalleled opportunity to make formal semantics into an integral part of compiler IR design. However, MLIR also comes with significant challenges. First, formal-methods-based tools for compilers such as LLVM or GCC have monolithic designs specifically crafted for their target environment. MLIR is an open system: we cannot anticipate what kinds of dialects people will define in the future. What is needed is basic infrastructure for not only helping compiler experts to define their dialect semantics, but also for formal verification experts to define an efficient encoding of these semantics to SMT queries, and for tool builders to create verification tools that do not rely on particular dialect semantics.

This paper: We support the definition of IR semantics at the framework level—in MLIR—rather than doing so after the fact at the level of a particular compiler IR. This supports the derivation of semantics-based tools "for free" and allows the reuse of otherwise siloed state-of-the-art mapping of high-level semantics to SMT. Since our thesis is difficult to evaluate directly, we indirectly evaluate it by showing that we can use our tools to build a translation validation tool, a peephole rewriter, and a verified dataflow analysis. By design, these tools can be used with any sufficiently compatible MLIR IR for which the developer can provide formal semantics.

Our strategy for defining the semantics of IRs within MLIR is to define a new set of dialects for expressing semantics—that we call, naturally enough, *semantic dialects*—that support encoding semantics as SMT expressions for IR dialects consisting of side-effect-free operations. We additionally define a set of higher-level dialects for expressing side effects, such as memory accesses and undefined behavior. All the semantic dialects are eventually lowered to SMT-LIB [7] queries.

[1] https://llvm.org/docs/LangRef.html

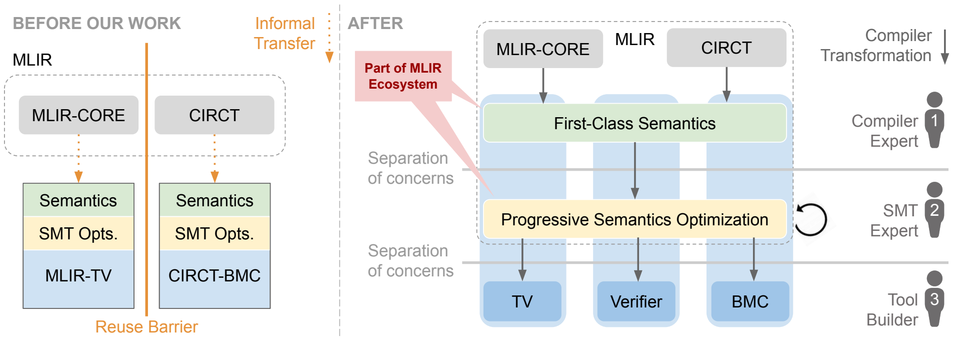


Fig. 1. *First-Class Semantics* as an integral part of the (MLIR) compiler ecosystem enable (1) compiler experts to manipulate semantics as part of their daily workflow, using familiar compiler technology; (2) SMT experts to optimize the SMT encoding while offering intuitive domain-specific interfaces to the compiler; and (3) tool builders to work across different dialects following fast-evolving dialects with ease. While previous semantics-based tools are kept apart by reuse-barriers between tools and an informal semantic connection to the compiler ecosystem, our proposal moves semantics into the core of the compiler.

By reusing the same IR infrastructure used to optimize a program to define semantics, we let compiler experts define semantics in a language they are already familiar with. This abstraction also allows tools to be built without relying on a specific dialect's semantics. Additionally, defining semantics as a combination of SMT-LIB expressions and higher-level dialects allows SMT experts to provide more efficient encodings of these semantics to SMT queries. Our contributions are:

- A set of reusable semantic dialects, composed of SMT-LIB-based dialects to interface with SMT solvers and higher-level semantic dialects to represent side-effects such as memory accesses and undefined behavior
- The extraction of a state-of-the-art SMT encoding of sequential memory semantics as a reusable compiler transformation, and domain-specific optimizations on our high-level semantic dialects to improve the efficiency of SMT queries
- A compilation transformation from key control-flow free MLIR dialects ('arith', 'func', 'builtin', 'memref', and 'comb') to our semantic dialects
- The creation of three semantics-agnostic tools for (1) translation validation, (2) verifying peephole rewrites, and (3) verifying dataflow analyses
- The usage of our verification tools to find five miscompilation bugs in upstream MLIR, verify a full canonicalization pass, and formally verify new transfer functions for a "known bits" and a "demanded bits" analysis for the 'comb' dialect in CIRCT

2 A Vision for Semantics-Based Compiler Engineering

The fundamental premise behind this paper is: *Defining the formal semantics of MLIR dialects early, and in an idiomatic MLIR style, supports the creation of valuable semantics-based tools and imposes a relatively low cost on compiler developers.* In contrast, at present, most IRs do not have a formal semantics at all, much less one that is created early on. We make—and evaluate in Section 6—the following specific claims:

1) Special-purpose dialects for expressing semantics are an effective way to formalize the meaning of MLIR dialects. In our work, the semantics of MLIR *program dialects*—those used to express application logic—are defined by how they are lowered to our set of *semantic dialects*, which are eventually translated into SMT-LIB queries. This strategy allows MLIR developers to continue using tools and compilation strategies with which they are familiar. The current state of the practice in the MLIR community is to define the semantics of a program dialect implicitly by how it is lowered to other dialects. This leads to loss of information across abstraction boundaries, and sometimes, conflicts can arise when a dialect supports multiple lowerings. In contrast, our semantic dialects are intended to be ergonomic and to be a single source of semantic truth for a program dialect. They are purpose-built for encoding programming language semantics: they directly and explicitly represent common features such as undefined behavior and memory.

2) Providing multiple levels of semantic dialects enables efficient SMT encodings. Similar to how MLIR dialects are defined at different levels of abstraction, we believe that dialects for expressing formal semantics must also work at different levels of abstraction. Furthermore, the presence of different levels of abstraction of semantic dialects allows SMT experts to provide a more efficient, and sometimes domain-specific, encoding of these semantics to SMT queries, which can be reused across different dialects and projects. Overall, we believe that MLIR's multi-dialect design, which is used to optimize high-level programs, can also be applied to represent MLIR semantics efficiently.

3) Formal verification tools for MLIR can be defined at the framework level. Once semantics are defined as a lowering to a semantic dialect, we can use these to give compiler developers useful tools. For example, translation validation is straightforward to implement. We have also supported MLIR's `pdl` dialect, which is used to define rewrite rules for other dialects, supporting ahead-of-time formal verification of rewrites (in contrast with translation validation, which validates only a single execution of a transformation at a time). Third, we have created a new `transfer` dialect; when dataflow transfer functions are implemented with it, they can be proved to be sound approximations of concrete instructions that make up dialects. Operations in the `transfer` dialect can also be lowered to C++ code that is suitable for inclusion in MLIR's dataflow framework, resulting in complete dataflow analyses. Since the tooling we are creating is generic and dialect-independent, it can be reused by MLIR developers to define their dialect's semantics using our semantic dialects.

Representing semantics as a first-class IR through our semantic dialects is the central idea of our architecture (Figure 1). It serves as the foundation for the declarative semantics we introduce for transfer functions and rewrites and allows us to derive our three semantics-agnostic tools, all of which are optimized using domain-specific SMT-based encodings.

3 Background

We first introduce intermediate representations in compilers, their realization in the MLIR and xDSL [13] compiler frameworks, and then the CIRCT EDA (electronic design automation) toolchain.

3.1 Static Single Assignment Intermediate Representations

Modern compilers use IRs for storing, analyzing, and transforming programs. The foundation of a typical imperative IR is an *operation* that takes zero or more *operands* as input arguments and returns zero or more *values*. To make data-flow information explicit, modern compiler IRs enforce static single assignment (SSA), assigning a unique name to the value returned by each operation. MLIR-style IRs—as used in this paper—also allow call-site-specific data to be associated with each call site through *attributes*, which are either explicitly written as a name-to-value dictionary or, if their meaning is clear from the context, as inline values.

The following two operations belong to the `arith` *dialect*, which groups operations that perform arithmetic on integers into their own namespace. The operation `arith.constant` takes no operands, has an integer attribute indicating the constant value to create, and returns a new value of type `i32`. The `arith.add` operation takes two `i32` operands and returns one `i32` result.

```
%0 = arith.constant 12 : i32
%1 = arith.add %0, %0 : i32
```

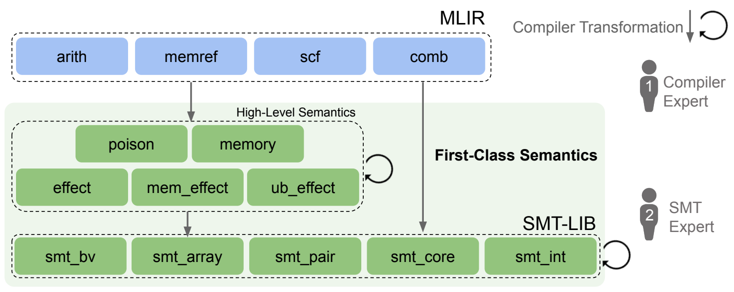


Fig. 2. We allow compiler developers to define high-level semantics using tools they are familiar with (lowerings) through multiple levels of semantic dialects. SMT experts then identify and implement the best lowering to low-level SMT abstractions, potentially exploring and offering multiple lowering strategies.

3.2 MLIR: A Framework for Designing Domain-Specific Compilers

MLIR is a production-grade compiler framework that provides a C++ implementation of data structures for instantiating an SSA-based IR consisting of operations, blocks, regions, and attributes, as introduced above. In addition, MLIR offers extensive tooling for analyzing and transforming these IRs, such as a pass manager and a rewrite engine. MLIR also offers core dialects that can be shared across different compilers, e.g., the `arith` dialect for arithmetic over n-bit integers, the `index` dialect for pointer arithmetic, and other dialects for modeling control flow, tensor computations, linear algebra, and more. MLIR's unique strength is its extensibility, allowing compiler developers to define their own IRs. MLIR also allows multiple dialects to be composed within a single translation unit, which makes it easy to build new dialects on a common, pre-existing framework. Although MLIR is primarily used for building software compilers, its CIRCT[2] subproject supports hardware design. A core abstraction used in CIRCT is the `comb` dialect for reasoning about combinatorial logic.

3.3 xDSL: a Python-Native SSA Compiler Framework

In our evaluation, we use the xDSL compiler framework [13], which makes it possible to rapidly prototype MLIR in Python. From the reader's perspective, both MLIR and xDSL are interchangeable in the sense that they are two implementations of the same IR abstractions, and in fact they can interoperate through textual IR files. In our work, we have used both MLIR and xDSL, choosing the most appropriate tooling for a given task.

4 Formal Semantics via a Collection of MLIR Dialects

We introduce a new set of dialects (Figure 2), rooted in the SMT-LIB language but extended with higher-level abstractions, to express MLIR dialects' semantics. By using dialects, we gain the property that defining semantics for an MLIR dialect can be done by writing a transformation from the program dialect to our semantic dialects. This allows the MLIR community to continue using tools and techniques they are already familiar with and also allows us to use MLIR as it was intended to be used: as the basis for avoiding the reinvention of compiler-like tools from scratch, such as a pass manager and a rewrite engine, both of which we use for manipulating semantics. We have created lower-level dialects that provide convenient MLIR access to SMT-LIB mechanisms, and also higher-level dialects for expressing language-specific semantics—such as memory—to create a separation of concerns between defining semantics and encoding them efficiently as SMT queries.

[2]https://circt.llvm.org

4.1 Interfacing with SMT Solvers using SMT-LIB Dialects

Our low-level dialects are based on the SMT-LIB v2 language [6], a mechanism for expressing SMT terms, formulas, and commands that interface with SMT solvers. We chose SMT-LIB because it enjoys broad support among modern SMT solvers; different solvers have different strengths, and we do not wish to tie our users to any single solver. However, while we base our work on SMT and SMT-LIB, these ideas could be applied to other formally specified languages and theories. SMT-LIB is based on S-expressions, making it easy to adapt to the MLIR dialect format. Each symbol in SMT-LIB is represented by a corresponding MLIR operation, and SMT-LIB sorts are translated to MLIR types. As SMT-LIB is composed of and can be extended with additional theories, which are sets of domain-specific symbols and axioms such as the bitvector and array theories, we define one dialect for each theory. The following SMT-LIB program checks if the integer formula $(x \cdot 2) \neq (x + x)$ is satisfiable. We show the same program in our `smt` and `smt_int` dialects (for the core and integer theory) alongside it.

```
(declare-const x Int)
(assert
  (distinct (* x 2) (+ x x)))
(check-sat)
```

```
%x = smt.declare_const : !smt_int.int
%two = smt_int.constant 2
%mul_two = smt_int.mul %x, %two
%add_twice = smt_int.add %x, %x
%eq = smt.distinct %mul_two, %add_twice : !smt_int.int
smt.assert %eq
smt.check_sat
```

As there is a clear one-to-one translation between SMT-LIB and our `smt` dialect, we can easily output SMT-LIB from an MLIR program using `smt`. It should also be easy to translate SMT-LIB to the `smt` dialect. Since this work was completed, the core `smt` dialect and associated theory dialects (such as `smt_int` and `smt_bv`) discussed here have been contributed to the upstream MLIR project as a single `smt` dialect.[3]

4.2 Efficient Encoding of High-Level Semantics to SMT Queries

Although SMT-LIB is an expressive language, it is also very low-level. There is a significant gap between the high-level semantics that compiler developers define and the low-level SMT-LIB language that SMT solvers understand. This gap is especially pronounced when encoding, for example, C or LLVM memory semantics [20], which require complex SMT-LIB representations of multiple memory blocks, their aliasing relationships, and the distinction between pointer and non-pointer values in memory. With only SMT-LIB as a target language for semantics, compiler developers would have to both do the work of defining the semantics for their dialect, and also the work of finding an efficient encoding of these semantics to SMT-LIB. This would both complicate the definition of semantics and make it harder to compose semantics between different dialects.

To bridge this abstraction gap, we define a set of high-level semantic dialects that are a better match for how compiler developers are likely to want to describe their dialects. These currently deal with memory semantics and undefined behavior, as these are the most complex parts of the MLIR dialects for which we have defined semantics. We believe this approach is generalizable to other language features and other domain-specific semantics. Defining semantics in these high-level dialects allows compiler developers to work in a language closer to the one they want to describe and allows SMT experts to provide efficient encodings of these semantics to SMT queries using specialized compilation passes (Figure 2). We believe that this separation of concerns between defining semantics and encoding them to SMT-LIB is necessary for our research program to succeed.

To illustrate what our high-level dialects look like, consider this example of the semantics of a program that writes the constant 42 to a memory location. The program triggers undefined behavior

[3] https://github.com/llvm/llvm-project/commit/de67293c093efddb9f9444b3a614695ad243355d

if the memory location is out of bounds. The world state is represented as an SSA value (with type `!effect.state`), and both the `mem_effect` and `ub_effect` dialects are used to interact with that state. Our high-level dialects make these semantics much more readable and understandable than their direct SMT-LIB encoding, as described for example by Lee et al. [20].

```
smt.declare_fun @write_42(%ptr: !mem_effect.ptr, %size: !smt.bv<64>, %index: !smt.bv<64>,
                          %state: !effect.state) -> !effect.state {
  %in_bounds = smt.bv.ult %index, %size : !smt.bv<64>
  %new_ptr = mem_effect.offset_ptr %ptr[%index]
  %val = smt.bv.constant 42 : !smt.bv<64>
  %state_noub = mem_effect.write %val, %state[%new_ptr] : !smt.bv<64>
  %state_ub = ub_effect.trigger %state
  %new_state = smt.ite %in_bounds, %state3_noub, %state3_ub : !smt.bv<64>
  smt.return %new_state : !effect.state
}
```

We describe below the semantics of our high-level semantic dialects using denotational semantics. + represents a disjoint union, × a Cartesian product, and ⟶ a map from one set to another, that is accessed with [], and updated with update($map, index, value$). Pairs are formed with (x, y), and their elements accessed with $_0$ and $_1$. inhabitant(T) is a value of type T. ⟦op⟧ represents the denotation of an MLIR op op as a mathematical function, and ⟦type⟧ the denotation of an MLIR type as a mathematical type (4). **poison**, **dead**, and **ub** are singleton types that represent the poison marker, a dead memory block, and an undefined behavior flag, respectively.

We currently define four high-level semantic dialects: the `poison` dialect (Figure 5) is used to add a poison marker to types, the `effect` dialect defines a type that is used to represent the world state, the `ub_effect` and `mem_effect` (Figure 6) dialects are used to represent undefined behavior and sequential memory effects, respectively, and the `memory` dialect (Figure 7) provides a lower-level abstraction for memory manipulation. We base our memory semantics on the work of Lee et al. [20], with a few simplifications as we currently do not handle writing pointers in memory or sub-byte memory accesses, which are not present in the MLIR programs we are considering.

$$\begin{aligned}
\texttt{byte} &:= 2^8 + \textbf{poison} & [\![\texttt{!poison.poison<T>}]\!] &:= T + \textbf{poison}\\
\texttt{bytes} &:= 2^{64} \longrightarrow \texttt{byte} & [\![\texttt{!effect.state}]\!] &:= \texttt{state}\\
\texttt{block} &:= 2^{64} \times \texttt{bytes} + \textbf{dead} & [\![\texttt{!mem\_effect.ptr}]\!] &:= \texttt{block\_id} \times 2^{64}\\
\texttt{block\_id} &:= \mathbb{N} & [\![\texttt{!memory.memory}]\!] &:= \texttt{memory}\\
\texttt{memory} &:= \mathbb{N} \longrightarrow \texttt{block} & [\![\texttt{!memory.block}]\!] &:= \texttt{block}\\
\texttt{state} &:= \texttt{memory} + \textbf{ub} & [\![\texttt{!memory.bytes}]\!] &:= \texttt{bytes}\\
& & [\![\texttt{!memory.block\_id}]\!] &:= \texttt{block\_id}
\end{aligned}$$

Fig. 4. Our high-level semantic dialects represent poison, memory, and undefined behavior.

$$\begin{gathered}
[\![\texttt{poison.from\_value}]\!](v) := v \qquad [\![\texttt{poison.poison}]\!] := \textbf{poison}\\
[\![\texttt{poison.is\_poison}]\!](p) := \texttt{true if } p = \textbf{poison} \texttt{ else false}\\
[\![\texttt{poison.to\_value}]\!](p) := \texttt{inhabitant}(T) \texttt{ if } p = \textbf{poison} \texttt{ else } p
\end{gathered}$$

Fig. 5. The `poison` dialect is used to explicitly mark values as poison.

4.3 Defining Formal Semantics Using Compiler Transformations

As we can encode semantics using the MLIR infrastructure, giving semantics to a dialect is a matter of defining a lowering (compilation pass) from the dialect to our semantic dialects. This fits nicely into the existing MLIR philosophy that a lowering from one dialect to another gives the semantics

$$
\begin{aligned}
\llbracket \texttt{ub\_effect.trigger} \rrbracket(s) &\coloneqq \mathbf{ub} \\
\llbracket \texttt{ub\_effect.to\_bool} \rrbracket(s) &\coloneqq \texttt{true if } s = \mathbf{ub} \texttt{ else false} \\
\llbracket \texttt{mem\_effect.offset\_ptr} \rrbracket(p, i) &\coloneqq (p_0, p_1 + i) \\
\llbracket \texttt{mem\_effect.read} \rrbracket(s, p) &\coloneqq (\mathbf{ub}, \mathbf{poison}) \quad \texttt{if } s = \mathbf{ub} \lor s[p_0] = \mathbf{dead} \lor p_1 \geq (s_0[p_0])_0 \\
&\coloneqq (s, ((s[p_0])_1[p_1])) \quad \texttt{otherwise} \\
\llbracket \texttt{mem\_effect.write} \rrbracket(v, s, p) &\coloneqq (\mathbf{ub}, \mathbf{poison}) \quad \texttt{if } s = \mathbf{ub} \lor s[p_0] = \mathbf{dead} \lor p_1 \geq (s_0[p_0])_0 \\
&\coloneqq \texttt{update}(s, p_0, ((s[p_0])_0, \texttt{update}((s[p_0])_1, p_1, v))) \quad \texttt{otherwise} \\
\llbracket \texttt{mem\_effect.alloc} \rrbracket(s, size) &\coloneqq \mathbf{ub} \quad \texttt{if } s = \mathbf{ub} \\
&\coloneqq \texttt{let } id \coloneqq \llbracket \texttt{memory.get\_fresh\_block\_id}(s) \rrbracket \texttt{ in} \\
&\quad \texttt{update}(s, id, (size, \{\mathbf{poison} \mid i \in \mathbb{N}\})) \quad \texttt{otherwise}
\end{aligned}
$$

Fig. 6. ub_effect and mem_effect define high-level operations to represent memory and undefined behavior.

$$
\begin{aligned}
\llbracket \texttt{memory.read\_bytes} \rrbracket(b, i) &\coloneqq b[i] \\
\llbracket \texttt{memory.write\_bytes} \rrbracket(v, b, i) &\coloneqq \texttt{update}(b, i, v) \\
\llbracket \texttt{memory.create\_bytes} \rrbracket &\coloneqq \{\mathbf{poison} \mid i \in 2^{64}\} \\
\llbracket \texttt{memory.create\_dead\_block} \rrbracket &\coloneqq \mathbf{dead} \\
\llbracket \texttt{memory.create\_live\_block} \rrbracket(s, b) &\coloneqq (s, b) \\
\llbracket \texttt{memory.is\_block\_live} \rrbracket(b) &\coloneqq \texttt{false if } b = \mathbf{dead} \texttt{ else true} \\
\llbracket \texttt{memory.unpack\_block} \rrbracket(b) &\coloneqq (0, \{\mathbf{poison} \mid i \in 2^{64}\}) \texttt{ if } b = \mathbf{dead} \texttt{ else } (b_0, b_1) \\
\llbracket \texttt{memory.get\_block} \rrbracket(m, i) &\coloneqq m[i] \\
\llbracket \texttt{memory.set\_block} \rrbracket(b, m, i) &\coloneqq \texttt{update}(m, i, b) \\
\llbracket \texttt{memory.get\_fresh\_block\_id} \rrbracket(m) &\coloneqq \texttt{min}(\{i \in \mathbb{N} | m[i] \neq \mathbf{dead}\})
\end{aligned}
$$

Fig. 7. The memory dialect provides a low-level abstraction to manipulate memory blocks.

of the source dialect. The key distinction is that our dialect has simple and well-defined semantics, compared to most MLIR dialects which often only have informal semantics with often poorly specified corner cases (the arith dialect only specified its overflow semantics two years after its creation). Additionally, while the llvm dialect, a popular compilation target in MLIR, has well-defined semantics, its semantics are quite hard to reason about as it has both poison and undef [21] values. Thus, compiling from a dialect without undef or poison semantics can be error-prone.

Our semantic dialects make concepts such as poison values explicit, using the poison dialect (Figure 5). Similar to how Alive2 [26] internally represents possibly-poison values, our poison type is compiled to a pair (defined using an SMT-LIB algebraic datatype) of a value and a Boolean poison flag. For instance, the following example shows how the arith dialect's addi operation is compiled into our semantic dialects.

```
%one = arith.constant 1 : i64
%r = arith.addi %x, %one : i64
```

```
%one = smt_bv.constant 1 : !smt_bv<64>
%x_value = poison.to_value %x : !smt_bv<64>
%x_poison = poison.is_poison %x : !smt.bool
%r_value = smt_bv.add %x_value, %one : !smt_bv<64>
%r_val = poison.from_value %r_value : !smt_bv<64>
%r = smt.ite %x_poison, %x, %r_val : !poison.poison<!smt_bv<64>>
```

Expressing semantics using the MLIR infrastructure allows users to define semantics using the same tools they use to define compilation passes and to use the same infrastructure to optimize the

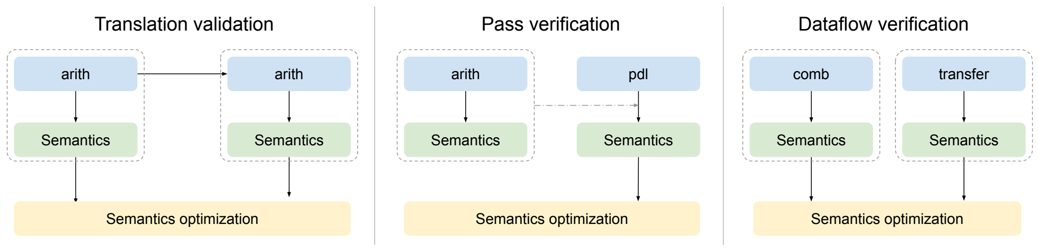


Fig. 9. Our semantics building blocks serve as the foundation for dialect-independent tools.

semantics. For most operations, this means that users can define their semantics using the MLIR peephole rewrite infrastructure, making the definitions familiar to MLIR users.

While our semantics are currently defined in Python (as we use xDSL, otherwise they would be defined in C++), we believe that defining them in a declarative language would enable new tools and workflows. For instance, one could imagine generating semantics documentation or reusing the same semantics in a theorem prover such as Lean [8]. In exploratory work, we tried out this idea and defined some of our semantics using the `pdl` (Pattern Descriptor Language) dialect, which allows the definition of rewrites as an MLIR program. However, we found that `pdl` was too limited to define the semantics of all operations we want to support, as it only supports DAG-to-DAG transformations (preventing us from compiling side-effectual operations) and has limited support for manipulating attributes and types (preventing us from handling operations such as `arith.constant`). We believe that either extending `pdl` or designing a more general dialect would be necessary to achieve a more declarative definition of semantics.

4.4 Optimizing SMT Queries Across Levels of Abstraction

As we encode semantics at multiple levels of abstraction, compiler experts and SMT experts can work together to define domain-specific optimizations at each level, reducing the complexity of the final SMT encoding and speeding up the verification process. Each of our semantic dialects defines constant folding and simple peephole rewrite rules to reduce the size of the SMT encoding and ease the process of debugging semantics when reading the IRs. By adding a common subexpression elimination pass and a set of domain-specific optimizations, the majority of our SMT queries for the `arith` dialect are reduced by half in terms of the number of operations. Similarly, many tests we wrote for our `memref` dialect semantics did not even need to be run through an SMT solver, as our simplification passes were able to prove simple examples (such as constant folding across memory operations) directly, while the original query was using around 200 `smt` operations. Previous work has shown that these simple peephole rewrite optimizations have great potential to reduce SMT solving times, as the subset of LLVM `instcombine` rules have been shown to speed up SMT queries when applied to SMT expressions [29].

As we define specialized semantic dialects, we can also define domain-specific optimizations that have more impact on the SMT encoding, as they are not always supported by SMT solvers. For instance, we define a compilation pass that removes algebraic datatypes from an SMT query, significantly improving the performance of Z3 solving time (Section 6.4). We also use state-of-the-art SMT encodings for memory semantics [20], which are not defined as an optimization, but as a compilation pass from our high-level `memory_effect` dialect to the `memory` dialect, which is then lowered to the SMT-LIB dialects.

5 Defining Verification Tools at the Framework Level

Figure 9 shows how semantics can be mixed and matched to create multiple dialect-independent compiler-adjacent formal verification tools. Given the open-ended nature of MLIR, a separation of concerns between defining semantics and building verification tools is essential. To demonstrate the applicability of our approach, we created three semantics-agnostic verification tools: a translation validation tool (Section 5.1), a peephole rewrite verification tool (Section 5.2), and a dataflow analysis verification tool (Section 5.3).

Beyond the tools we have created, our approach opens up broader possibilities for tools built on top of our semantic dialects. We could for instance create a program verification tool, to check that a program is free of undefined behavior or that it follows a specification. We could also try to define superoptimizers or rewrite synthesizers, like the work of Souper [35] or Hydra [30]. Another possible avenue is to generate other sections of a compiler, such as an interpreter or a constant folder, which are easy to get wrong due to corner cases in the semantics such as undefined behavior. Finally, we believe that generating documentation for the semantics of a dialect would be quite valuable, as the semantics of dialects are usually not well documented, or only informally specified.

5.1 Translation Validation

A translation validation tool takes a *source* and a *target* program and checks whether the target refines the source. In the presence of undefined behaviors, non-trivial refinements—compiler transformations that are legal in one direction, but that cannot be soundly reversed—are ubiquitous and can be difficult for humans to reason about. Translation validation can be used to gain formal confidence in a compiler's work without reasoning about the compiler's implementation. Alternatively, in combination with a fuzzer, translation validation can be used to look for compiler bugs. Our semantic dialects support and simplify the process of writing a translation-validation tool that works for multiple program dialects. Two programs are passed to the tool, which compiles them to the semantic dialects by using the provided lowerings, combines their lowered forms with a dialect-specific refinement relation, and inserts a check that the target program is a refinement of the source program for all possible inputs.

The refinement relation is split into two components: the state refinement relation, which checks a refinement of the final function state (consisting of the memory and an undefined behavior flag), and the function results refinement relation, which checks a refinement for each result of the function. The state refinement relation is always the same, and is the one defined by Lee et al. [20]. The function results refinement, however, is provided by the user for each type that appears in a function result, with defaults provided for common types such as integers and poison. In particular, the refinement might be from one dialect type to another in the case of a lowering.

5.2 Verifying Peephole Rewrites

Peephole optimizers are a common source of compiler bugs; for example, the Csmith paper reports that InstCombine, LLVM's primary peephole optimization pass, was its single buggiest component [39]. Verifying an entire peephole rewrite offers strong correctness guarantees for a given optimization, compared to translation validation, which only checks the correctness for a given input code. Several important MLIR passes rely on peephole rewrites and will consequently benefit from a tool for verifying them. A rewrite that performs lowering from a higher-level to a lower-level dialect is also an attractive target for formal verification. In particular, MLIR's use of specialized dialects tends to yield progressive lowerings that pass through multiple abstraction levels and, consequently, have numerous opportunities for bugs to occur.

Simple peephole rewrites consisting of a specific input IR snippet that must be rewritten to a specific output IR snippet, e.g., `MUL(x, 2)` $\rightarrow$ `ADD(x, x)`, can be verified easily by translating both sides of the rewrite to the SMT dialect as described in Section 5.1. However, when the rewrite involves literal constants, we would prefer to verify it for all possible values of the constant—translation validation is not helpful for this. Consider, for example, the rewrite `MUL(x, C)` $\rightarrow$

`SHL(x, log2(C))`, which is valid when `C` is a power of two and which leads to the following transformation on MLIR's `arith` dialect when `C=32`:

```
%C = arith.constant 32 : i32                →    %log2_C = arith.constant 5 : i32
%r = arith.muli %x, %C : i32                     %r = arith.shli %x, %log2_C : i32
```

We want to preserve the generality of the rewrite when translating to the `smt` dialect.

5.2.1 Peephole Rewrites in `pdl`*.* To automatically reason about the correctness of peephole rewrites, the rewrites should be written declaratively in a DSL that can generalize over IR properties such as constants and types. MLIR has a DSL called PDLL (Pattern Descriptor Language Language) and offers `pdl` (Pattern Descriptor Language), a high-level dialect that describes the transformation as introspectable compiler IR. While PDLL is meant to be the entry point for users, `pdl` is an MLIR dialect, which is well-suited for connecting to our semantic dialects.

```
pdl.pattern @MulToShift : benefit(0) {
  // Match an SSA value of type i32
  %type = pdl.type : i32
  %x = pdl.operand : %type

  // Match a power of two constant and the root multiplication
  %C_attr = pdl.attribute : %type
  %C_op = pdl.operation "arith.constant" {"value" = %C_attr} -> %type
  %C = pdl.result 0 of %C_op
  pdl.apply_native_constraint "is_power_of_two"(%C_attr)
  %mul_op = pdl.operation "arith.muli"(%x, %C) -> %type

  // Rewrite the multiplication
  pdl.rewrite %mul_op {
    // Compute log2(C) and create the corresponding constant
    %log2_C_attr = pdl.apply_native_rewrite "log2"(%C_attr)
    %log2_C_op = pdl.operation "arith.constant" {"value" = %log2_C_attr} -> %type
    %log2_C = pdl.result 0 of %log2_C_op

    // Replace the multiplication with a shift
    %shift_op = pdl.operation "arith.shli"(%x, %log2_C) -> %type
    pdl.replace %mul_op with %shift_op
}}
```

A `pdl`-based pattern is defined in the region of a `pdl.pattern` operation and uses `pdl` operations to match and constrain a rooted DAG of MLIR operations. The last operation in the region is a `pdl.rewrite`, which specifies new operations to create and how to replace the root operation with them. The preceding `pdl` pattern (with a simplified syntax) represents the rewrite `MUL(x, C)` $\longrightarrow$ `SHL(x, log2(C))`, where C is constrained to be a power of two.

The pattern matches on an expression tree with a multiplication at its root where the multiplication is composed of an unconstrained operand `%x` on the left-hand side, and a constant `%C` on the right-hand side that is constrained to be a power of two using so-called native constraints. The rewrite then replaces the multiplication with a shift operation by a constant representing the log base two of the original constant `C`.

5.2.2 Lowering `pdl` *Programs to the* `smt` *Dialect.* We lower `pdl` programs to the `smt` dialect in a single pass consisting of local rewrites. Consequently, we describe the lowering of each `pdl` operation to the `smt` dialect independent of other operations. While our tool fully supports poison values and undefined behavior, we simplify the presentation in this section by not describing this.

We translate `pdl.operand`, which matches a single SSA value, to an `smt.declare_const` operation that introduces a new value of the given type. Each MLIR type has a corresponding SMT sort. For instance, the i32 type is lowered to a 32-bit bitvector (ignoring poison). Similarly, we convert `pdl.attribute`, which matches attributes, to an `smt.declare_const` operation. While, in our examples, we translate both an operand and an attribute of type `i32` to a 32-bit bitvector, an `attribute` can have different semantics than an SSA value of the same type. In practice, because of poison semantics, an `i32` is encoded as a pair of a 32-bit bitvector and a boolean poison flag, while an `i32` attribute, which cannot be poison, is encoded simply as a 32-bit bitvector.

```
%op = pdl.operation "arith.muli"(%x, %y) -> %type        ⟶  %res = smt.bv.mul %x, %y
%res = pdl.result 0 of %op                                          : !smt.bv<32>
```

```
%op = pdl.operation "arith.constant"                     ⟶  %res = %attr
    {"value" = %attr} -> %type
%res = pdl.result 0 of %op
```

```
%op = pdl.replace %op1 with %op2                         ⟶  %r = smt.eq %res_op1, %res_op2
                                                             smt.assert %r
```

Fig. 11. pdl code referencing operations such as `arith.muli` and `arith.constant` lowers to native SMT operations or just an assignment, respectively. The `pdl.replace` operation is lowered to a refinement check, here an equality test, since we are omitting undefined behavior checks to simplify the presentation.

We translate `pdl.operation` and `pdl.result`, which match operations and their results to the corresponding semantics of the matched operation (Figure 11). For operations that take attributes, e.g., `arith.constant`, we expect the operation semantics to refer to the attribute as an SSA value, as we cannot determine statically the attribute value. Finally, `pdl.replace`, which erases an operation and replaces its results with the results of another operation, is lowered to a refinement check (provided by the user) for each result. Ignoring poison, the refinement is simply an equality check.

Our lowering of the `pdl` dialect to SMT consists of `pdl`-specific transformations that are generic over all dialects paired with the dialect-specific semantics for types (including refinements), attributes, and operations. All dialect-specific semantics are shared across our implementation of translation validation and `pdl`. This removes the risk of introducing bugs by duplicating information and allows us to build multiple tools from the same semantics.

5.2.3 *Handling* `pdl` *Native Rewrites and Constraints.*

To keep `pdl` concise while allowing for user-defined constraints and rewrites, `pdl` has two operations, `pdl.apply_native_constraint` and `pdl.apply_native_rewrite`, which allow users to call out arbitrary code during a rewrite. These two operations are essential to represent realistic rewrites, as they allow users to, for example, constrain attributes using user-defined code. In our example, `MUL(x, C) ⟶ SHL(x, log2(C))`, constraining `C` to be a power of two requires a native constraint, and the computation of the value of `log2(C)` requires a native rewrite.

To lower native constraints and rewrites to the `smt` dialect, we provide a mapping to their corresponding SMT encoding. We assume that the arbitrary code that the native operations perform is correct and corresponds to these SMT semantics. For instance, the lowering of a `pdl.apply_native_constraint` that checks that an integer is a power of two, and the lowering of a `pdl.apply_native_rewrite` that computes the logarithm base two both lowers to the semantic dialects encoding of these operations. While this requires the user to provide a correct semantics implementation of these operations, rewrites often reuse the same constraints and rewrites across multiple rewrites (Section 6.2).

5.2.4 *Enumerating Feasible Bitwidths.*

While our earlier examples of `pdl` rewrites fixed all types to 32-bit integers, most rewrites are valid for several or even arbitrary bitwidths. `pdl` has native support for expressing rewrites over a family of types by not constraining the type. However, a known limitation of SMT is that it cannot reason about bitvectors with unknown or parametric bitwidth. Hence, when translating from pdl to SMT, we first iteratively specialize the pattern for each combination of feasible bitwidth up to a configurable limit. Then, we individually verify each of the specialized instances of the rewrite. For certain operations, the set of valid types is constrained. For instance, if a rewrite matches an `arith.trunci` operation, which truncates an integer to a smaller bitwidth, the rewrite is only valid if the source bitwidth is larger than the target bitwidth. This condition must be checked before lowering the rewrite to the `smt` dialect, as the semantics of an `arith.trunci` cannot be expressed in the SMT type system if the output bitwidth is larger than the input bitwidth. To ensure these checks are performed in time, we lower the corresponding native constraints first and only synthesize the full SMT expression if the native constraints confirm that a valid lowering to the SMT types is possible.

5.2.5 Bitwidth-Independent Reasoning. Despite the limitation detailed above, Niemetz et al. [33] have demonstrated that it is sometimes feasible to prove properties on integers of parametric bitwidth. Their approach involves representing integers with two SMT integers: one for its value and another for its bitwidth. The bitwidth variable is then used to implement bitwise modular arithmetic for integers. Thus, for $x \in \mathbb{N}$, the result of an arithmetic operation, and $w \in \mathbb{N}^*$, its bitwidth, $x\%2^w$ calculates its bitwise value. Finally, we can prove that a property about this bitwise value holds true for all bitwidths by quantifying over w.

We implemented a second version of the `arith` semantics that supports this bitwidth-independent reasoning. As our `pdl` verification tool is not dependent on the particular semantics, we can use it with either version of the semantics. Each of the two versions has its own advantages: the bitwidth-independent version is more general and can prove a rewrite for all bitwidths at once instead of enumerating feasible bitwidths (which takes exponential time in the number of independent bitvectors in the `pdl` pattern). However, the bitwidth-independent version is in an undecidable fragment of SMT and, in practice, does not solve all of our queries.

5.3 Formal Verification of Dataflow Transfer Functions

Optimizing compilers use dataflow analyses to derive facts about SSA values that are valid for all program executions. Since these facts are then used to justify optimizations, an incorrect dataflow analysis will tend to make an optimization unsound, leading to miscompilations. To make a dataflow analysis for an IR, a dataflow transfer function must be implemented for each operation in the IR. Implementing sound, precise, and efficient transfer functions can be difficult, and in fact, both GCC and LLVM have had miscompilation bugs rooted in unsound analyses.[4,5] Additionally, having an automatic verification tool for dataflow analysis encourages compiler developers to be more courageous when writing highly precise transfer functions. In this section, we show how we can build a tool to verify dataflow analyses using our lowering-based semantics.

5.3.1 A Forward and a Backward Analysis for CIRCT. The `comb` dialect used by the CIRCT EDA suite employs a *known bits* analysis that attempts to prove that individual bits are always either zero or one. It represents an abstract value as a pair of bitvectors $(zeroes, ones)$, such that $zeroes_i = 1$ if the i-th bit is provably always zero, and $ones_i = 1$ if the i-th bit is provably always one. The top element of the per-bit semilattice, corresponding to the situation where a bit cannot be proved to always be either zero or one, is represented by $zeroes_i = 0$, $ones_i = 0$. There is no bottom element, and $zeroes_i = 1$, $ones_i = 1$ is an invalid representation. Given this representation, a maximally precise transfer function for the bitwise OR operation (in pseudocode) is:

[4] https://reviews.llvm.org/D60846
[5] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=54031

```
KbValue<W> KbOr(KbValue<W> lhs, KbValue<W> rhs) {
  return (lhs.zeroes & rhs.zeroes, lhs.ones | rhs.ones)
}
```

Known bits is a forward dataflow analysis, meaning that the analysis of an operation's result is computed from the analysis of its operands. To this, we added *demanded bits*, a backward analysis, meaning that the analysis computes properties of an operation's operands based on the properties of its result. A bit of an SSA value is not demanded if its value provably does not affect the ongoing computation. For example, when an integer-typed value is truncated (and has no other uses), its upper bits are not demanded. A bit is considered to be demanded if it cannot be proven to be not demanded. The representation for an abstract value in the demanded bits domain is simply a bitvector. (CIRCT does not explicitly provide a demanded bits analysis, but it does gain some of the benefits by optimizing the special case of a bitwise operation followed by a bit extraction. The bitwise operation can be removed when it does not modify the extracted bits.)

5.3.2 *First-Class MLIR Support for Transfer Functions.* We created the `transfer` MLIR dialect that can be lowered to C++ for inclusion in a compiler, and it can also be lowered to `smt` dialects to support mechanical reasoning. A transfer function is defined as a `func.func` operation that takes abstract values as input and outputs. Each abstract value is represented as a `!transfer.abs_value`, and we additionally define the `transfer.integer` type to manipulate integer values with unknown bitwidth that are used in our abstract values. The `transfer` dialect defines arithmetic and logical operations for bitvectors of parametric width, and it also supports operations that we have observed to be useful in practice for building transfer functions, such as counting leading zeroes and looping for a fixed number of iterations. This dialect is expressive enough to define numerous sound and precise transfer functions for the known bits and demanded bits abstract domains (Section 6.3).

This example shows the transfer functions for the bitwise OR operation for the known bits domain using the `transfer` dialect:

```
!int = !transfer.integer
!knownBits = !transfer.abs_value<[!int, !int]>

func.func @ORImpl(%arg0: !knownBits, %arg1: !knownBits) -> !knownBits {
    %arg0_0, %arg0_1 = transfer.unpack(%arg0) : !int, !int
    %arg1_0, %arg1_1 = transfer.unpack(%arg1) : !int, !int
    %result_0 = transfer.and(%arg0_0, %arg1_0) : !int
    %result_1 = transfer.or(%arg0_1, %arg1_1) : !int
    %result = transfer.make(%result_0, %result_1) : !int, !int
    func.return %result : !knownBits
}
```

After lowering transfer functions to C++ we can plug them into the existing MLIR dataflow framework, resulting in a complete dataflow analysis.

5.3.3 *Proving Properties of Transfer Functions.* The Galois connection describes the relationship between concrete and abstract values; we use it to prove the soundness and precision of transfer functions [10]. We can additionally check if a transfer function is maximally precise.

To reason about known bits, we start by defining a "wellFormed" predicate insisting that abstract values obey their representational invariant and also an "includes" predicate that is true when a concrete value x is a member of the concretization set of an abstract value a. Here are their definitions for the known bits domain:

$$\text{wellFormed}(a) \overset{\text{def}}{=} (\text{a.zeroes} \mathbin{\&} \text{a.ones}) = 0$$
$$\text{includes}(a, x) \overset{\text{def}}{=} ((\sim x \mid \text{a.zeros}) =\sim x) \wedge ((x \mid \text{a.ones}) = x)$$

Then, if $\mathcal{A}$ is the set of abstract values in the known bits domain and C is the set of concrete values, a transfer function is sound if:

$$\forall a, b \in \mathcal{A}, \forall x, y \in C, \text{wellFormed}(a) \land \text{wellFormed}(b) \land \text{includes}(a, x) \land \text{includes}(b, y)$$
$$\implies \text{includes}(\text{abstractOp}(a, b), \text{concreteOp}(x, y))$$

In other words, the result of applying a concrete operation—which comes from our lowering-based semantics—must always be contained within the result of the abstract operation. Once we implement this formula in our `smt` dialect, it can be generically applied to any known bits transfer function for a binary operation. We have implemented a similar formula to check precision; we omit it for brevity (and, in any case, many transfer functions of interest are not maximally precise).

To formally verify the soundness of demanded bits transfer functions, we define an "isSame" predicate that is true when two bitvectors x_1 and x_2 are indistinguishable from each other under a given demanded bit mask:

$$\text{isSame}(a, x_1, x_2) \overset{\text{def}}{=} (a \mathbin{\&} x_1) = (a \mathbin{\&} x_2)$$

Then, for all pairs of concrete bitvectors that are indistinguishable, we insist that the concrete operation has the same result:

$$\forall a \in \mathcal{A}, \forall x_1, x_2, y \in C,$$
$$\text{isSame}(\text{abstractOp}(a), x_1, x_2) \implies \text{isSame}(a, \text{concreteOp}(x_1, y), \text{concreteOp}(x_2, y))$$

This formula is for the first operand of a binary operation; its partner for the second operand is straightforward. And, again, we omit the formula for precision.

6 Evaluation

In Section 2 we made three claims. First: special-purpose dialects for expressing semantics are an effective way to formalize the meaning of MLIR dialects. Second: providing multiple levels of semantic dialects enables efficient SMT encodings. And third: formal verification tools for MLIR can be defined at the framework level. This section presents results from using our semantic dialect framework to make improvements to the MLIR ecosystem.

We created three dialect-agnostic tools that are built on top of our semantic dialects: a translation-validation tool (Section 6.1), a peephole rewrite verification tool (Section 6.2), and a transfer function verification tool (Section 6.3). We additionally show that domain-specific optimization passes on top of semantics expressed in our SMT dialect can lead to improved performance (Section 6.4). All three tools are generic over the dialect semantics and work with any dialect that can be lowered to the SMT dialect. Our performance tests are run on an AMD Ryzen 9 5950X 16-core CPU with 64 GB of DRAM using Z3 version 4.12.1, CIRCT commit `6133e783`, and xDSL version 0.22.0.

6.1 Translation Validation For Free

The `arith` dialect for computations running on CPUs and the `comb` dialect for combinatorial logic in hardware design are both used in industrial projects. We used translation validation, in combination with systematic and random testing, to look for bugs in three transformation passes, all of which are implemented in C++: `arith-expand`, which expands arithmetic to target-supported sizes, `arith-unsigned-when-equivalent`, which turns signed operations into unsigned ones when this is semantics-preserving, and `canonicalize`, which iteratively applies a set of peephole rewrites, called canonicalization patterns, and also runs a few global transformations such as dead code elimination and hoisting constants to the beginning of functions. To support this effort, we defined semantics for all 26 `arith` integer operations (we are not covering the floating point operations) as well as all 20 `comb` operations. In particular, we only defined semantics for control-flow free operations, though our work could easily be extended to cover control-flow operations without loops. Additionally, we defined semantics for the integer types and attributes offered by the `builtin` dialect, as they are used by both `arith` and `comb`.

Our semantics use both the SMT-LIB core and bitvector theory, as well as the algebraic datatypes as added in SMT-LIB 2.6. Integer values are encoded as a pair of a bitvector and a boolean, where the boolean indicates if the value is poison, a concept that is used in MLIR to indicate deferred undefined behavior. 14 out of the 20 `comb` operations can be directly mapped to a single SMT operation, while only 6 out of the 26 `arith` operations map to a single SMT operation. The remaining operations require either multiple SMT operations, a more complex mapping to SMT-LIB, or have to emit poison values for certain inputs. Yet, all operations can be expressed as lowerings to the SMT dialect, such that we obtain complete semantics for both dialects.

Since translation validation requires the compiler to execute before it can find bugs, we drove the passes in two ways. First, we exhaustively generated all `arith` and `comb` functions containing at most two operations, but using a restricted set of constants: −1, 0, 1, `INT_MAX`, and `INT_MIN`. This resulted in 443,106 MLIR functions. To this, we added 100,000 randomly generated functions, each containing up to 100 operations. We ran each of the resulting 1.6 million translation validation tests under a 32-second timeout; this took 8.5 hours using all cores on a 32-core machine (Table 1).

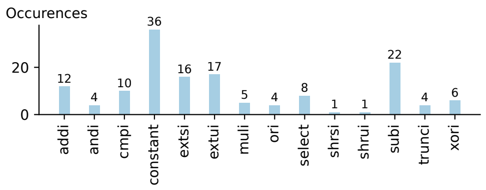
Pass	Ops	Tests	T/O	Time	Ops	Tests	T/O	Time
`canonicalize`	1-2	443 k	0.04%	5416 s	100	100 k	0.49%	3759 s
`arith-expand`	1-2	443 k	0.29%	6660 s	100	100 k	1.29%	8159 s
`arith-unsigned-when-equivalent`	1-2	443 k	0.00%	5302 s	100	100 k	0.01%	1478 s

Table 1. Summary of using bounded-exhaustive and randomized testing to drive translation validation of three MLIR transformation passes. T/O indicates how many of the tests timed out.

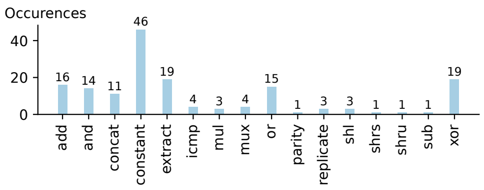
Name	Pattern
SelectI1Simplify	select(pred, x, y) ⟶ or(and(pred, x), and(xor(pred, 1), y)) for x, y ∈ `i1`
SelectAndCond	select(predA, select(predB, x, y), y) ⟶ select(and(predA, predB), x, y)
SelectAndNotCond	select(predA, select(predB, y, x), y) ⟶ select(and(predA, not(predB)), x, y)
SelectOrCond	select(predA, x, select(predB, x, y)) ⟶ select(or(predA, predB), x, y)
SelectOrNotCond	select(predA, x, select(predB, y, x)) ⟶ select(or(predA, not(predB)), x, y)

Table 2. Testing MLIR's `canonicalize` pass using our translation validator revealed these five miscompilation bugs, which have all been fixed.

This testing campaign resulted in the discovery of five bugs in rewrites used in the `canonicalize` pass, that we reported and that have been fixed in upstream MLIR (Table 2). All five of these bugs had the effect of making the code less defined by introducing poison values in cases where the original code was not poisonous. This is a consequence of a subtle rule where `arith.select` (a ternary operator like LLVM's `select` instruction or C and C++'s `?:` operator) blocks a poison value coming in on its not-selected operand. For example, given the `arith.select %pred, %x, %y : i1` operation, where all three operands are Booleans, the `SelectI1Simplify` would rewrite it to a combination of `arith.or`, `arith.and`, and `arith.xor` operations, creating straight-line code. This rewrite is incorrect, for example, when `%pred` is false, `%x` is poison, and `%y` is not poison. In the original pattern, the result would be equal to the value of `%y` (since `arith.select` blocks poison from its not-selected operand). However, all of `arith.or`, `arith.and`, and `arith.xor` produce poison output when either of their inputs is poison. Introducing new poison values is a dangerous miscompilation because subsequent passes can act on the poison values, producing further breakage.



(a) `arith` dialect



(b) `comb` dialect

Fig. 12. Operations from MLIR's `arith` and `comb` dialects are used (repeatedly) across our peephole rewrites demonstrating a need for semantics across a wide set of operations.

Category	**# rewrites**	**Description**
Mathematical operations	8	Operations on integer attributes, such as addition
Constants	3	Getting a constant with a specific type
Integer type width	2	Convert integer attributes to type with specific width
Comparison predicates	2	Inverting a comparison opcode (e.g. eq to ne)

Category	**# constraints**	**Description**
Equality to constants	4	Constraining an attribute to a given constant value
Equality of attributes	3	Checking if two attributes are equal or not
Integer type width	2	Checks on the bit-width of an integer type
Comparison predicates	2	Checking if an attribute is a comparison opcode

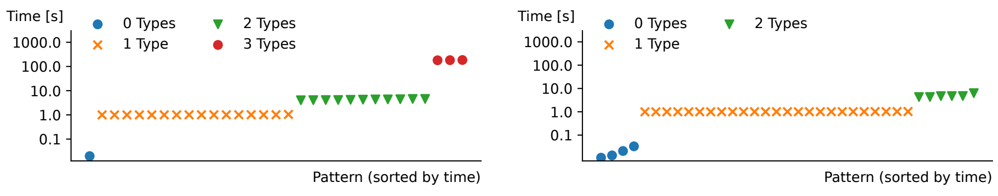
Table 3. Most native rewrites and patterns are not specific to either `arith` nor `comb`.

Bugs related to undefined behavior are tricky to catch using standard testing techniques, but they are easily detected by UB-aware translation validation tools. This testing campaign greatly increased our confidence in the correctness of these three MLIR passes. Moreover, translation validation is a foundation for other technologies such as superoptimization that we plan to explore.

6.2 Verified Peephole Rewrites for a Full MLIR Pass

In Section 6.1, we looked at MLIR transformations implemented in C++, where the best we can do is to check their work using translation validation. Next, we look at once-and-for-all formal verification (up to a maximum bitwidth) of transformations that are expressed declaratively. To do this, we used `pdl` to reimplement 31 `arith` peephole rewrites and 35 `comb` rewrites from their respective `canonicalize` passes. These rewrites cover a large variety of operations (Figure 12); constants are used in most rewrites, and otherwise they cover addition, subtraction, multiplication, division, shifts, and various bitwise operations. These peepholes account for most of the complexity and most of the opportunities for error in the canonicalizers. Since we have already formalized the operations from the `arith` and `comb` dialects, we could simply reuse those semantics here.

Our `pdl` patterns also rely on a set of native constraints and rewrites (Section 5.2.3), which we newly define to cover their corresponding uses in our patterns (Table 3). As upstream MLIR currently does not use `pdl` to optimize `arith` or `comb`, no native rewrites or constraints had been defined for our use case, which means that we had to define them ourselves. Interestingly, most of the constraints we defined can be reused across the `arith` and `comb` dialects: they are not specific to individual patterns. This is not surprising as most native rewrites are either simple mathematical operations such as addition or subtraction, conversion from integer attributes to integer types, or constant factories given a specific type. Similarly, most constraints are either checking equality to a constant or an attribute, or constraints on the bitwidth of integers.



(a) Rewrites for the `arith` dialect (b) Rewrites for the `comb` dialect

Fig. 13. Most patterns can be quickly verified up to 64 bits, except for a few outliers in the `arith.extsi` and `arith.trunci` canonicalizations, which involve more than three different integer types in the same pattern.

A particular weakness of SMT-based tools is that they generally cannot reason about bitvectors of parametric size. Thus, we follow the example of Alive [27] and individually verify every feasible bitwidth for each `pdl` pattern, up to a configurable maximum width, which we set here to 64. Although most patterns contain two or fewer integer types and can be verified using a few queries, the canonicalization patterns in `arith.extsi` and `arith.trunci` contain up to three different integer types, which results in a large number of queries and thus a verification time of around three minutes per pattern. Figure 13 gives more details.

Using our verifier, we proved correct all the `pdl` rewrite rules that we took from a version of MLIR that contained bug fixes for the five miscompilations we described in Section 6.1. We also ensured that we could detect buggy transformations, including the ones we reported. The correctness guarantee that we provide for the MLIR `canonicalize` pass is that its rewrites, as expressed by the `pdl` patterns we derived from the C++ implementation, are semantics-preserving up to 64 bits with respect to the semantics we defined for the `arith` and `comb` dialects. We also managed to prove correct some of these patterns for any bitwidth. For that, we used our arbitrary bitwidth semantics, which proves in less than a second most patterns, in particular 2 of the 3 patterns that take more than 3 minutes to prove correct with our fixed-bitwidth semantics.

MLIR already offers an execution engine for `pdl`. Also, the executable C++ specifications of native constraints and rewrites, which are needed to transition to our `pdl` patterns, are straightforward to implement. Thus, we have high confidence in the correctness of these patterns, and we believe that the MLIR community could increase the trustworthiness of its transformations by adopting our declarative `pdl` implementation.

6.3 Formally Verified Known and Demanded Bits Analysis for CIRCT

CIRCT's `comb` dialect has a known bits analysis supporting five operations (and, or, xor, mux, concat) implemented in C++. We implemented a new known bits analysis for `comb` using our `transfer` dialect; it additionally supports shl, mul, add, sub, extract, and cmp. We also created a demanded bits analysis for its bitwise operations, add, sub, extract, concat, and mux (Table 4). We lowered these transfer functions to our semantic dialects and then proved that—for bitwidths up to 64—they are sound with respect to our concrete semantics for `comb`. We also proved that eight out of the eleven transfer functions are maximally precise; mul and shl are not (as expected, as a maximally precise function for these operations would be likely to be too complex to be practical).

We empirically compared the precision of the CIRCT known bits analysis with ours by running both analyses over all of the `comb` code in two open-source RISC-V processors: Rocket [1] and

	AND/		ADD/						NE/
Operations	**OR**	**XOR**	**SUB**	**MUL**	**SHL**	**MUX**	**CONCAT**	**EXTRACT**	**EQ**
Known Bits - CIRCT	✓✓	✓✓	×	×	×	✓	✓✓	×	×
Known Bits - Ours	✓✓	✓✓	✓✓	✓	✓	✓✓	✓✓	✓✓	✓
Demanded Bits - Ours (new)	✓	✓✓	✓	×	×	✓✓	✓✓	✓✓	×

Table 4. Comparing static analyses from upstream CIRCT's comb dialect with our improved analyses. Transfer functions marked ✓ are implemented but not maximally precise. Transfer functions marked ✓✓ are implemented and also maximally precise.

			Total	Known Bits	Known Bits		Bits Not
	Class	**# Ops**	**bits**	**(Upstream)**	**(Ours)**	**Increase**	**Demanded**
Rocket	Small	28 812	180 249	30 640	40 620	32.6%	20 063
	Medium	31 272	199 815	33 374	43 418	30.1%	20 739
	Large	40 338	273 388	41 438	51 992	25.5%	24 080
BOOM	Small	81 325	559 150	77 669	103 727	33.6%	50 593
	Medium	116 797	718 087	81 931	115 483	41.0%	53 731
	Large	182 594	1 203 763	138 246	191 455	38.5%	94 913
	Mega	285 723	1 947 627	233 865	362 412	55.0%	169 989

Table 5. Our known bits analysis is consistently more precise (from 30.1% to 55%) than CIRCT's default analysis across the combinatorial logic of real-world RISC-V processors with up to 286 k comb operations. We also provide a demanded bits analysis that is not implemented in CIRCT.

BOOM [2]. Rocket is an in-order core, and BOOM is out-of-order; both are derived from Chisel [3] specifications, passing through Chisel's FIRTL IR, to CIRCT's RTL dialects: `comb`, `seq`, and `hw`. For the BOOM design, we instantiated four variants—small, medium, large, and mega—that share the same overall core design but differ in size and complexity, while for Rocket processor, we have three variants —small, medium and large.

Table 5 shows that, compared to CIRCT's analysis, we see a 30% to 55% increase in the number of bits that are statically known. For instance, our analysis statically determines that 362,412 out of 1,947,627 possible bits in the BOOM Mega design are known, whereas CIRCT's current analysis pass only determines 233,865 bits. We found that by supporting more operations, the analysis result could be improved. For the BOOM Small design, if we remove all newly added transfer functions but only replace transfer functions from the upstream with more precise ones, we only observe a 3.4% improvement in detected known bits. Our analysis is not only substantially more precise than the upstream one, but it has also been proved correct with respect to the Comb's concrete semantics. An advantage of declarative, formally verified transfer functions is that since correctness is assured, compiler developers can be more courageous in pursuing efficiency and precision.

To make use of our improved analysis results, we implemented 13 optimizations that exploit known and not-demanded bits. These are an optimization that replaces a value with a constant when every bit is either not-demanded or known, and also four optimizations for each of bitwise AND, OR, and XOR, replacing instructions with constants or with one of their operands, when this is safe to do. Table 6 shows how many times these optimizations fire during compilation. Figure 14 shows an example of an optimization that fired during the compilation of the Large BOOM processor. As a final validation step, we ensured that the synthesized RISC-V processors (in simulation) continued to correctly run the Dhrystone benchmarks.

Class	Const.	AND	OR	XOR
Small	121	76	9	2
Medium	124	77	11	2
Large	123	76	15	1

(a) Rocket

Class	Const.	AND	OR	XOR
Small	79	2	20	1
Medium	105	2	28	1
Large	88	1	30	1
Mega	99	1	36	1

(b) BOOM

Table 6. Number of times our dataflow-driven optimizations fire while compiling various processors.

```
func.func @unoptimized(%arg0: i17,
  %arg1: i1, %arg2: i53) -> i34 {
  %c0_i36 = hw.constant 0 : i36
  %0 = comb.concat %arg0, %c0_i36  : i17, i36
  %c0_i53 = hw.constant 0 : i53
  %1 = comb.mux %arg1, %0, %c0_i53 : i53
  %2 = comb.or bin %1, %arg2 : i53
  %3 = comb.extract %2 from 0 : i53 -> i34
  return %3 : i34
}
func.func @optimized(%arg0: i17, %arg1: i1,
  %arg2: i53) -> i34 {
  %0 = comb.extract %arg2 from 0 : i53 -> i34
  return %0 : i34
}
```

	Demanded Bits	Known Bits
%c0_i36	N/A	$(0...0)_{36}$
%0	$(1...1)_{53}$	$(?...?)_{17}(0...0)_{36}$
%c0_i53	N/A	$(0...0)_{53}$
%1	$(0...0)_{19}(1...1)_{34}$	$(?...?)_{17}(0...0)_{36}$
%2	$(0...0)_{19}(1...1)_{34}$	$(?...?)_{53}$
%3	$(1...1)_{34}$	$(?...?)_{34}$

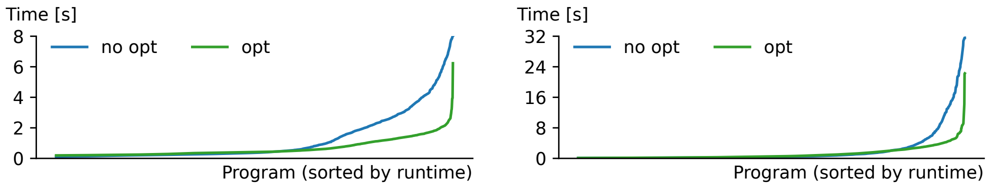
Fig. 14. An example of an optimization that fires during compilation of the Large BOOM processor. In the dataflow results on the right side of the figure, $(1...1)_{34}$ indicates a run of 34 one bits, $(?...?)_{53}$ indicates a run of 53 unknown bits. By demanded bits analysis, we know %3 only uses the low 34 bits from %2. By known bits analysis, we know the low 34 bits of %1 are zeros and they won't contribute to %2 because %2 is a bitwise OR. As a result, %2 can be replaced by %arg2.

6.4 Optimizing SMT Queries in the Presence of Poison Semantics

An advantage of expressing SMT expressions as a compiler IR is that compiler IRs are designed to be optimized. In Section 4.4, we mentioned that simple peephole optimizations, applied to our semantic dialects, typically eliminate about half of the code in them. In this section, we answer the question: Are there more interesting compiler optimizations that can be performed on our semantic dialects, that make solver queries measurably faster?

The default lowerings from our semantic dialects use tuples, defined as an algebraic datatype in SMT-LIB, to model functions that return multiple values, and also to model integer values that might be poison. However, SMT solvers do not uniformly perform well when presented with higher-level abstractions such as tuples. In particular, they can cause Z3 to slow down significantly. To retain the ergonomics of tuples, but avoid the performance penalty, we wrote an optimization pass that removes them when possible.

To evaluate the performance benefit of our datatype elimination pass, we evaluate it in the context of our translation validation tool and compare the solver time with and without our optimization pass. We evaluate our tool on the `arith-expand` pass, on all `arith` programs of at most two non-constant operations, and on 100 k programs with up to 100 operations. We use a timeout of 8 s for the small programs benchmark and 32 s for the larger one. Our datatype elimination pass is currently written in Python and takes 6684 s and 1542 s, an overhead that we expect to be drastically reduced once the pass is moved to C++.



(a) Translation validation with two operations (b) Translation validation with up to 100 operations

Fig. 15. Optimizing our semantic dialects can make SMT queries faster.

Our optimization pass produces clear benefits. Ignoring queries that time out, the total run time of our queries drops from 6050 s to 4853 s for the small programs benchmark, and from 3746 s to 2349 s for the large programs benchmark. This results in a speedup of 24.6% and 59.5%, respectively. A closer look at the runtime of queries that take more than 0.1 s to solve (Figure 15) shows that our optimized queries can consistently solve more queries given the same time budget. We see similar improvements in the number of timeouts, which our pass reduces from 0.26% to 0.22%, and from 2.29% to 1.50%, respectively.

Overall, expressing semantic dialect optimization as a normal compiler pass greatly reduces query time for our translation validation tool. While specific query optimizations may not be beneficial in all settings, domain-specific optimizations on the SMT queries are a powerful tool to make optimal use of an SMT solver with potentially complex performance behavior.

7 Related Work

Validating existing compiler transformations. Several projects have been developed to use formal methods to validate existing compiler transformations. MLIR-TV [4] defines a translation validation tool for MLIR using SMT solvers, specifically for machine learning dialects. This approach was inspired by the Alive [27] and Alive2 [26] projects, which defined both a translation validation tool and a peephole rewrite verification tool for LLVM, finding and preventing hundreds of bugs and still being actively developed and used in the LLVM ecosystem. Another project, Alive-mutate [11], found bugs by adding a mutation-based fuzzer to the existing Alive2 translation validator. Other automatic verification tools than SMT solvers have been used as well. For instance, Peggy [36] and LLVM-MD [37] use e-graphs for defining translation validation tools for LLVM. Despite all these projects defining translation validation tools, and most of these projects targeting LLVM, none of these projects share any significant code. In contrast, our semantics-agnostic tools are designed to work with any IR that is lowered to our `smt` dialect, allowing both reuse of the verification tools for new IRs, and allowing reuse of the semantics for new verification tools.

Automatically proving transformations and analyses correct. Multiple other projects have used automated theorem provers to verify the absence of miscompilations. Similar to our work on `pdl` and the `transfer` dialect, Cobalt [22] and Rhodium [23] define DSLs for defining both compiler transformations and analysis as local rewrites, and then use the Simplify automatic theorem prover to prove them correct. Similar to our translation of PDL to SMT-LIB, Crocus [38] translates Cranelift instruction selector rules into SMT-LIB to prove their correctness. In particular, their translation of arbitrary helper terms to SMT-LIB is similar to our handling of `pdl.apply_native_constraint`. Previous work also used verification of transformations to synthesize them, for instance in the Halide compiler [32]. Other work focused on proving and synthesizing dataflow analysis, such as VeRA [9], which targets a subset of the C++ language to prove a range analysis, and AMURTH [16], which can automatically synthesize sound and possibly most-precise transformers.

Interactive theorem provers for compiler verification. Another approach for proving compilers correct, much more costly in terms of engineering, but also more powerful, is to formally verify compilers using proof assistants. A well-known example is the CompCert [24, 25] project, which defined a C compiler in the Rocq theorem prover, which allowed them to prove the absence of miscompilations, and later on the correctness of the Verasco [15] static analyzer. Similarly, CakeML [17] defined a verified ML compiler in HOL4 that is proven correct. At the framework level, Vellvm [40] aims at providing a verified LLVM compiler, by defining executable semantics for LLVM IR in Rocq, which has been used to prove the correctness of a mem2reg [41] transformation. Lean-MLIR [8] also formalized the structural semantics of SSA peephole rewrites in Lean and implemented a few MLIR dialect semantics, though does not handle side-effects yet. We believe that our approach is complementary to the one of Lean-MLIR, as while Lean-MLIR is not restricted to SMT-LIB, it does not have yet the same level of automation as SMT solvers.

Intermediate Verification Languages. Existing tools such as Boogie [5], Why3 [31], and Viper [14] define intermediate languages that can efficiently be mapped to SMT solvers. Compared to these tools, our work is focused on integrating such intermediate languages into a compiler IR, and allowing extensibility of the intermediate language. We believe that in an open ecosystem like MLIR, which contains very specialized and domain-specific dialects, flexibility is needed in which abstractions are supported. The goal of our work is to provide infrastructure for domain-specific semantic dialects, and tools for optimizing and lowering them, allowing semantics IRs tailored to domain-specific dialects.

8 Conclusion

We are currently in an era of radical innovation in compiler IRs, driven by machine learning and other new application domains, and aided by technologies such as MLIR that make it easier to create and interoperate new specialized IRs. Our research is addressing fundamental infrastructure-level issues in MLIR in order to make it easier for compiler developers to specify a formal semantics for IRs that they create. We do this in an idiomatic MLIR style, where we have created multiple *semantic dialects*—in MLIR—for specifying the meaning of *program dialects*: the bread and butter dialects that represent application code. We aim to generate a change in the MLIR ecosystem where semantics are defined, as early as possible, by compiler developers. We have shown that we can repay the engineering effort devoted to this formal specification work by generating useful formal-methods-based tools that exploit the semantics to perform important tasks such as translation validation and proving the soundness of dataflow analysis transfer functions. In summary, our first-class verification dialects now bring a long-required formal semantics ecosystem to MLIR.

Data-Availability Statement

The artifact associated with this paper is available at DOI: 10.5281/zenodo.15231119 [12].

Acknowledgments

We thank the MLIR and CIRCT communities for their support and feedback. We especially thank Martin Erhart, Maksim Levental, Bea Healy, Mike Urbach, Will Dietz, and Leon Hielscher for the development and integration of the `smt` dialect into the CIRCT and MLIR projects. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/W007940/1, as well as the European Union's Horizon EUROPE research and innovation program under grant agreement no. 101070375 (CONVOLVE), This material is based upon work supported by the National Science Foundation under Grant No. 1955688.

References

[1] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The Rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), 6–2.

[2] Krste Asanovic, David A Patterson, and Christopher Celio. 2015. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. *University of California at Berkeley Berkeley United States, Tech. Rep* (2015).

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*. 1216–1225.

[4] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *International Conference on Computer Aided Verification*. Springer, 386–407.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2024. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org.

[7] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, Vol. 13. 14.

[8] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting in SSA Compiler IRs. *arXiv preprint arXiv:2407.03685* (2024).

[9] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a verified range analysis for JavaScript JITs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 135–150.

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 238–252.

[11] Yuyou Fan and John Regehr. 2024. High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 349–358.

[12] Mathieu Fehr. 2025. First-Class Verification Dialects for MLIR - PLDI 2025 Artifact. doi:10.5281/zenodo.15231119

[13] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Lücke, Théo Degioanni, Michel Steuwer, and Tobias Grosser. 2023. Sidekick compilation with xDSL. *arXiv preprint arXiv:2311.07422* (2023).

[14] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *European symposium on programming*. Springer, 125–128.

[15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified C static analyzer. *Acm Sigplan Notices* 50, 1 (2015), 247–259.

[16] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing abstract transformers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 171 (oct 2022), 29 pages.

[17] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.

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

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[20] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P Lopes. 2021. An SMT encoding of LLVM's memory model for bounded translation validation. In *Computer Aided Verification: 33rd International Conference, CAV 2021*. Springer, 752–776.

[21] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. 2017. Taming undefined behavior in LLVM. *ACM SIGPLAN Notices* 52, 6 (2017), 633–647.

[22] Sorin Lerner, Todd Millstein, and Craig Chambers. 2003. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 220–231.

[23] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. *ACM SIGPLAN Notices* 40, 1 (2005), 364–377.

[24] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

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

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

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

[28] Nicholas D Matsakis and Felix S Klock. 2014. The Rust Language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[29] Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT Solving via Compiler Optimization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1177–1189.

[30] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing peephole optimizations with program synthesis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 725–753.

[31] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*. Springer, 41–62.

[32] Julie L Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and improving halide's term rewriting system with program synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[33] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. 2019. Towards Bit-Width-Independent Proofs in SMT Solvers. In *Automated Deduction – CADE 27*, Pascal Fontaine (Ed.). Springer International Publishing, Cham, 366–384.

[34] The Swift Project. 2024. Swift Intermediate Language (SIL). https://github.com/swiftlang/swift/blob/main/docs/SIL.rst.

[35] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422* (2017).

[36] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 737–742.

[37] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 295–305.

[38] Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. 2024. Lightweight, modular verification for webassembly-to-native instruction selection. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 231–248.

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. doi:10.1145/1993498.1993532 http://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf.

[40] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 427–440.

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

Received 2024-11-14; accepted 2025-03-06