Ratte: Fuzzing for Miscompilations in Multi-Level Compilers Using Composable Semantics

Pingshi Yu
Imperial College London
London, UK

Nicolas Wu
Imperial College London
London, UK

Alastair F. Donaldson
Imperial College London
London, UK

Abstract

Multi-level intermediate representation (MLIR) is a rapidly growing compiler framework, with its defining feature being an ecosystem of modular language fragments called *dialects*. Specifying dialect semantics and validating dialect implementations presents novel challenges, as existing techniques do not cater for the modularity and composability required by MLIR. We present Ratte,[1] a framework for specifying *composable* dialect semantics and *modular* dialect fuzzers. We introduce a novel technique for the development of semantics and fuzzers for MLIR dialects, enabling a harmonious cycle where the fuzzer validates the semantics via test-case generation, whilst at the same time the semantics allow the generation of high-quality test cases that are free from undefined behaviour. The composability of semantics and fuzzers allows generators to be cheaply derived to test combinations of dialects. We have used Ratte to find 6 previously-unknown miscompilation bugs in the production MLIR implementation. To our knowledge, Ratte is the first MLIR fuzzer capable of finding such bugs. Our work identified several aspects of the MLIR specification that were unclear, for which we proposed fixes that were adopted. Our technique provides composable reference interpreters for important MLIR dialects, validated against the production implementation, which can be used in future compiler development and testing research.

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

Keywords: Compiler testing; differential testing; fuzzing; composable semantics; MLIR

ACM Reference Format:
Pingshi Yu, Nicolas Wu, and Alastair F. Donaldson. 2025. Ratte: Fuzzing for Miscompilations in Multi-Level Compilers Using Composable Semantics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716270

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716270

1 Introduction

Multi-level intermediate representation (MLIR) [14] is a relatively new but rapidly growing framework to reduce the cost of developing compilers, and already has many influential users [1, 14, 36, 37, 39]. MLIR helps compiler developers by providing a framework for defining reusable modular and composable fragments of intermediate representations (IRs) known as *dialects*. Each dialect contains syntax for a particular aspect of a programming language (e.g. control flow, memory accesses, floating point arithmetic), or a language feature specific to a particular domain (such as machine learning, hardware design or parallel programming) [26]. The library of composable dialects reduces development work for new programming languages by making it cheaper to define intermediate layers in compilation. As a result, MLIR-based compilers often introduce many abstraction levels to maximise the benefit of the dialect-specific optimisations available within MLIR [37–39].

Due to the increasing adoption of MLIR in a range of important domains, it is critical that MLIR-based compilers are thoroughly validated to ensure they are reliable, since bugs in MLIR can impact all of its users. However, compiler validation for this open framework of IRs and transformations leads to several new challenges:

Challenge 1: Lack of reference dialect semantics. Users expect compilers to do their job correctly. This means they translate the source code into *semantically equivalent* target code. The translation can be bug-prone, especially in the optimisation and code-generation stages. A class of bugs called *miscompilations*, are when the compiler silently generates wrong code, and are particularly difficult to detect.

To validate semantic correctness of compilers, there must first *exist* unambiguous semantics against which correctness can be checked. However, MLIR does not yet have clear semantics for individual dialects, nor for dialect compositions. The existing documentation for MLIR is the basis for current implementations, and it provides only an English-language account of dialect semantics. This leaves room for interpretation ambiguities and potential for unspecified edge cases.

The issue extends beyond the clarity of single dialect semantics. Since dialect syntax are designed to be composable, their semantics should also be composable. A novel challenge that MLIR presents is that the semantics of dialects

[1] Named in honour of Mimi, the first author's German pet rodent.

can *interact* when composed, and existing documentation on the semantics of dialect interactions is even less well-specified than that of individual dialects. This is a problem because MLIR-based compilers almost always use IR consisting of multiple dialects [37–39], and thus understanding the semantics of dialect compositions is important.

Challenge 2: Generating effective test programs. Even if clear semantics are available, a compiler still needs to be thoroughly validated against the semantics. *Differential testing* [25] is a standard approach for validating compiler correctness. It uses either multiple compiler implementations or multiple configurations of a single compiler (e.g. at different optimisation levels) to compile the same program, followed by executing and comparing the outputs of the resulting executables. Any *unexpected* output differences indicate that at least one of the compilers or compiler configurations has not compiled the program correctly.

However, putting differential testing into practice for MLIR is challenging for two main reasons. First, it is unlikely that multiple implementations of MLIR will be available for cross-checking, as MLIR is a unifying compiler framework intended to be reused. Applying differential testing across different optimisation levels will not reliably find errors in *lowering passes*, as all compilation paths go through the same lowering steps. Secondly, effective differential testing requires a large number of high quality inputs—diverse programs that produce well-defined results. Such input programs are non-trivial to generate. This is already a challenge for traditional programming languages [43], and the open ecosystem of MLIR only adds to the difficulty. The MLIR framework covers a large and ever-growing family of dialects. Each abstraction can introduce new syntactic and semantic restrictions that program generators must account for to generate programs suitable for differential testing.

Challenge 3: Scaling compiler validation to dialect combinations. Whilst it would be possible in principle to build a custom program generator for any MLIR dialect combination of interest, treating each combination as an individual language, such an approach would be very costly in terms of human effort, since their shared features mean many domain-specific generation techniques and analyses [18, 19, 32, 43] would have to be repeated for each dialect combination.

Sharing fuzzer code for different languages is an open problem in compiler fuzzing, where existing fuzzers are tightly coupled to one language [43] (or family of languages in similar domains [18]), and not easily adaptable to other languages [5]. Since MLIR is a unifying composable IR shared by different compilers, composably fuzzing the MLIR framework is both a challenge and an exciting opportunity to enable code-sharing between fuzzers. This requires a general framework allowing generators, analyses and test oracles to be defined in a *modular* fashion for dialects, and reused in program generators for dialect combinations.

Our work. To address the above challenges, we present a general method for specifying both semantics of dialects through composable interpreters, and composable generators (fuzzers) that are enhanced by utilising semantic information *during* program generation. Our method is inspired by the previous approach taken by YARPGen [18] for C-like languages, where a fuzzer is co-developed with an interpreter. The interpreter is executed in-step with program generation to ensure that the values of variables at all program points are known. Our method generalises the YARPGen approach to composable languages and allows for different analyses to be written for the language and used by the fuzzer during generation. Like the initial YARPGen approach [18], our approach is limited to generation of loop-free programs, which makes it possible to accurately track values of program variables during generation. Despite this limitation, we still test compilation of certain MLIR looping constructs by generating higher-level operations that are *lowered* into loops.

We enable a harmonious cycle for developing validated semantics for dialects, continuously cross-checking the semantics against existing compiler implementations using our fuzzer, in combination with static unit tests. Using the semantics, we create the first generators of deterministic, well-defined programs for MLIR-based compilers, to be used in end-to-end testing. Importantly, this allows deeper bugs, such as miscompilations, to be detected, which is not possible with the current state-of-the-art MLIR testing tools [34, 41].

Our framework handles the challenges of composable semantics and fuzzing by leveraging techniques such as effect systems [12, 42] for addressing the expression problem [40] when describing compositional semantics for programming languages, and QuickCheck style property-based testing libraries [7] for writing composable program generators. We have implemented our ideas in a practical tool, called Ratte.

We demonstrate the effectiveness of Ratte on key MLIR dialects, finding 8 previously-unknown bugs during end-to-end testing of dialect implementations—of which 4 have been confirmed and 3 fixed. In particular, 6 out of 8 bugs are miscompilations, which are difficult to detect and are beyond the scope of existing works on MLIR compiler testing. During the development of the framework, we also contributed to the MLIR specification by identifying ambiguities and inconsistencies in the documentation, which led to clarifying changes to the specification.

Our main contributions are as follows:

- A framework for describing MLIR dialect semantics modularly via composable interpreters (Section 3.2).
- A general approach for constructing modular and composable fuzzers for MLIR, using semantic information to allow the generation of undefined-behaviour free programs (Section 3.3).
- The first fuzzers for generating well-defined MLIR programs of several key MLIR dialects (Section 3.5).

- Eight bugs found within the MLIR framework, most of which are miscompilations, which are difficult to find with existing MLIR testing techniques (Section 4.1).
- Validated and modular reference semantics for a subset of MLIR dialects, and improvements to the MLIR specification, arising from our work on modular MLIR semantics (Section 4.3).

2 Background

MLIR is a family of IR components. New components are definable by the user, and each component is an instance of the foundational structure, following a basic syntax, presented in Figure 1. The syntax of Figure 1 is in a one-to-one correspondence with the "generic IR format" within the official documentation [27].

At its core, MLIR is a *single static assignment* (SSA) IR with the addition of `Regions`, which are a convenient abstraction to encode scoping directly in the IR. A `Region` is a piece of MLIR program. A regions allows accesses to variables defined within the region, and *potentially* to the variables of parent regions.

Like traditional SSA, an MLIR `Operation` has a name, `Operands`, `Results`, and compile time information attached, called `Attributes`. In addition to traditional SSA, however, MLIR `Operations` can also contain `Regions`.

Programming constructs can be modelled as `Operation` instances using the name, operands, attached `Regions`, and storing static information as `Attributes`. A collection of related operations on the same domain is called a *dialect*.

For example, in Figure 2, `arith.constant` is an operation that has no operands, one result, and an attribute containing a statically-known value (−1). Another example is `func.return`, in the last line, which is an operation that has one operand, no results, and no attributes attached. The two occurrences of `func.func` are examples of operations that have regions. In each case, the operation takes no operands and yields no results, but contains attributes storing the function type, its name ("`main`" and "`one`"), and a single region, contained within the "`{ }`" braces.

Dialects can be considered syntaxes, and are composable in the sense that multiple instances of dialects can exist in the same IR. Although an arbitrary IR (potentially using operations from multiple dialects) is valid syntactically, it is not necessarily the case that such an arbitrary IR has meaningful semantics.

To check the IR for validity, apply optimisations, and translate them down to executable code, MLIR provides an ecosystem of modular *passes*. These passes perform tasks in a modular way. For this paper, we consider them broadly classifiable into two categories: 1. *lowering* passes, which transform dialects of higher abstractions to lower abstractions, and 2. *optimisation* passes, which do not change the abstraction level, but either perform optimisations or compute additional information for other passes.

⟨*operation*⟩ ::= ⟨*results$*⟩? '"'⟨*id*⟩'"' '(' ⟨*operands*⟩? ')' ⟨*successors*⟩? ⟨*regions*⟩? ⟨*attributes*⟩? ':' ⟨*op-type*⟩

⟨*id*⟩ ::= [A-Za-z0-9]+

⟨*type*⟩ ::= ⟨*id*⟩

⟨*result*⟩ ::= '%'⟨*id*⟩

⟨*results*⟩ ::= ⟨*result*⟩ (',' ⟨*result*⟩)*

⟨*results$*⟩ ::= ⟨*results*⟩ '='

⟨*operand*⟩ ::= ⟨*result*⟩

⟨*operands*⟩ ::= ⟨*operand*⟩ (',' ⟨*operand*⟩)*

⟨*region*⟩ ::= '{' ⟨*entry-block*⟩? ⟨*block*⟩* '}'

⟨*regions*⟩ ::= ⟨*region*⟩ (',' ⟨*region*⟩)*

⟨*block-label*⟩ ::= '^'⟨*id*⟩

⟨*block-args*⟩ ::= '(' ⟨*result*⟩ ':' ⟨*type*⟩ (, ⟨*result*⟩ ':' ⟨*type*⟩)* ')'

⟨*block*⟩ ::= ⟨*block-label*⟩ ⟨*block-args*⟩? ⟨*operation*⟩+

⟨*successor*⟩ ::= ⟨*block-label*⟩

⟨*successors*⟩ ::= '[' ⟨*successor*⟩ (',' ⟨*successor*⟩)* ']'

⟨*op-type*⟩ ::= '(' ⟨*type*⟩, (, ⟨*type*⟩)* ')' '->' '(' ⟨*type*⟩, (, ⟨*type*⟩)* ')'

⟨*attribute-value*⟩ ::= dialect-attribute | builtin-attribute

⟨*attribute*⟩ ::= ⟨*id*⟩ '=' ⟨*attribute-value*⟩

⟨*attributes*⟩ ::= '{' ⟨*attribute*⟩ (',' ⟨*attribute*⟩)* '}'

Figure 1. Grammar that forms the basis of Ratte's representation of MLIR syntax tree, presented in extended Backus–Naur form. In a one-to-one correspondence with the MLIR generic representation [27].

MLIR passes can be composed to form a *pipeline*. Pipelines take IRs (dialect combinations) from high abstraction levels into a lower, executable abstraction level like LLVM, and thus can be seen as implicitly giving *semantics* to dialect combinations (when a valid pipeline exists). Passes have preconditions for their correct application (e.g. a statically and dynamically well-defined IR).

3 Methodology

Fundamentally, Ratte is a framework for the simultaneous specification of MLIR semantics, and generators for executable MLIR programs. Ratte-generated programs produce a well-defined output, and Ratte uses the semantics information to ensure this, by satisfy the static and dynamic validity constraints of individual dialects.

```
func.func @main() {
    %n1 = arith.constant -1 : i1
    %0 = call @one() : () -> i1
    %low, %high = arith.mulsi_extended %0, %n1 : i1
    vector.print %low : i1
    vector.print %high : i1
    return
}
func.func @one() -> i1 {
    %n1 = arith.constant -1 : i1
    return %n1 : i1
}
```

Figure 2. Example MLIR program using a combination of `func`, `arith`, and `vector` dialects. This Ratte-discovered program that computes -1×-1 revealed a miscompilation in the production compiler, which evaluated to -1 rather than 1 due to its special treatment of 1-bit integers types (`i1`).

Ratte's goal is to test the correctness of MLIR optimisations and lowerings passes by generating inputs for the MLIR framework (MLIR programs) and using differential testing to detect bugs. For programs to be usable for this purpose, in addition to being syntactically correct according to Figure 1, they must also respect the static constraints of the language and must not trigger any dynamic undefined behaviours. Altogether these constraints ensure a correct implementation will successfully compile the program, triggering passes throughout the compilation stack, and yield an output that can be used for differential testing.

As mentioned in Section 2, MLIR is an open ecosystem of composable dialects. Each dialect introduces operations that cover some abstraction domain and can impose constraints on which programs have well-defined semantics. Since compiler engineers using MLIR can also introduce new dialects, it is infeasible to create a fuzzer for the entirety of MLIR. Therefore, Ratte is an extensible and composable framework that allows generators of programs with well-defined semantics to be defined on a per-dialect basis. Generators for multiple dialects can then be used together for the dialect combination that the user requires.

For differential testing, there are a host of behaviours that are *undesirable*, which we define here as one of two things. First, *explicit undefined behaviours*, which make any compilation trivially correct. Second, behaviours that make it challenging to work with a differential testing oracle, such as non-deterministic behaviours, compile-time rejections or runtime crashes.[2]

[2]In principle it is possible to use differential testing to check that two compilers produce comparable error messages, or generate code that produces comparable runtime error messages when crashes occur during execution, but this is not the focus of our work.

To avoid generating these undesirable programs, Ratte keeps track of semantic information evaluated on the program during generation. Figure 4 shows some examples of undesirable behaviours introduced by various dialects, and the semantic information required to avoid generating programs that trigger the undefined behaviours.

Whilst it is possible to apply rejection sampling based on the grammar of MLIR, by generating ASTs of MLIR programs and rejecting those that contain undesirable behaviours, the process would be very inefficient. This is because a vanishingly small percentage of programs following the grammar would be free from such behaviours, resulting in a large amount of time wasted on generation.

An overview of Ratte is shown in Figure 3, and we refer back to components of the figure throughout this section. Ratte constructs programs incrementally, alternating between generation and semantic evaluation/analysis. Incremental semantics (Section 3.1) is used in the generation loop to efficiently evaluate a partially generated program, yielding information that will guide future generation choices. In order to evaluate the semantics in a modular way, the expression problem [40] needs to be addressed. Ratte uses *effect systems* [31] for this purpose and embeds the MLIR syntax using an effects-based AST for modular semantic evaluation and analysis (Section 3.2). Generators rely on the semantics of the partial program to inform future choices, in order to avoid making choices that lead to programs with undesirable behaviours (Section 3.3). At the end of generation, Ratte also appends code that produces output by printint the values of certain program variables, which allows generated programs to be used to detect bugs via differential testing (Section 3.4).

3.1 Generation Principles

As mentioned before, Ratte incrementally generates programs to avoid rejection sampling whenever possible, evaluating partial programs during generation to derive information for future generation steps. Each step randomly *extends* the partial program, and keeps the evaluated information up to date with the extended partial program. For efficiency, evaluating partial programs should also be incremental, in that evaluating an extension to a partial program should not require re-evaluating the entire partial program.

Similar design choices are also made in other program generators such as YARPGen [18] and Csmith [43], which use this observation implicitly. They generate incrementally as much as possible, falling back to rejection sampling for instances where the desired property is not straightforwardly incremental, like undefined-behaviour-free loops.

To make the concepts discussed above concrete, we give definitions for partial programs (what Ratte processes during generation), their extensions (how Ratte modifies programs), and evaluations that can be defined on partial programs (information used to decide on next generation steps).

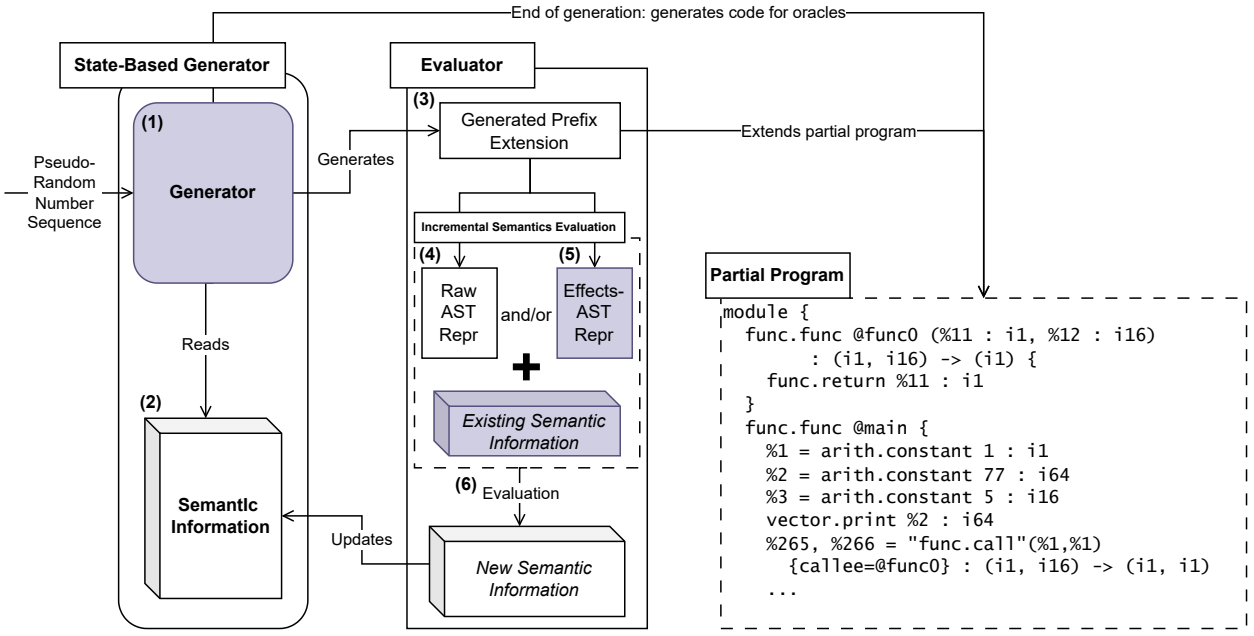

Figure 3. Overview of the Ratte framework. The generator (1) reads the semantics information (2) and generates an extension to the program. This extension (3) is used to update both the generated partial program and the semantic store. The semantic update step either uses the generated extension as is (4), if updating dialect-agnostic semantics, or first converts the extension into an effects-AST representation (5), if updating dialect-specific semantics, before evaluating the semantics update (6). The components to be updated for extending Ratte to new dialects are in purple: a) new generators for the new dialect, b) new effects-AST definitions, and c) modular semantics definitions for the new dialect.

```
%x = op1 %a1, %a2, ..., %an : t1
%x = op2 %b1, %b2, ..., %bn : t2
```
Requires: fresh value name generator

```
%0 = arith.constant 3 : i64
%1 = arith.constant 7 : i32
%2 = arith.addi %0, %1 : i32
```
Requires: typing information for values

```
%0 = arith.constant 0 : i64
%1 = arith.constant 1 : i64
%n = arith.divsi %1, %0 : i64
```
Requires: concrete interpretation for values

```
// %0 : tensor<3x3xi64>
%1 = arith.constant 9 : index
%2 = tensor.extract %0[%1] : tensor<3x3xi64>
```
Requires concrete interpretation for values

Figure 4. Examples of undesirable behaviours in MLIR dialects. From top to bottom, 1: compile-error (reuse of ID within a scope); 2: compile-error (mismatched types); 3: undefined behaviour (division by zero); 4: runtime error/undefined behaviour (out-of-bounds access).

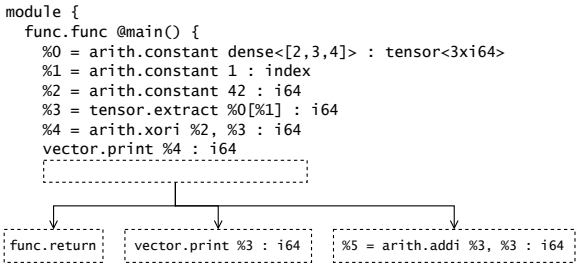

Figure 5. An example program prefix, along with some possible extensions to the prefix (in dashed boxes).

Partial Programs. Incremental fuzzers, such as Ratte or YARPGen [18] construct programs piece by piece. Figure 5 is an example of a partially generated program, and some possible operations that can be used to extend the program.

To formally describe the generation process above, we introduce the concept of *prefixes* and *extensions*. A prefix of an MLIR AST (following the grammar of Figure 1) is a subtree obtained from some *partial* depth-first traversal of the AST—treating operations without regions as leaves, and others operations (operations with regions) as nodes.

Definition 3.1 (Prefixes)**.** Given a valid MLIR program P with syntax tree T_P, a depth-first traversal visiting the operations in T_P in order $[o_1, o_2, \ldots, o_n]$, a *prefix* of P is a sub-tree of T_P containing only the operations $[o_1, \ldots, o_m]$, $m \leq n$. Let K denote the set of all prefixes, defined as:

$$K = \{P' \mid P' \text{ is a prefix of some program } P\}$$

Definition 3.2 (Prefix Extensions)**.** For an MLIR prefix P derived from the operation ordering $[o_1, \ldots, o_m]$, its prefix extensions P^+ are defined as

$$P^+ = \{P' \in K \mid \exists e \in \text{Op} \,.\, P' \text{ has ordering} [o_1, \ldots, o_m, e]\}$$

We say P is *extended* by e if some $P' \in P^+$ has ordering $[o_1, \ldots, o_m, e]$.

That is, for a given prefix P, the set of extensions P^+ are prefixes that contain all of the operations in P in the same order, with one extra operation e, such that they are still valid prefixes for *some* MLIR program.

A prefix is not expected to be a complete program: stopping the depth-first traversal halfway through a region would mean the program fails to satisfy validity properties, such as being terminated by a specific *terminator* operation. Despite this, it is possible nonetheless to infer facts that hold at the end of the prefix, which the fuzzer can then use to inform the next generation steps.

Incremental Semantics. In this work, for ease of exposition and convenience during development, we refer to both static analysis of programs (i.e. used to infer facts on programs) and the dynamic behaviour of programs as *semantics*.

Figure 6 presents two examples of useful information (semantics) that can be inferred from partial programs to be used by generators: the type of each value in the program, and the next available fresh variable name. For these to be used effectively in a fuzzer, where efficiency is important, it must be possible to compute these semantics efficiently on a prefix extension by reusing the previously-computed semantics for the prefix that is being extended.

We formalise this property via the notion of *incremental semantics*. In the following definition, the output of the semantics, A, is intentionally left abstract. This allows the definition to be instantiated for different domains, which is necessary since dialects can describe arbitrary abstractions.

Definition 3.3 (Incremental Semantics)**.** Semantics $S : K \to A$ is *incremental* if there exists a function $f : A \to \text{Op} \to A$ such that, for any prefix P, the semantics of its extension with operation e, P', satisfies:

$$S(P') = f(S(P), e)$$

That is, the semantics of a prefix extension can be derived as a function of the semantics of the prefix.

The incremental property of S means it can be evaluated on the fly by keeping the output of S as a *state*.

Figure 6 contains two possible instantiations for A: either as a table mapping values to their types or as a single string, keep tracking of fresh variable names. The partial program in the diagram makes use of the dialects `scf` (structured control flow), `arith` (arithmetic operations) and `vector` (vector operations, and printing operations for scalars).

Constraints on the Generator. The generator is responsible for producing prefix extensions that preserve key program properties, referring to computed semantics to guide its choices. For example, the properties of being well-typed, avoiding variable name reuse in the same scope, and being free from unwanted behaviours are all properties the extension should maintain.

The computed semantics can be seen as a set of facts on the partial program, and the generator uses these facts to make its decisions. For example, one possible semantics is the set of values that each variable can take, and the generator's choices are based on this information. Therefore, the generator must also ensure that the extension, after evaluation, does not invalidate the facts the previous generation steps were based on. Such a situation can occur, for instance, if generating an extension *increases* the set of possible values that a variable can take (e.g. allowing the variable to take the value zero), which leads to invalidating an operation that was generated based on an outdated fact (e.g. that the divisor in a division is non-zero).

3.2 Evaluating Semantics

Ratte evaluates semantics (see point (6) in Figure 3) when incrementally generating programs to guide the next steps of the generation. To keep track of semantic information in a uniform way, many of the semantics in Ratte make use of a parameterisable and hierarchical symbol table that captures MLIR's scoping structure. The table has keys of type X (such as the set of MLIR values) an interpretation result of type Y, both kept arbitrary in the definition for flexibility:

$$\begin{aligned}&\mathsf{ScopeType} := \{\mathsf{Standard}, \mathsf{IsolatedFromAbove}\}\\&\mathsf{ScopedTable}_{X,Y} := [\\&\quad (s_1 \in X \to Y, t_1 \in \mathsf{ScopeType})), \ldots,\\&\quad (s_n \in X \to Y, t_n \in \mathsf{ScopeType})], n > 0\end{aligned}$$

where the nested scopes are represented as a stack, the first element of the list (stack) is the inner-most scope, and the parent of the scope indexed by i is the $(i+1)^{\text{th}}$ scope, if it exists. The ScopeType tag denotes the visibility of the current scope, following MLIR's value scoping rules [27]: Standard denotes the scope can access everything its parent is able to access, and IsolatedFromAbove indicates that the visible variables are limited only to those in the current scope.

3.2.1 Dialect-Agnostic Semantics.

Some semantic information, such as the types of values or the next available value ID, is *dialect agnostic*. These can be defined directly on

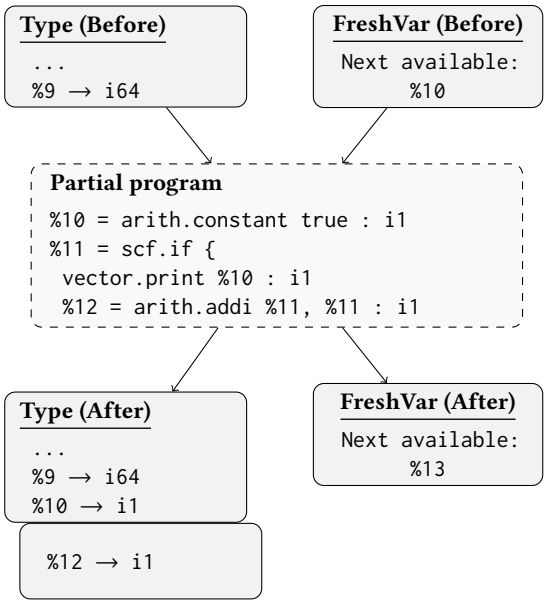

Figure 6. Two different dialect-agnostic semantics (value types, and the next available value ID) evaluated on a generated partial program segment. In the value types example, the scopes have Standard type.

the MLIR AST, as no dialect-specific information is involved in these semantics (see point (4) on Figure 3). We give two examples in Figure 6, showing different dialect-agnostic incremental semantics being evaluated on a generated (partial) program segment. The type semantics, which track typing information for each value, are updated to reflect the new typing environment. The next-fresh variable tracker is updated to the next available name for a fresh ID.

3.2.2 Modular Dialect Semantics. Since MLIR is a modular IR, any semantics for MLIR also needs to be specified *modularly* in order to support new dialects that can be defined by the user, without modifications to the semantics of existing dialects. For example, although it would be possible to write a function that interprets the string-based MLIR AST directly, semantics for all supported operations would need to be specified within this single monolithic interpretation function. This would mean that changes to operations in *any* dialect would require modifications of the monolithic interpretation function. This would make the code for the interpretation function difficult to scale and maintain. For an extensible IR like MLIR, such a non-modular approach would quickly become unmaintainable.

To define semantics modularly, Ratte makes use of *effect libraries* based on algebraic effects [31]. Effect libraries facilitate the specification and interpretation of domain-specific languages (DSLs). The constructs of a particular DSL are specified as an effect, analogous to an AST node. The semantics

Table 1. Table of embeddings for MLIR features (of the grammar in Figure 1) into a programming language supporting effect libraries. The exception is *Value*, which is not a term of the grammar, but is a synonym for *operand* and *result* (following convention of the official documentation [27]).

Feature	Embedding
type	Interface (as in Figure 10)
Value	The Value's ID and type (String, String)
operand	Same as Value
result	Same as Value
operation	Effect constructor
region	Funcs of type [Value] → Effect-AST
block	Labelled funcs (String, [Value] → Effect-AST)
attribute	Argument to effect constructor

are defined as transformations, or handlers, on the *effects-based AST* (hereon referred to as the *effects-AST*) representation of the DSL. This allows semantics to be defined modularly: the semantics can be defined for a single effect as a handler, and then the handler can be used on any effects-AST that contains the effect. To give an effects-AST semantics, one can compose together multiple handlers that, together, transforms the ASTs down to a value (with no effects remaining). Although the early implementation of effect libraries has been in functional languages, they are not intrinsically tied to functional programming, and effect libraries are also available for imperative languages like C++ [10] and Java [3].

Embedding into Effects-AST. The first step to making use of effect frameworks is to convert the string-based MLIR AST into the effects-based one so that modular semantics can be specified (see point (5) in Figure 3).

The specifics of such a conversion process are dependent on the exact effect framework chosen. However, we will highlight several aspects of the conversion that would apply to all embeddings of MLIR into any effect framework. The interested reader can refer to the associated source code [44] for details on the implementation within the Polysemy effects framework [23].

Effect frameworks allow DSLs to be defined modularly, and we take advantage of this feature for MLIR.

An overview of the embeddings for MLIR features into effect systems is shown in Table 1.

Each MLIR operation is embedded as an effect, with the operation name as the effect constructor's name, and the operation arguments as the effect arguments. Operations with assignments are embedded as effects that return a unit value, and operations without assignments are embedded as effects that return the output values. In Figure 7 and Figure 8, we show examples embedding instances of `arith.addi` and `func.func` operations within an effect system.

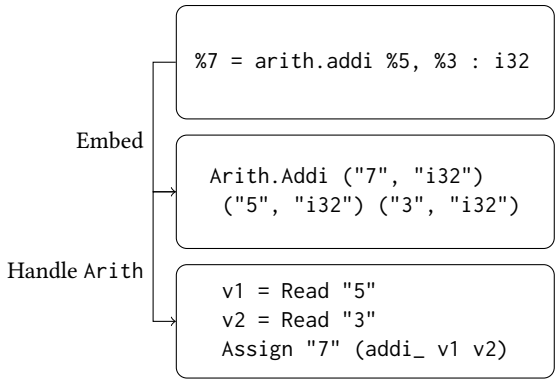

Figure 7. Embedding of an instance of `arith.addi` operation as an effect, and the result of applying handlers (partial interpretation) on the effect embedding.

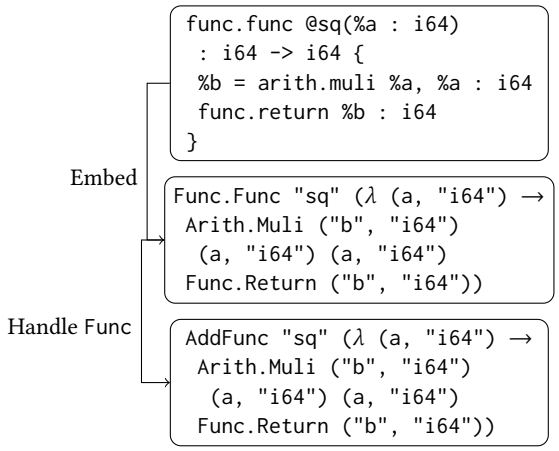

Figure 8. Embedding of an instance of `func.func` operation as an effect, and the result of the handler application on the effect. The `AddFunc` effect stores the effect-AST, to be called later with arguments via the `CallFunc` effect.

A region is embedded as a function that takes a number of region arguments (arguments to the first block [27]) and returns an effects-AST for the region's body. A block is embedded as a labelled function from block arguments to the effects-AST of its body. Terminator operations (block terminators) use function labels to dispatch calls to next blocks.

Since MLIR has a recursive structure between the operations, regions, and block constructs, embedding the AST into the effects needs to be done recursively. The particular conversion scheme used in Ratte proceeds in a depth-first way (through the MLIR AST): for each operation, the converter is applied on any associated regions first, to obtain their embeddings, before constructing the effect instance using the region embeddings.

Modular Semantics on Effects-ASTs. We now illustrate how MLIR semantics are evaluated by effect libraries through several examples. Interpretation of the effects-AST can be seen as transformations that convert the AST at a higher level of abstraction into one at a lower level.

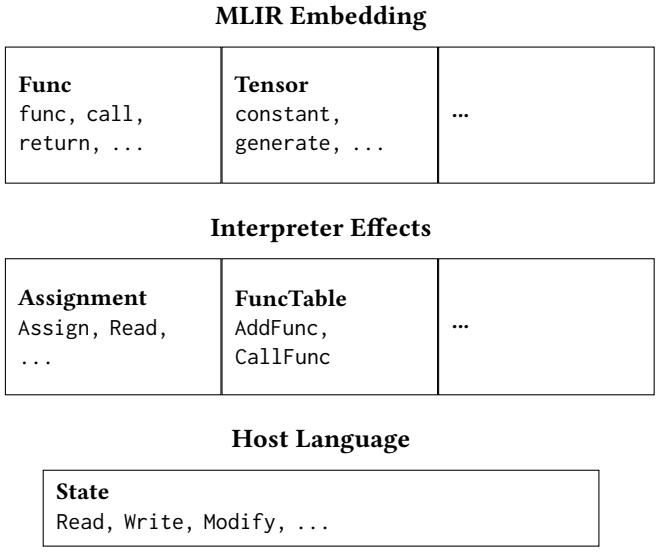

Figure 9. Structure of effects-based interpreters for MLIR programs. Each layer is an abstraction level in the interpreter, and boxes represent a modular effect component, containing the available operations for the component. The components shown are for illustrative purposes, as the combination depend on the semantics supported. All stages are written using the effect framework, and code produced in the final stage is in the host language for execution.

Once embedded as higher-level effects, interpretations can be defined arbitrarily. Although other interpretations are possible, Ratte primarily uses the concrete value interpretation to compute semantics used by the generator. The interpreter converts the MLIR effects-AST into a lower-level DSL, which contains assignment, error, scoping, function table, and writer operations. These are then translated further to executable code (in the host language—in Ratte's case this is Haskell) that implements the semantics. To provide concrete semantics for a dialect in this way, the user of Ratte needs to provide function that transform the operations of the dialect into the lower-level DSL. An overview of the main stages of this interpretation is illustrated in Figure 9, along with the operations available within each DSL component.

The lower-level DSL that MLIR dialects translate to is not fixed, and the DSL combination is variable based on the implementation needs of the higher-level semantics.

Dialects can introduce new types into the abstraction, and an interpreter needs to know the combination of types used to record their values during interpretation. However, to provide modular semantics, it is necessary to *delay* specifying the concrete combination of types used as much as possible.

```
class IntegerInterface T where
    constanti_ :: Attribute -> T
    addi_ :: (T, T) -> T
    muli_ :: (T, T) -> T
    muluiExtended_ :: (T, T) -> (T, T)
    divsi_ :: (T, T) -> Nullable T
    ...
    iTypeWidth :: T -> Int
    zero :: Int -> T

class BoolIntegerInterface TI TB X where
    adduiExtended_ :: (TI, TI) -> (TI, TB)
    cmpi_ :: (Attribute, TI, TI) -> TB
    select_ :: (TB, X, X) -> X
    ...
    false :: TB
    true ::  TB
```

Figure 10. Interfaces for the integer type in Haskell-like pseudocode, where `class` denotes an interface, `T, TI, TB, X` all denote type variables, and `m :: T` denotes the type of the method `m` is `T`. There is a distinction between the basic integer interface, `IntegerInterface`, implemented by types such as `i1, i8, ..., index`, and the interface for integer functions that uses booleans, `BoolIntegerInterface` (`TB` parameter satisfied by e.g. `i1`, and the `TI` parameter satisfied by standard integer types e.g. `i1, i8, ... index`). Interfaces for types contain all non-side-effecting computations on the type.

This allows semantics to be written that make the minimum stipulation on the types combination, and make them usable in all contexts where their required type is available.

Ratte treats types as an *interface*, and semantics are written against the type's interface rather than any concrete type implementations. Type interfaces contain all *pure* computations that can be performed on the value of a type. For example, some pure operations on the integer type are listed in Figure 10. Note that some pure operations make use of multiple types (e.g. the `cmpi, select` operations uses an integer type, and also another type that can be treated as a boolean), and these are encoded as a multi-parameter interface. A combination of types is often used during interpretation, and the values stored in the symbol table is a *union* of the types used.

Interactions Between Dialect Semantics. MLIR semantics can also *interact*, for instance by operations working on values from multiple dialects (e.g. `vector.print`), regions that contain code written in other dialects (e.g. `linalg.generic`, `scf.if`, `func.func`), or specific conversions between types (`index` to and from integer). Whilst the semantics of individual dialects can be unclear from the prose-based language documentation, the interaction of semantics between different dialects is even less well specified. Through the Ratte project, we developed a framework for specifying the semantics of dialects in terms of definitional interpreters. This process distilled the types of interactions we encountered and how these can be handled in a semantics framework.

Noninteracting Unions. This situation is when two dialects are used within the same IR layer, but are noninteracting in the sense that instances of operations in one dialect do not use any values of the other dialect. In this case, no extra work needs to be done to specify their interactions, as the semantics of the dialects are independent.

Parameter Interfaces. Operations such as `vector.print` can accept values from other dialects (i.e. outside of the `vector` dialect). This interaction is handled by requiring types usable by the operation to implement an operation-specific interface. For `vector.print`, this would be an interface that produces a string from the type, for printing.

Regions. Operations that contain regions, for example, `linalg.generic` and `func.func`, interact with the semantics of other dialects only *implicitly* by calling the region. The parent operation treats the region as a black box and interacts with the semantics of the region only through execution and extraction of execution results. For instance, by calling the region repeatedly on different arguments in the case of `linalg.generic`, or by storing the continuation within the state to be called later in the case of `func.func`. No additional work is needed to specify the interactions of the semantics.

Specific Instances. Some operations act specifically between abstractions, for instance, the `arith.index_cast` operation converting between `index` and integer types. These conversions have specific semantics and would need to be done on a case-by-case basis through a specific interface that bridges the two types. The interface specifies the two types that are needed (e.g. one implementing the integer interface, the other implementing the index interface), and the semantics is defined for the two types only. This is different from parameter interfaces, as the interface is written for two specific types, rather than one interface shared by all types.

3.3 Semantics-Guided Generators

Ratte generates programs in an incremental way. The generator (see point (1) on Figure 3) takes a source of randomness (i.e. a pseudorandom number sequence), and the semantics evaluated on the partially generated program (see point (2) in Figure 3). The generator uses the semantic information to inform its choices, and it is the generator's responsibility to ensure that the programs generated maintain necessary invariants (e.g. the program being free from undesired behaviours).

Generators are structured to follow the syntax of the IR [11], where an operation-generator instantiates its parameters and attributes randomly, and calls a region-generator for its region(s); a region-generator chooses the scoping type (`Standard` or `IsolatedFromAbove`) and calls a block-generator for its blocks; finally, a block-generator chooses its name, and calls an operation-generator for its body.

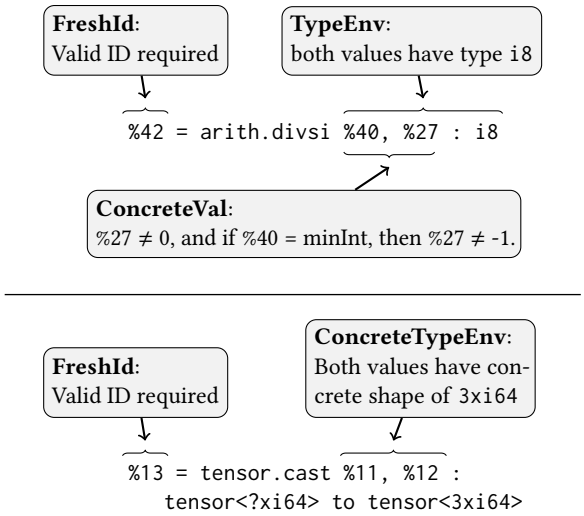

Figure 11. Usage of semantics in the generation of the `arith.divsi` and `tensor.cast` operations.

The semantics is evaluated when a group of related operations, called *IR fragments* are generated (see point (3) on Figure 3), and appended onto the partial program. IR fragments are multiple chained extensions (described in Definition 3.3), which can be convenient if the extensions are related, for instance, generating runtime checks alongside a potentially unsafe operation at the same time.

The generator requests semantic information through *interfaces*. As in Definition 3.3, incremental semantics can be thought of as the state of a partial program, and generators use the computed state by asserting that the state implements specific interfaces/semantics. Multiple semantics may be required by the generators through interfaces, and this can be satisfied by a state consisting of a tuple of individual states, each implementing one of the required interfaces. Semantic updates proceed in the expected way for tuples of states, by updating each state of the tuple independently.

As shown in Figure 11, generating the `arith.divsi` operation requires several interfaces to ensure the absence of undesirable behaviours. `arith.divsi` requires the inputs to be integer types of the same width, and therefore the *typing information* is needed to choose values of valid types for the operation. The operation also creates *new* values, and these need to be assigned valid IDs that do not clash with existing ones, meaning the store should have a mechanism for generating fresh IDs for the current scope. Moreover, the operation contains undefined behaviours if either the divisor is zero, or if `arith.divsi MIN_INT, -1` is attempted to be computed, resulting in a signed-division overflow as the value `-MIN_INT` is not representable by the two's complement representation the `arith` dialect uses.

Similarly, in the second example of Figure 11, `tensor.cast` operation is casting from a tensor with dynamic sizes (but statically known) to concrete sizes. Dynamic sizes are a part of the `tensor` dialect and are type annotations where the concrete sizes of the containers are *forgotten* by the type system—allowing, for instance, functions to be written for a family of tensors with a variable size component. Whilst casting does not alter the value of the tensor, a runtime error will be triggered if the sizes or the base types of the containers used in the cast do not match. Since a new value is being created for this operation, we also require a fresh ID generator. In addition, we need the state to implement an interface tracking the concrete types of compound containers, in order to avoid triggering runtime errors by ensuring the types chosen by the generator for the operation are between compatible types. Note that the concrete type is different to the syntactical types within MLIR: syntactical types allow for components of the types to be elided, whereas the concrete types track the statically inferred *runtime* values of the types in the `main` function, and no components are elided.

3.4 Test Oracles

The generator should generate programs P that produce *some* observable output. This allows the program to be compiled (and possibly executed) with an MLIR implementation, and checked against a known expectation (oracle) for the output to identify implementation bugs. A weak oracle is simply checking that the compiler does not crash; we call this the *non-crash oracle*. Alternatively, a stronger oracle is via differential testing—either across different compiler implementations, or cross-checked against the Ratte-defined semantics; we call this the *differential testing oracle*.

Output-producing code (e.g. prints) can be appended to the end of the program, as a final step of generation, to allow these oracles to be checked. For the non-crash oracle, the extra code is not necessary but can be useful in triggering/preventing some compiler passes. For the differential testing oracle, the variables that are known to be well-defined are used to produce an output. To ensure this is the case, Ratte performs a well-definedness analysis of variables. For instance, `tensor.empty` produces values that are not well-defined,`tensor.fill` always produces a well-defined value, whilst other operations propagate the undefinedness.

3.5 Implementation and Support Code

The semantics, parsing, and generation components are implemented using ≈ 6000 lines of Haskell. Support code for orchestrating fuzzing experiments is implemented with ≈ 300 lines Python. The implementation uses *Polysemy* [23] as the effect system, the *Parsec* [16] combinators for parsing MLIR, and the *QuickCheck* [7] library for building generators.

Thus far, Ratte provides detailed support for generation and concrete interpretation oracles for the `arith` compilation, covering all operations on the integer and index types. In the `linalg` dialect, all operations are syntactical sugar forms of the `linalg.generic` operation. Ratte provides a

best-effort implementation for generation and interpretation of `linalg.generic`, supporting instances with permutation-based indexing maps, and with any valid region. Finally, for the `tensor` dialect, Ratte supports a subset of operations (`empty, fill, extract, generate, constant, yield`). Overall, Ratte currently supports the interpretation and generation of 43 operations across core dialects (`arith, func, scf, linalg, vector, tensor`) of the MLIR framework. The validated semantics are also usable as an independent reference interpreter (by parsing the MLIR generic textual format [27]), which can help both MLIR developers and users.

The full source of Ratte generators, reference interpreters and support code are available in the appendix.

4 Evaluation

We now present an empirical evaluation based on our experience using Ratte at various stages in its development to find bugs in the production implementation of MLIR. Moreover, we also compare Ratte against an existing state-of-the-art MLIR fuzzer, MLIRSmith [41]. Our empirical evaluation centres on the following two research questions:

RQ1 How effective is Ratte for finding lowering and optimisation bugs? (Section 4.1)

RQ2 How effective is Ratte at generating programs for end-to-end testing of MLIR-based compilers compared to MLIRSmith? (Section 4.2)

We also discuss how our work led to reliability improvements to the MLIR ecosystem, including enhancements to the MLIR language specification (Section 4.3).

A by-product of our work on modular semantics-aware fuzzers is a composable reference implementation for a number of MLIR dialects that has been thoroughly validated against the production implementation. We have made this validated semantics usable as an independent reference implementation by combining it with a parser, which can help both developers and users of MLIR.

4.1 Bugs found

Ratte is used to validate the correctness of *compilations*, rather than single passes. We applied Ratte at various stages of development to test *end-to-end* compilations, with an executable target dialect. These experiments were conducted in bursts, but overall we estimate the total associated runtime to be one week, each run executing overnight on an 8-core laptop CPU, continuing until no new bugs were found for a significant amount of time ($\approx$ 6 hours). Our experiments focused on compilations targeting `{llvm}`, which is executable using the `mlir-cpu-runner` tool, producing an output that can be validated against the oracles (Section 3.4).

We implemented several semantics-guided generators of full programs, covering a range of abstractions, by composing together the generators for individual operations. These generators yield compileable programs that are free from undefined behaviours by construction (which we call UB-free). Details on the generators are summarised in Table 2.

Table 2. Generators used for bug-finding. The *Name* column is the primary dialect being fuzzed. The *Dialects* column is the dialects used by the generated programs. *Target* is the target dialect of the tested compilation in MLIR.

Name	Dialects	Target
Arith	`{arith, scf, func, vector}`	`{llvm}`
LinAlg	`{linalg, arith, func, vector}`	`{llvm}`
Tensor	`{tensor, arith, func, vector}`	`{llvm}`

So far, our use of Ratte led to the discovery of 8 previously unknown bugs, which are summarised in Table 3. Of the 8 bugs detected, 6 were miscompilations, and 2 of the 6 miscompilations were caused by bugs in the lowering passes. The remaining 2 bugs are programs that have been wrongly rejected by the static checks of the compiler, yielding no code when they should have been compiled successfully.

All bugs found depend on Ratte generating *statically correct* MLIR programs. This is the case even for verifier bugs: with generators like MLIRSmith that produce an unpredictable mixture of valid vs. invalid programs, there is no way to know if the verifier rejecting a program is the correct behaviour.

All of the miscompilation bugs depend on Ratte generating programs that are also *dynamically well-defined*, computing deterministic, well-defined results. Without such guarantees, there would be no way to know whether a mismatch between the result of the "production" compilation stack vs. the Ratte reference semantics is acceptable. With well-defined programs, such a difference is never acceptable.

The miscompilations triggered by buggy optimisations (for instance, in Figure 2) could in principle be detected by applying differential testing over optimisation passes on Ratte-generated programs, i.e. without referencing the Ratte semantics. However, they would still rely on the UB-free property of the Ratte-generated programs.

In contrast, the miscompilations triggered by lowering cannot be detected by cross-optimisation level testing, because the incorrect lowering is applied regardless of whether optimisations are executed. Detecting these bugs thus depends on Ratte's full power—its ability to generate well-defined programs and to provide alternative, unambiguous reference semantics for differential testing. Figure 12 is a reduced test case found by Ratte, triggering a bug in the *lowering* of the `arith.floordivsi` operation. The cause was an incorrect implementation of the lowering pass for `floordivsi`, where one of the intermediate values computed was $-2^{63}/-1$. This is a signed division overflow in the `llvm` target [21] and introduced a `poison` value into the result.

Table 3. List of bugs found[1]. Symptom denotes observed error: *Miscompile* is when wrong code is generated, and *Rejection* is when the verifier wrongly rejects valid programs. Oracle are ones from Section 3.4 that detected the bug: *NC*=non-crash oracle, *DT-O*=optimisation-level differential testing, and *DT-R*=differential testing against concrete Ratte semantics. *Detected With* is the operation/operation generator that was isolated to have triggered the bug, after reducing the bug-triggering test case.

	Compiler Phase	Symptom	Status	Pass	Oracle	Detected With	Issue
1	Optimisation	Miscompile	Submitted	`canonicalize`	DT-R	`arith.index_castui`	90238
2	Optimisation	Miscompile	Confirmed	`canonicalize`	DT-R	`arith.index_cast`	90296
3	Verifier	Rejection	Confirmed	`remove-dead-values`	NC	`func.call`	82788
4	Verifier	Rejection	Confirmed	`convert-arith-to-llvm`	NC	`arith.addui_extended`	84986
5	Optimisation	Miscompile	Fixed	`canonicalize`	DT-R	`arith.mulsi_extended`	88732
6	Optimisation	Miscompile	Fixed	`convert-arith-to-llvm`	DT-R	`arith.ceildivsi`	89382
7	Lowering	Miscompile	Fixed	`arith-expand`	NC	`arith.floordivsi`	83079
8	Lowering	Miscompile	Confirmed	`arith-expand`	DT-R	`arith.ceildivsi`	106519

```
func.func @main() {
    %cm, %cn1 = call @func1() : () -> (i64, i64)
    %1 = arith.floordivsi %cm, %cn1 : i64
    vector.print %1 : i64
    return
}
func.func @func1() -> (i64, i64) {
    %cm = arith.constant -9223372036854775807 : i64
    %cn1 = arith.constant -1 : i64
    return %cm, %cn1 : i64, i64
}
```

Figure 12. Example lowering bug found by Ratte. The program computes and prints $(-2^{63}+1)/-1$, which should yield $2^{63}-1$, but produces an undefined value.

4.2 Comparison with MLIRSmith

We now turn to a comparison with the state-of-the-art in program generation for MLIR prior to our work: MLIRSmith [41]. Though MLIRSmith focuses on crash bugs and not miscompilations, it still generates syntactically valid MLIR programs on which passes can be executed. However, MLIRSmith programs are not expected to compile fully. Nonetheless, as a generator of syntactically valid MLIR programs, we investigate MLIRSmith's utility in producing programs for differential testing to detect semantic bugs. For this, only compileable and UB-free programs can be used.

For each dialect combination in Table 2, we generated 1,000 programs using MLIRSmith, and evaluated each for compileable/UB-freeness using the Ratte interpreter. Our results are presented in Table 4. In comparison, all Ratte-generated programs are compileable and UB-free by design. As Ratte also interprets during generation, it has a lower throughput compared to MLIRSmith. For Arith, Linalg and Tensor dialect combinations, Ratte generated 1000 programs in 191s, 193s, and 196s respectively, whilst MLIRSmith generated 1000 programs in 67s, 59s, and 82s.

Table 4. Compileability/UB-freeness of MLIRSmith generator restricted to known valid dialect combinations (of Table 2) and of the *Unmodified* MLIRSmith. *Compiled*=proportion of programs that successfully compile, *UB-Free*=percentage of programs found by the Ratte interpreter to be free from UB. (*) For the unmodified version, the pipeline ran is not a full compilation (applying only the `-canonicalize` pass), since to the best of our knowledge, there are no pass combinations able to lower the produced IR down to executable code. (†) Of the compileable operations, most contained no `linalg` operations, and ones that did often had long runtimes due to large loop bounds, making UB-checking infeasible.

Name	Dialects	Compiled	UB-Free
Unmod	All MLIRSmith dialects	7.80*%	N/A
Arith	Arith (of Table 2)	100.00%	1.10%
LinAlg	Linalg (of Table 2)	6.90%	N/A†
Tensor	Tensor (of Table 2)	99.40%	0.00%

To effectively apply *different optimisation levels* (DOL) testing, any validity rate significantly lower than 100% can mean a large number of false positives in stable projects like MLIR. This greatly diminishes the usability of MLIRSmith for DOL testing, as every false positive requires costly manual intervention to differentiate between a real bug vs. a UB.

4.3 Specification improvements

Our work identified several aspects of the MLIR documentation, which serves as the specification for MLIR, that were incorrect, incomplete, or ambiguous, and for which we have contributed fixes. Ambiguities were found in the documentation for the `arith` dialect, on the semantics of shifts-past-end, the underlying representation, and the exact behaviour of divisions by zero. Although previous MLIR users mostly assumed [13] the corresponding LLVM semantics [20] for `arith`, our pull request clarified this for MLIR. In the `linalg`

dialect, ambiguities were found when our semantics disagreed with the outputs of the compilation [30]. The case involves incomplete indexings of the input arguments by the indexing maps, treated as undefined in the production MLIR implementation. However in our semantics, the values were assumed to be the initial value. We submitted a PR clarifying this behaviour, currently under review [29].

5 Related Work

Compiler Fuzzing. There exist many approaches to compiler fuzzing [22, 24], and Ratte is an instance of an analysis-guided grammar-based generator, a technique similar to that used in Csmith [43] and YARPGen [18], the previous and current state of the art in the validation of C compilers. Csmith uses information from static analysis to ensure the generation of UB-free C programs. Whilst in YARPGen, ad hoc analysis (e.g. variable usage, expression safety) is combined with a *concrete interpreter*, queried during generation to create UB-free programs that allow certain conservative dynamic checks used in Csmith to be elided. Ratte generalises the YARPGen approach for MLIR dialects, to describe both analysis and concrete interpretations, and in a composable way. Ratte embeds this observation within a flexible framework for describing composable MLIR interpretations and analysis. Unlike prior work, Ratte applies this to a language (MLIR) with open syntax and composable semantics, rather than the fixed syntax/semantics of C/related languages. To achieve this, Ratte leverages functional programming techniques to describe composable dialect semantics and an extensible framework for building fuzzers of MLIR-based compilers.

The only known prior works to us on MLIR fuzzing are MLIRSmith [41], MLIROd [34], and SynthFuzz [17]. MLIRSmith and MLIROd focus on *crash* bugs and do not aim to generate programs with oracles strong enough to detect miscompilations. SynthFuzz is a general mutation technique applicable to all MLIR programs. However, it does not produce programs for finding semantic bugs, as the general mutation does not maintain dialect-specific semantic constraints, making it usable with crash oracles only. To the best of our knowledge, Ratte is the first generator of well-defined programs for MLIR, which requires significant semantic information—made possible through its modular interpretation framework.

An interesting approach to compiler testing involves generating equivalent programs and then checking that, after compilation, they indeed behave in an equivalent manner. This is an example of metamorphic testing [6], and one possible application is by directly generating equivalent programs from scratch [35]. Alternatively, semantics-preserving mutations can be applied to an existing program to obtain a set of equivalent programs [8, 9, 15, 33]. Applied to MLIR, such a technique also has the potential to find miscompilations, including lowering bugs. The main challenge is describing per-dialect transformations, as well as having principled ways to compose transformations when working with multiple dialects, and is an interesting avenue for future work.

Modular Semantics. Ratte uses effect systems to address the expression problem [40], which arises when defining semantics on extensible data: MLIR syntax. Although functional implementations of effect systems can have a steep learning curve due to their type-driven design, they help eliminate bug-prone programs and integrate well with type-driven property-based testing tools like QuickCheck [7]. Previously, effect systems have been used to specify and verify modular semantics for LLVM [45], and for verifying some MLIR passes [2]. Other approaches for the expression problem include multiple dispatch [4], or object algebras [28].

6 Conclusions

We have introduced, to the best of our knowledge, the first fuzzing framework for MLIR that focuses on generating dynamically and statically well-defined programs. Unlike previous approaches, our focus is on miscompilations, the most insidious compiler bugs with no obvious symptoms. Our design is influenced by the prior art in C-language fuzzers, YARPGen. In building our framework, we also introduce an embedding of MLIR using higher-order effects, through which we describe the semantics of MLIR operations, and particularly the semantics of regions, a novel and major defining characteristic of MLIR. We demonstrated the effectiveness and generality of our approach by finding 8 previously unknown bugs, including 6 miscompilations within core parts of the MLIR codebase. Through the successful marriage of techniques developed by the functional programming community with fuzzing, our framework is composable and fully extensible to new dialects and semantics.

7 Acknowledgements

We are grateful to Luke Geeson, James Lee-Jones, Hamid Maazouz, Alyssa Renata, John Wickerson and our ASPLOS reviewers for their feedback on earlier drafts of this work. This work was supported by an EPSRC Programme Grant on *Interface reasoning for interacting systems* (grant number EP/R006865/1).

A Artifact Appendix

A.1 Abstract

This artifact contains the supporting code, executables, and results data as described in the paper on Ratte, a framework for building composable fuzzers and interpreters for compilers. Ratte generates executable programs with known, deterministic behaviours for MLIR, which is a composable intermediate representation (language) used by compilers. These programs are used for the purpose of finding miscompilation bugs within compilers by differential testing. As a

result of the Ratte methodology, independent interpreters for the language are also developed alongside the fuzzers, and are validated by cross-checking the results against the reference interpreter on the Ratte-generated programs. This artifact contains the bug reports of the bugs found by Ratte, executable program generators for several MLIR dialects, and the executable reference interpreter for the supported operations of Ratte (listed below).

A.2 Artifact check-list (meta-information)

- **Program: Ratte, MLIR**
- **Compilation: MLIR, LLVM, Haskell, Python**
- **Binary: mlir-quickcheck, ref-interpreter**
- **Hardware: Intel, AMD**
- **Output: MLIR programs, interpretation result**
- **Experiments: fuzzing, interpretation**
- **How much disk space required (approximately)?: 3Gb**
- **How much time is needed to prepare workflow (approximately)?: 30 minutes**
- **How much time is needed to complete experiments (approximately)?: 2 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPL v3.0**
- **Archived (provide DOI)?: 10.5281/zenodo.14790908**

A.3 Description

A.3.1 How to access. Through the provided DOI link to the repository.

A.3.2 Hardware dependencies. Minimum 4Gb of RAM, x86 processor.

A.3.3 Software dependencies. Docker / other container managers

A.4 Installation

This artifact is intended to be built using Docker (Podman would probably work, but is untested).

From the source directory, run:

```
docker build -t ratte-artifact .
```

This will build the image and take ≈ 25 minutes.

A.5 Evaluation and expected results

A.5.1 Verify Setup. Enter the image and check, as a first step, that the unit tests pass:

```
docker run -it ratte-artifact
cabal test
```

This step will take ≈ 1 minutes to build and run the tests.

Generate a program using the `arith, scf, func` dialects of size 30, and its expected execution result, run:

```
cabal run mlir-quickcheck - -d=ariths -n=30 \
   2> /dev/null
```

which should produce output of the form:

```
/app# cabal run mlir-quickcheck -- -d=ariths -n=30 2>
↪  /dev/null
"builtin.module"() ({
    ^bb0():
...
--== Expected output:
%6=...
...
--== End of output.
```

A.5.2 Bugs found. Each of the reduced bugs can be found within `bugs/` folder of the artifact, named according to their numbers on the table below and in the paper (e.g. `3.mlir`). The table below contains more details on each bug, along with a link to their respective GitHub issue. Each GitHub issue also contains a link to a Godbolt to reproduce the bug.

Table 5. Description of the bugs mentioned within the paper, and their respective GitHub issue.

	Pass	Issue
1	`canonicalize`	90238
2	`canonicalize`	90296
3	`remove-dead-values`	82788
4	`convert-arith-to-llvm`	84986
5	`canonicalize`	88732
6	`convert-arith-to-llvm`	89382
7	`arith-expand`	83079
8	`arith-expand`	106519

Verify that the bug reports exist and are as described in the paper.

A.5.3 Proposed MLIR Spec and Testing Changes. The spec changes mentioned in Section 4.3 of the paper are as follows.

Arith representation clarifications: 1 and 2; Linalg dialect documentation updates: 1.

The proposed additional integration tests is here.

Verify that the proposals exist and are as described in the paper.

A.5.4 Fuzzers. The binary `mlir-quickcheck` implements several preset fuzzers based on the generators implemented.

The following will run the fuzzer for a given preset, generating a program of size (roughly) 50 variables, alongside its expected execution output:

```
cabal run mlir-quickcheck - \
  -d <PRESET> -n 50 2> /dev/null
```

where <PRESET> is one of:

- `ariths`
- `linalggeneric`
- `tensor`

The output should be a random MLIR program, using operations defined by the preset (respectively, `ariths` uses `arith` ops, `linalggeneric` uses `linalg.generic` ops, `tensor` uses

tensor ops), along with the expected output in a format similar to that in section "Verify Setup".

The output program should also be compile by `mlir-opt` using the following passes (one way to access `mlir-opt` is via Godbolt):

arith: `-arith-expand -test-lower-to-llvm`

linalggeneric: `-one-shot-bufferize -func-bufferize -cse -canonicalize -convert-vector-to-scf -test-lower-to-llvm`

tensor: `-one-shot-bufferize -func-bufferize -cse -canonicalize -convert-vector-to-scf -test-lower-to-llvm`

A.5.5 Reference Interpreter.

The interpreter based on the MLIR semantics implemented within Ratte can be found in the `ref-interpreter` binary.

The interpreter parses code in the generic MLIR format, which is defined as the "`mlir-generic-op`" within the official documentation.

The generic form can be obtained for MLIR programs by running `mlir-opt` with pass `-mlir-print-op-generic`.

To run the interpreter on an example file:

```
cabal run ref-interpreter -- \
  -f=app/Interpreter/examples/example2.mlir \
  -m=main
```

Optional. To run the interpreter for another MLIR file, e.g. `prog.mlir` with entry function `entry`, first convert into generic form (e.g. as follows: using Godbolt), then run the interpreter on the generic form file:

```
mlir-opt -mlir-print-op-generic prog.mlir > \
  prog-generic.mlir
cabal run ref-interpreter -- \
  -f=prog-generic.mlir -m=entry
```

Current limitations is that the interpreter only supports natural-number variable names.

A.6 Notes

The current operations supported by the reference interpreter are as follows: `arith.constant`, `arith.ceildivui`, `arith.ceildivsi`, `arith.floordivsi`, `arith.divui`, `arith.divsi`, `arith.remui`, `arith.remsi`, `arith.shli`, `arith.shrsi`, `arith.shrui`, `arith.cmpi`, `arith.addi`, `arith.andi`, `arith.maxsi`, `arith.maxui`, `arith.minsi`, `arith.minui`, `arith.muli`, `arith.ori`, `arith.subi`, `arith.xori`, `arith.addui_extended`, `arith.mulsi_extended`, `arith.mului_extended`, `arith.extsi`, `arith.extui`, `arith.trunci`, `arith.select`, `arith.index_cast`, `arith.index_castui`, `func.func`, `func.return`, `func.call`, `linalg.generic`, `linalg.yield`, `scf.yield`, `scf.if`, `tensor.constant`, `tensor.cast`, `tensor.extract`, `tensor.insert`, `tensor.dim`, `tensor.empty`, `tensor.yield`, `vector.print`.

References

[1] AMD. MLIR-Based AI Engine Toolchain. https://github.com/Xilinx/mlir-aie, 2025. Accessed: 2025-02-09.

[2] Bhat, S., Keizer, A. C., Hughes, C., Goens, A., and Grosser, T. Verifying peephole rewriting in SSA compiler IRs. In *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia* (2024), Y. Bertot, T. Kutsia, and M. Norrish, Eds., vol. 309 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:20.

[3] Brachthäuser, J. I., Schuster, P., and Ostermann, K. Effect handlers for the masses. *Proc. ACM Program. Lang. 2*, OOPSLA (2018), 111:1–111:27.

[4] Chambers, C., and Leavens, G. T. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst. 17*, 6 (1995), 805–843.

[5] Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., and Zhang, L. A survey of compiler testing. *ACM Comput. Surv. 53*, 1 (2021), 4:1–4:36.

[6] Chen, T., Cheung, S., and Yiu, S. Metamorphic testing: a new approach for generating next test cases. Tech. Rep. HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.

[7] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000* (2000), M. Odersky and P. Wadler, Eds., ACM, pp. 268–279.

[8] Donaldson, A. F., Evrard, H., Lascu, A., and Thomson, P. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang. 1*, OOPSLA (2017), 93:1–93:29.

[9] Donaldson, A. F., Thomson, P., Teliman, V., Milizia, S., Maselco, A. P., and Karpinski, A. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021* (2021), S. N. Freund and E. Yahav, Eds., ACM, pp. 1017–1032.

[10] Ghica, D. R., Lindley, S., Bravo, M. M., and Piróg, M. High-level effect handlers in C++. *Proc. ACM Program. Lang. 6*, OOPSLA2 (2022), 1639–1667.

[11] Goldstein, H., and Pierce, B. C. Parsing randomness. *Proc. ACM Program. Lang. 6*, OOPSLA2 (2022), 89–113.

[12] Kiselyov, O., Sabry, A., and Swords, C. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013* (2013), C. Shan, Ed., ACM, pp. 59–70.

[13] kuhar. [RFC] Define precise arith semantics. https://discourse.llvm.org/t/rfc-define-precise-arith-semantics/65507l, 2022. Accessed: 2025-02-08.

[14] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J. A., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021* (2021), J. W. Lee, M. L. Soffa, and A. Zaks, Eds., IEEE, pp. 2–14.

[15] Le, V., Afshari, M., and Su, Z. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, pp. 216–226.

[16] Leijen, D., and Meijer, E. Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-27, Universiteit Utrecht, July 2001.

[17] Limpanukorn, B., Wang, J., Kang, H. J., Zhou, Z., and Kim, M. Fuzzing mlir compilers with custom mutation synthesis. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (2024),

IEEE Computer Society, pp. 457–468.

[18] Livinskii, V., Babokin, D., and Regehr, J. Random testing for C and C++ compilers with yarpgen. *Proc. ACM Program. Lang. 4*, OOPSLA (2020), 196:1–196:25.

[19] Livinskii, V., Babokin, D., and Regehr, J. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proc. ACM Program. Lang. 7*, PLDI (2023), 1826–1847.

[20] LLVM Foundation. LLVM language reference manual. https://llvm.org/docs/LangRef.html, 2024. Accessed: 2025-02-08.

[21] LLVM Foundation. LLVM language reference manual: 'sdiv' instruction. https://llvm.org/docs/LangRef.html#sdiv-instruction, 2024. Accessed: 2025-02-08.

[22] Ma, H. A survey of modern compiler fuzzing. *CoRR abs/2306.06884* (2023).

[23] Maguire, S. polysemy-research/polysemy. https://hackage.haskell.org/package/polysemy. Accessed: 2025-02-08.

[24] Marcozzi, M., Tang, Q., Donaldson, A. F., and Cadar, C. Compiler fuzzing: how much does it matter? *Proc. ACM Program. Lang. 3*, OOPSLA (2019), 155:1–155:29.

[25] McKeeman, W. M. Differential testing for software. *Digit. Tech. J. 10*, 1 (1998), 100–107.

[26] MLIR Authors. Dialects. https://mlir.llvm.org/docs/Dialects/, 2024. Accessed: 2025-02-08.

[27] MLIR Developers. MLIR language reference. https://mlir.llvm.org/docs/LangRef/. Accessed: 2025-02-08.

[28] Oliveira, B. C. d. S., and Cook, W. R. Extensibility for the masses - practical extensibility with object algebras. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings* (2012), J. Noble, Ed., vol. 7313 of *Lecture Notes in Computer Science*, Springer, pp. 2–27.

[29] pingshiyu. [mlir][linalg] added some conditions for values being undefined in the documentation for linalg.generic. https://github.com/llvm/llvm-project/pull/96251, 2024. Accessed: 2025-02-08.

[30] pingshiyu. [mlir][linalg] linalg.generic miscompilation in case of multiple return values. https://github.com/llvm/llvm-project/issues/94179, 2024. Accessed: 2025-02-08.

[31] Plotkin, G. D., and Pretnar, M. Handlers of algebraic effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings* (2009), G. Castagna, Ed., vol. 5502 of *Lecture Notes in Computer Science*, Springer, pp. 80–94.

[32] Sharma, M., Yu, P., and Donaldson, A. F. Rustsmith: Random differential compiler testing for rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023* (2023), R. Just and G. Fraser, Eds., ACM, pp. 1483–1486.

[33] Sun, C., Le, V., and Su, Z. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016* (2016), E. Visser and Y. Smaragdakis, Eds., ACM, pp. 849–863.

[34] Suo, C., Chen, J., Liu, S., Jiang, J., Zhao, Y., and Wang, J. Fuzzing MLIR compiler infrastructure via operation dependency analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024* (2024), M. Christakis and M. Pradel, Eds., ACM, pp. 1287–1299.

[35] Tao, Q., Wu, W., Zhao, C., and Shen, W. An automatic testing approach for compiler based on metamorphic testing technique. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010* (2010), J. Han and T. D. Thu, Eds., IEEE Computer Society, pp. 270–279.

[36] Tensorflow Developers. Tensorflow MLIR. https://www.tensorflow.org/mlir, 2024. Accessed: 2025-02-09.

[37] The Flang Developers. Flang. https://github.com/flang-compiler/flang, 2024. Accessed: 2025-02-09.

[38] The IREE Authors. IREE - Intermediate Representation Execution Environment. https://iree.dev/, 2024. Accessed: 2025-02-09.

[39] Torch-MLIR Developers. The Torch-MLIR Project. https://github.com/llvm/torch-mlir, 2024. Accessed: 2025-02-09.

[40] Wadler, P. The expression problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998. Accessed: 2025-02-08.

[41] Wang, H., Chen, J., Xie, C., Liu, S., Wang, Z., Shen, Q., and Zhao, Y. Mlirsmith: Random program generation for fuzzing MLIR compiler infrastructure. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023* (2023), IEEE, pp. 1555–1566.

[42] Wu, N., Schrijvers, T., and Hinze, R. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014* (2014), W. Swierstra, Ed., ACM, pp. 1–12.

[43] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), M. W. Hall and D. A. Padua, Eds., ACM, pp. 283–294.

[44] Yu, P., Wu, N., and Donaldson, A. F. Artifact for Ratte: Fuzzing for miscompilations in multi-level compilers using composable semantics. https://doi.org/10.5281/zenodo.14768650, 2025. Accessed: 2025-02-09.

[45] Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., and Zdancewic, S. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang. 5*, ICFP (2021), 1–30.