IRDL: An IR Definition Language for SSA Compilers

Mathieu Fehr
mathieu.fehr@ed.ac.uk
University of Edinburgh
United Kingdom

Jeff Niu*
js2niu@uwaterloo.ca
University of Waterloo
Canada

River Riddle†
riverriddle@modular.ai
Modular AI
United States

Mehdi Amini
aminim@google.com
Google
United States

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

Tobias Grosser
tobias.grosser@ed.ac.uk
University of Edinburgh
United Kingdom

Abstract

Designing compiler intermediate representations (IRs) is often a manual process that makes exploration and innovation in this space costly. Developers typically use general-purpose programming languages to design IRs. As a result, IR implementations are verbose, manual modifications are expensive, and designing tooling for the inspection or generation of IRs is impractical. While compilers relied historically on a few slowly evolving IRs, domain-specific optimizations and specialized hardware motivate compilers to use and evolve many IRs. We facilitate the implementation of SSA-based IRs by introducing IRDL, a domain-specific language to define IRs. We analyze all 28 domain-specific IRs developed as part of LLVM's MLIR project over the last two years and demonstrate how to express these IRs exclusively in IRDL while only rarely falling back to IRDL's support for generic C++ extensions. By enabling the concise and explicit specification of IRs, we provide foundations for developing effective tooling to automate the compiler construction process.

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

Keywords: Compilers, Intermediate Representation, MLIR

ACM Reference Format:
Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. 2022. IRDL: An IR Definition Language for SSA Compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519939.3523700

*Jeff Niu is meanwhile working for Google.
†This work was performed in parts while at Google.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523700

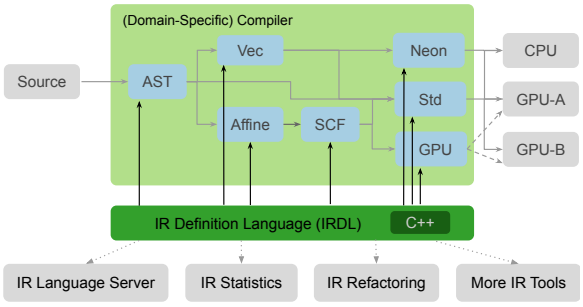


Figure 1. The domain-specific IR Definition Language (IRDL) combined with a small amount of generic C++ code (IRDL-C++) enables the concise specification of compiler IRs for the use within a multi-IR compilation flow. We expect IRDL to serve as a foundation for a future ecosystem of productivity-increasing tooling around IR design.

1 Introduction

Today, a production-quality compiler is typically developed in a manual process where (almost) all parts of it are implemented in a general-purpose programming language, making compiler construction slow and nearly impossible to reason about. In particular, the different intermediate representations (IRs) a compiler uses are hidden in internal data structures. For instance, Open64 uses a 5-level WHIRL representation [19]. Newer programming language frontends often introduce their own language-specific IRs to represent higher-level constructs. Swift [26], for instance, developed the Swift Intermediate Language (SIL) on top of LLVM [13] to implement high-level optimizations [17]. LLVM, as a compiler infrastructure, has not only its user-facing LLVM IR but additionally uses various internal ones that are typically not visible to its users: SelectionDAG, MachineInst, and MCInst [12]. All of these IRs are deeply embedded into their respective compilers. Hence, modifications require detailed compiler-specific knowledge, and even specialists are very hesitant to evolve existing IRs. While there exist approaches for generating parts of a compiler (e.g., parsers, backends,

```
func @conorm(%p: !cmath.complex<!f32>,
             %q: !cmath.complex<!f32>) -> !f32 {
  %norm_p = cmath.norm(%p) : !f32
  %norm_q = cmath.norm(%q) : !f32
  %pq = std.mulf %norm_p, %norm_q : !f32
  return %pq : !f32
}
```

(a) Before optimization

```
func @conorm(%p: !cmath.complex<!f32>,
             %q: !cmath.complex<!f32>) -> !f32 {

  %pq = cmath.mul(%p, %q) : !f32
  %conorm = cmath.norm(%pq) : !f32
  return %conorm : !f32
}
```

(b) After optimization

Listing 1. Optimizing conorm with an MLIR cmath dialect. High-level IRs enable simple peephole optimizations in MLIR.

code-generators), we lack a solution that streamlines the design of IRs themselves and enables a strong library and tooling ecosystem around them.

MLIR [14] took an important step towards facilitating the implementation of IRs. As a new compiler framework under the LLVM umbrella, MLIR reduces the cost of implementing an SSA-based compiler IR by providing a shared infrastructure for implementing new IRs, so-called "dialects". Dialects provide a unified way to interact with IRs in C++ and a generic textual syntax that resembles LLVM-IR. While MLIR IRs are always implemented in C++, MLIR's initial release also shipped a prototype of the so-called "Operator Definition Specification" (ODS). Using the TableGen generic record format that LLVM relied on for defining some of its backend components [11], the first variant of ODS allowed developers to synthesize some of the C++ code needed to implement individual MLIR operations. Over the last few years, we gradually expanded ODS together with the MLIR community until it became clear that we were building, hidden in a difficult-to-read record format, a tailor-made DSL for designing and working with compiler IRs.

With IRDL, we present a dedicated domain-specific language for concisely representing a diverse set of compiler IRs. IRDL is the result of working with the MLIR community with the objective to streamline the IR design process. In an IRDL-based compilation flow (Figure 1), we compile a program through multiple stages of abstraction while translating across various (combinations of) IR dialects. An IRDL specification describes each IR dialect composed of operations, types, and attributes (static annotations to operations), including special control-flow constructs such as terminator operations or nested control flow regions (see Section 2 for details). We also provide with IRDL-C++ an extension for representing IRs with complex properties and invariants that are best expressed in a generic programming language. Together, IRDL and IRDL-C++ are sufficient to define the IRs that arise in a typical SSA-based compiler in a self-contained IR definition file.

Together with the MLIR community, we have been working carefully towards decoupling the default set of MLIR IR definitions from the actual compiler stack and, as a result, have made these definitions inspectable. We can now analyze the full set of IRs defined in MLIR and can provide detailed data on the degree to which IRs can be specified in a self-contained domain-specific language. Our new streamlined IR design process enables a detailed analysis of the IRs in the MLIR ecosystem, which overall demonstrates the power of meta-tooling for IR design. We envision that the structured and self-contained IR format IRDL provides a foundation for a full suite of IR tooling, e.g. LSP (Language Server Protocol) [15] support, IR refactoring tools, and others, which we expect to simplify the process of designing and working with IRs.

Our contributions are:

- IRDL, a high-level language for defining arbitrary SSA IRs as dialects composed of operations, types, attributes, as well as expected invariants → Section 4;
- IRDL-C++, an extension of IRDL for using inline C++ in IR specifications that require Turing-complete language support → Section 5;
- An analysis of the suitability of IRDL to express 28 domain-specific IRs without a broad need for manually written C++ code → Section 6.

2 SSA, Regions, and MLIR

Static Single Assignment (SSA) intermediate representations (IRs) [2] are standard in compilers today. An SSA-based IR uses program variables - called SSA values - with the particular property that each variable is assigned a value at exactly one program location. As a result, each named SSA value uniquely identifies the location and operation that computes its value. Each operation takes as arguments a list of previously defined SSA values and returns zero or more (SSA) values. Values are typically associated with a type, which carries compile-time information. Similarly, operations may have attributes that describe static information. For example, we express a program to compute the norm of the product of two complex numbers (Listing 1) as a sequence of two MLIR operations, `cmath.mul` for the multiplication, and `cmath.norm` for the norm. Those operations use `!cmath.complex<f32>` types, which model complex numbers composed of single-precision floating-point values.

Basic blocks represent a sequence of operations and end with a special terminator operation, which passes control to one or multiple basic blocks. In addition, basic blocks

accept optional block arguments (semantically equivalent to ϕ nodes [14]) that enable terminator operations to pass values across block boundaries or leave the current set of basic blocks. A set of connected basic blocks with a single entry block and zero or more exit blocks forms a control flow graph (CFG). CFGs are contained in regions, which are usually used to implement functions (Listing 1). However, some extensions of SSA (e.g., MLIR [14]) allow operations to contain nested regions, which makes it possible to represent hierarchical control flow (e.g., if statements) and the resulting nested control flow graphs.

MLIR does not provide a predefined set of operations but relies on the concept of extensibility, with few built-in constructs leaving most of the IR customizable. Operations, types, and attributes are grouped into dialects, similar to namespaces or modular libraries. Each dialect sits at a given abstraction level. For example, the Linalg dialect in MLIR models linear algebra operations on either tensor or buffer operands. SCF represents structured control flow, and Arith, at a lower abstraction level, consists of standard arithmetic on integers and floating points. In our previous example (Listing 1), the `cmath` dialect represents computations on complex numbers, which ease the writing of compiler optimizations by providing the right abstraction level. For instance, we can write a domain-specific transformation that converts the multiplication of two norms into the norm of a complex multiplication (Listing 1a), an equivalent but faster computation.

In MLIR, operations can take by default an arbitrary number of operands, results, regions, or attributes. However, operations can additionally define verifiers which constrain the operands, results, regions, or attributes an operation can have. For instance, `cmath.mul` defines a verifier (Listing 2, `MulOp::verify`) which constrains the operation to have only two operands, one result, and no regions. It also constrains the operands and results to have the same complex type. Types and attributes may also define verifiers. For instance, the `cmath.complex` type defines a verifier to ensure that the type parameter `elementType` is a floating-point type. Verifiers are essential to define invariants on arbitrary operations, types, and attributes.

```
class CmathDialect : Dialect {
  StringRef getNamespace() { return "cmath"; }
  ... // More C++ code
}
class ComplexType : Type<CmathDialect> {
  Type elementType;
  static bool verify() {
    return elementType.isa<FloatType>();
  }
  ... // More C++ code
}
class MulOp : Op<CmathDialect> {
  Value lhs() { return operands()[0]; }
  Value rhs() { return operands()[1]; }
  Value res() { return results()[0]; }

  bool verify() {
    return numOperands() == 2 &&
           numResults() == 1 &&
           numRegions() == 0 &&
           lhs().type().isa<ComplexType>() &&
           lhs().type() == rhs().type() &&
           rhs().type() == res().type();
  }
  ... // More C++ code, e.g., constructors, parser,
      // printer, ...
}
class NormOp : Op<CmathDialect> { ... // More C++ }
```

Listing 2. C++ code defining an SSA IR with MLIR.

3 Deriving a Dialect with IRDL

We introduce IRDL by developing a simple domain-specific compiler IR for complex numbers as introduced in Listing 1. A given IRDL specification has sufficient information to derive: (1) parsers and printers for conversion to and from a textual representation, (2) data structures for the internal representation of the IRs, and (3) verifiers to assert IR invariants. Section 4 will provide a specification of IRDL that is sufficient to define the full feature set of the `cmath` IR.

To define the `cmath` IR used in our initial complex math example (Listing 1), we use IRDL to define the dialect itself together with the **`complex`** type and the **`mul`** and **`norm`** operations (Listing 3). Types define parameters, as well as constraints over them. For instance, **`complex`** defines a single parameter named **`elementType`**, which is constrained to be a floating-point type. These constraints are used to generate the corresponding C++ verifiers. Operations also define constraints for operands, results, attributes, and regions. In our example, **`mul`** constrains its operands and results to be equal **`complex`** types and, additionally, defines a printing and parsing format. While our simple example and many more sophisticated ones can be expressed entirely in IRDL (Section 4), other IRs may require specialized constraints that are beyond the scope of IRDL. IRDL-C++ (Section 5) provides the necessary directives to also express such use cases. Together, IRDL and IRDL-C++ are capable of expressing a full compiler IR in just a single self-contained text file.

We see IRDL as *the means* of defining and working with IRs in MLIR and potentially other compilers. Compiler developers can simply register a new dialect (e.g., `cmath`) in MLIR by providing an IRDL specification file instead of writing, compiling, and linking several complex C++ or TableGen files. The compiler then instantiates all necessary data structures at runtime (without recompilation) and is immediately prepared to reason about such IRs. Together with the dynamic pattern rewriting support currently in construction in MLIR [14], this provides the components needed to define

Constraint constructor	Description
type|*attr*	Match only the given type or attribute
typename|*attrname*	Match any type or attribute with the same base name
(*typename*|*attrname*)<pc_1, ..., pc_N>	Match any type or attribute with same base name, with parameter i satisfying pc_i

(a) Type and attribute constraints

Constraint constructor	Description
tc|*ac*	Match the type or attribute satisfying the constraint
(int8_t|uint8_t|int16_t| ...)	Match integers of a given bitwidth and signedness
int_literal : (*int8_t*|...)	Match exactly an integer literal with a given bitwidth and signedness
string	Match any string
string_literal	Match exactly a string literal
enumname	Match any constructor of a particular enum
enum_constructor	Match a particular enum constructor
array<*pc*>	Match an array that has all elements satisfying *pc*
[pc_1, ..., pc_N]	Match an array of N elements, where the i-th element is constrained by pc_i

(b) Parameter constraints

Constraint constructor	Description
!AnyType | #AnyAttr | AnyParam	Match any type, attribute, or parameter
AnyOf<c_1, ..., c_N>	Match any type, attribute, or parameter satisfying at least one c_i
And<c_1, ..., c_N>	Match any type, attribute, or parameter satisfying all c_i
Not<*c*>	Match any type, attribute, or parameter not satisfying *c*

(c) Generic constraint constructors

Figure 2. Only a few type of constraints are defined in IRDL. *c* refers to a type, attribute, or parameter constraint. *tc* refers to a type constraint, *ac* refers to an attribute constraint, and *pc* refers to a parameter constraint.

a simple pattern-based compilation flow (e.g., the optimization in Listing 1) without the need for additional C++ code. While more work is needed to define an entire transformation pipeline dynamically, the self-contained nature of IRDL already provides concrete benefits. The most immediate benefit is the concise, well-defined, and well-documented interface that IRDL provides, making it easy to understand the IR concepts compiler designers have at their disposal. IRDL also makes it easy to introspect and generate IRs, which provides foundations for the development of tooling around IR design, e.g. statistic and analysis tools or code completion when writing IR files. The ability to dynamically instantiate IRs could enable interesting research in programming languages, e.g. clang could generate IRs on the fly to represent and optimize domain-specific user-defined concepts. We implemented the majority of IRDL in close collaboration with the MLIR community and are now upstreaming our changes into the main MLIR repositories (details in Section 6.1). As a result, IRDL will be easily accessible to a wide set of users.

4 Declarative IR Specification with IRDL

IRDL enables developers to define dialects using a high-level description by providing a structured format to define types, attributes, and operations. As a result, defining dialects with IRDL is more efficient than defining them with a general-purpose programming language, following the usual benefits of a DSL [9]: definitions are concise, can be written faster, and can be analyzed for correctness and tool support. In this section, we will explore IRDL's different language constructs.

4.1 Dialect Definitions

Dialect definitions in IRDL start with a top-level `Dialect` statement whose body contains the type, attribute, alias, and operation definitions (Listing 3). In our example, the `cmath` dialect definition defines a parametric `complex` type, a `mul` operation, and a `norm` operation in its `Dialect` body.

4.2 Symbol Names and Namespaces

IRDL identifies dialects and their components by their name. References to types and type constraints are prefixed with `!`, and references to attributes and attribute constraints with `#`. Dialect definitions create a namespace, and thus references to a dialect component from outside the dialect must be prefixed with the dialect name. The prefix is optional when inside the dialect or for the `builtin` and `std` dialects. For example, `mul` uses abbreviated *type names* to refer to the `complex` type instead of the full `cmath.complex` name. `f32` is also a shorthand for `builtin.f32`, even though it is referred outside the `builtin` dialect.

```
Dialect cmath {
  Alias !FloatType = !AnyOf<!f32, !f64>

  Type complex {
    Parameters (elementType: !FloatType)

    Summary "A complex number"
  }

  Operation mul {
    ConstraintVar (!T: !complex<FloatType>)
    Operands (lhs: !T, rhs: !T)
    Results (res: !T)

    Format "$lhs, $rhs : $T.elementType"
    Summary "Multiply two complex numbers"
  }

  Operation norm {
    ConstraintVar (!T: !FloatType)
    Operands (c: !complex<!T>)
    Results (res: !T)

    Format "$c : $T"
    Summary "Compute the norm of a complex number"
  }
}
```

Listing 3. Self-contained IRDL specification of an IR dialect.

4.3 Constraints

Constraints define invariants on types, attributes, or type and attribute parameters. Type constraints specify invariants on operands and results in operation definitions. Furthermore, they can be used to define invariants on region arguments. Attribute constraints are used in operation definitions for specifying invariants on attributes, and parameter constraints are used in type and attribute definitions to specify invariants over parameters. Constraints are essential to represent operation, type, and attribute verifiers from a high-level description. Thus, IRDL provides a rich interface for defining and composing them (Figure 2).

Type and attribute constraints. The most general type and attribute constraints are `#AnyAttr` and `!AnyType` and are satisfied respectively by all types and all attributes. Any type or attribute reference where a constraint is expected is coerced into an equality constraint. For instance, `!f32` represents a constraint that is only satisfied by the `!f32` type. Similarly, constraints can also be nested directly in parameterized types. For instance, `!complex<!AnyType>` represents the constraint satisfied by any `complex` type. This can be shortened by only referring to the type name, `!complex`.

Parameter constraints. IRDL also provides support to define parameter constraints for built-ins, such as integers, floats, arrays, strings, and enums. Some constraints are provided for parameters expecting a specific parameter type. For instance, `string` corresponds to the constraint expecting a string, `int32_t` for a 32-bit signed integer, and `array` for any array. Furthermore, it is possible to constrain array contents by providing constraints that need to be satisfied by the elements. The constraint `array<!AnyType>` describes an array of types, and `[!AnyType]` describes an array containing exactly one element, which is a type. Constraints can also be defined for values. For instance, `"foo"` defines a constraint expecting the string literal `"foo"`, and `3 : int32_t` expects the value 3 encoded in a 32-bit signed integer.

Combining constraints. IRDL also provides ways to combine constraints. For instance, `AnyOf<!i64, !f64>` constrains a type to be either `!i64` or `!f64`. `Not` and `And` are used to express both the negation and the conjunction of constraints. For instance, `And<int32_t, Not<0 : int32_t>>` represents a constraint satisfied by any non-null integer.

```
Alias !Complexf32 = !complex<!f32>
Alias !ComplexOr<T> = AnyOf<!complex<!AnyType>, T>
```

Listing 4. IRDL aliases are used to shorten definitions.

```
Operation create_constant {
  Results (res: !complex<!f32>)
  Attributes (re: #f32_attr, im: #f32_attr)
  Summary "Create a constant complex number"
}
```

Listing 5. Attributes add static information to operations.

```
Operation log {
  Operands (c: !complex<!f32>, base: Optional<!f32>>)
  Results (res: !complex<!f32>)
}
```

Listing 6. Optional operands can encode a default parameter.

4.4 Type and Attribute Definitions

Besides the keyword, type and attribute definitions are identical in IRDL. The definitions are identified by a name and optionally specify a named parameter list that allows the type or attribute to carry static information. Each parameter is associated with a parameter constraint, representing the parameter invariants. For example, in our complex dialect, the `complex` type defines a single parameter, `elementType`, constrained to a floating-point type (Listing 3). Type and attribute definitions may also define a `Summary` field, which describes the operation for documentation purposes.

4.5 Aliases

IRDL allows the definition of aliases, which are shorthands for existing types, attributes, parameters, and constraints over them. We define two aliases in Listing 4. `Complexf32` is a shorthand for the `!complex<!f32>` type. `ComplexOr` is a parametric alias representing the constraint satisfied either

by a `complex` type with any parameter, or by the given type `T`. For instance, `!ComplexOr<f32>` is satisfied by `!f32`, or any other `complex` type.

4.6 Operation

Operations are defined using the `Operation` directive with a name. In their simplest form, operation definitions specify operands, results, and attributes, all defined with a named list of constraints. For example, the `create_constant` operation (Listing 5), which represents a constant complex number, expects no operands and one result of type `!complex<!f32>`. The operation additionally defines two attributes, `re` and `im`, representing the real and imaginary part of the complex number. They are both expected to be `#f32_attr`, a built-in attribute containing a single-precision floating-point value. Note that, similar to types and attributes, operations can optionally define a `Summary` field for documentation purposes.

Variadic operands and results. Having a non-fixed number of operands and results sometimes allows for more obvious abstractions. In IRDL, it is possible to specify that an operand or a result definition is variadic, meaning that the operand or result definition refers to multiple consecutive operands or results. This is done with the `Variadic` constraint, which can only be used as a top-level constraint in operand, result, and region argument definitions. Additionally, the `Optional` constraint specifies that an operand or a result is variadic and of size 0 or 1. For example, the logarithm base operand is optional in the `log` operation (Listing 6), meaning that the operation expects either 1 or 2 operands. Note that for the matching to be non-ambiguous, an attribute containing the size of the variadic operands and results is expected when `Operands` or `Results` contain more than one variadic definition.

Constraint variables. Operations often require operands or results to have the same type. The `ConstraintVars` directive allows users to define constraint variables, which are constraints that need to be satisfied by the same type at each use. For instance, in our leading example (Listing 3), the `norm` operation specifies a constraint variable, `T`, used in the operand and result definitions. It constrains the operand type parameter `c` to be equal to the result type, and constrains them both to be floating-point types.

Regions. Operations may also define multiple regions that are expected to be attached to the operation using the `Region` directive. Region definitions need to specify the argument constraints of the entry basic block. Additionally, region definitions may specify that a region should only be composed of a single basic block by providing a terminator instruction, which is expected to be the last instruction of the single block. For example, the `range_loop` operation (Listing 7), representing a loop iterating over an integer range, defines one region named `body`, which expects a single block, with a single `!i32` operand, and a `range_loop_terminator` terminator instruction.

```
Operation range_loop_terminator {}

Operation range_loop {
  Operands (lower_bound: !i32, upper_bound: !i32,
            step: !i32)
  Region body {
    Arguments (induction_variable: !i32)
    Terminator range_loop_terminator
  }
}
```

Listing 7. Regions can specify arguments and terminators.

```
Operation conditional_branch {
  Operands (condition: !i1)
  Successors (next_bb_true, next_bb_false)
}
```

Listing 8. Successors pass control to other basic blocks.

```
Enum signedness { Signless, Signed, Unsigned }

Type integer {
  Parameters (bitwidth: uint32_t, signed: signedness)
}

Alias signed_integer =
    !integer<uint32_t, signedness.Signed>
```

Listing 9. Enumerations are used in types or attributes.

Successors. Operation declarations may also specify a list of successor names using `Successors`, representing the list of basic blocks that a terminator operation may give control to. For example, the `conditional_branch` operation (Listing 8) defines two successors corresponding to the blocks that will get control depending on the value of `condition`. Defining a `Successors` field (even empty) will define an operation as a terminator, meaning that it can only be used as last operation in a basic block.

4.7 IR Formatting

IRDL supports the specification of parsers and printers for each defined `Type`, `Attribute`, and `Operation`, using MLIR's generic IR syntax. However, operations and types can define a custom declarative format to increase the readability and conciseness of the generated IR. For instance, `mul` (Listing 3) defines a format that will parse operations with the format `%res = cmath.mul %p, %q : f32`, where `%p`, `%q` and `%res` are of type `!cmath.complex<f32>` (Listing 3).

4.8 Enumerated Types

IRDL also provides basic support to define enums (enumerated types) that can be used as parameters of attributes

```
Constraint BoundedInteger : uint32_t {
  Summary "integer value between 0 and 32"
  CppConstraint "$_self <= 32"
}

Type BoundedVector {
  Parameters (typ: !AnyType, size: BoundedInteger)
}

Operation append_vector {
  ConstraintVars (T: !AnyType)
  Operands (lhs: Vector<T, BoundedInteger>,
            rhs: Vector<T, BoundedInteger>)
  Results (res: Vector<T, BoundedInteger>)

  CppConstraint "$_self.lhs().size() +
                 $_self.rhs().size() ==
                 $_self.res().size()"
}
```

Listing 10. IRDL-C++ allows the definition of additional invariants using inline C++.

and types. Enums are defined using the `Enum` directive, and expect a list of names that will define the different enum constructors. While enum names can be referred directly (or prefixed with the dialect namespace), enum constructors need to be prefixed by the enum's name. For instance, the `integer` type (Listing 9) has a `signed` parameter that expects a `signedness` enum. We can also define an alias for signed integers which constrains this parameter to be a `Signed` constructor. In this case, the `Signed` constructor needs to be referenced by `signedness.Signed`.

5 Augmenting Dialects with Generic C++

IRDL encourages a declarative specification of operations, types, attributes, and dialects. However, these definitions necessarily restrict the kind of operations IRDL can represent. To define more complex structures, IRDL is extended with IRDL-C++, which additionally provides directives for expressing invariants as generic C++ code.

5.1 Defining Constraints and Verifiers using C++

The `Constraint` directive can define arbitrary constraints from a C++ specification. `Constraint` is expected to be used when the declarative subset of IRDL is insufficient to express complex type, attribute, and parameter constraints. `Constraint` is composed of a base constraint that must be fulfilled and a `CppConstraint` directive that defines any additional constraints using generic C++ expressions. For instance, to define a bounded unsigned integer (Listing 10), we create a new constraint, `BoundedInteger`, with the base constraint `uint32_t`, and the C++ invariant checking that the uint32_t value is in the correct bounds. Note that the

```
TypeOrAttrParam StringParam {
  Summary "A string parameter"
  CppClassName "char*"
  CppParser "parseStringParam($self)"
  CppPrinter "printStringParam($self)"
}

Type StringAttr {
  Parameters (data: StringParam)
}
```

Listing 11. Declaration of types and attributes with C++

embedded C++ code refers to `$_self`, which refers to the actual parameter passed to the constraint. Finally, like `Type` or `Operation`, the `Summary` directive can be used to document the definition.

The `CppConstraint` directive can also be used in operation, type, and attribute definitions to define additional constraints specified in C++. Similar to constraint definitions, `$_self` refers to the operation that is being checked, and the embedded code can use accessors generated by IRDL-C++ to easily access members of the operation. For instance, we define an `append_vector` operation (Listing 10) which concatenates two vectors of known length. We need to use IRDL-C++ to represent the invariants of this operation since we need to check that the sum of the two operand vector sizes is equal to the result vector size. In the additional invariant specified in C++, we can directly access the `lhs` and `rhs` operand vectors using the generated accessors. Note that we access the `BoundedVector` parameter `size` with a similarly generated accessor.

5.2 Type and Attribute Parameters

In MLIR, type and attribute parameters are C++ classes and primitives holding arbitrary data. However, IRDL does not allow to define new parameters. IRDL-C++ allows the definition of new parameters with the `TypeOrAttrParam` directive by defining wrappers around C++ types. The directive requires a `CppClassName` field specifying the name of the C++ class to represent, as well as `CppParser` and `CppPrinter` fields, which expect C++ code used to parse the corresponding C++ type. The definition can also have a `Summary` field for documentation purposes. For instance, the `StringAttr` attribute (Listing 11) is defined with a single `StringParam` parameter, which is defined using the `TypeOrAttrParam` directive as a wrapper around a C++ `char*`.

6 Evaluation

To evaluate our work, we characterize SSA-based intermediate representations (IRs) across a wide range of application domains and analyze how well IRDL can express these IRs. As active participants in the MLIR community, we streamlined the design of IRs and their declarative nature in close

Table 1. MLIR today contains 28 different dialects covering programming languages and models, state machines for pattern matching, arithmetic over different mathematic domains, as well as instruction sets for CPUs and accelerators.

Affine Affine loops and memory operations	**NVVM** LLVM's IR for GPU compute kernels
AMX Intel's advanced matrix instruction set	**PDL** Rewrite pattern description language
Arith Arithmetic operations on integers and floats	**PDLInterp** The IR for a PDL interpreter
ArmSVE ARM's scalable vector instruction set	**Quant** Quantization
ArmNeon ARM's SIMD architecture extension	**ROCDL** AMD's IR for GPU compute kernels
Async Asynchronous execution	**SCF** Structured control flow, e.g. 'for" and 'if'
Builtin MLIR's builtin intermediate representation	**Shape** Shape inference
Complex Complex arithmetic	**SparseTensor** Sparse tensor computations
EmitC Printable C code	**SPIRV** Graphics shaders and compute kernels
GPU GPU abstraction	**Standard** Non domain-specific operations
LinAlg High-level linear algebra operations	**Tensor** Dense tensors computations
LLVMIR LLVM's intermediate representation in MLIR	**Tosa** Tensor operator set architecture
Math Scalar arithmetic beyond simple operations	**Vector** A generic vector abstraction
MemRef Multi-dimensional memory references	**X86Vector** The Intel x86 vector instruction set

collaboration with the community. After less than three years of public development, MLIR today offers a diverse set of IRs, which we will use as the foundation of our evaluation. We evaluate our work by using IRDL to express the full set of domain-specific IRs currently defined in MLIR's code repository. Using this data, we will answer the following three research questions:

1. Which IR designs have been developed in the context of MLIR and how did their development evolve over time? → Section 6.1
2. What characterizes the IRs in MLIR and are extensions beyond classical SSA widely used? → Section 6.2
3. Which IR features can be expressed purely in IRDL and how much general-purpose C++ code is still needed when defining IRs? → Section 6.3 and Section 6.4

Our evaluation will show that the MLIR ecosystem has grown a very diverse set of IRs. We will see that while most operations across IRs indeed heavily rely on the features of classical SSA, extensions such as variadic operands and return values as well as region statements are used across a vast number of IR dialects. Our evaluation will also show that only a limited number of types and attributes require the use of IRDL-C++ for either parameter definitions or additional C++ verifiers. Moreover, the majority of operations can define all of their operands, results, and attribute constraints in IRDL, and only 30% of all operations require an additional C++ verifier. Overall, our evaluation will provide evidence that an IR definition language such as IRDL in combination with a small amount of generic C++ code can facilitate the design of real-world compiler IRs.

6.1 The Evolution of IR Design in MLIR

We ground our evaluation on a representative set of IR dialects by analyzing the full set of production-focused IR dialects that the MLIR community has developed as part of MLIR. Just in the last 20 months,[1] the MLIR community grew its abstractions from 18 dialects and 444 operations defined in the public MLIR git repository to 28 dialects with 942 operations today (Figure 3), which more than doubled the number of operations on offer. In addition to the dialects developed in the main MLIR repository, IR dialects are increasingly developed as part of independent projects such as CIRCT [5], Flang, or Tensorflow [16]. Today, [2] MLIR defines 28 different dialects (Table 1) covering programming languages and models, state machines for pattern matching, arithmetic over different mathematic domains, as well as instruction sets for CPUs and accelerators. Across these dialects, MLIR defines 942 operations, 62 types, and 30 attributes. These dialects range from very small dialects with just 3 operations, e.g., arm_neon and builtin, to large dialects, e.g., LLVM and SPIR-V, with more than 100 operations each (Figure 4). The diversity in size, abstraction level, and application domain of these IRs exceeds the one of IRs in typical compilers, like LLVM and GCC. Taking the collaborative design process with a strong focus on production applications into account, we consider the resulting set of IRs as representative of typical compiler IRs today.

IRs, operations, types, and attributes in the MLIR ecosystem have traditionally been defined through various means. MLIR always provided a C++ interface for defining any MLIR object. Early on, the MLIR community used TableGen, a generic record format initially designed for the specification of LLVM backends, to simplify the specification of operations definitions. We developed the ideas in the paper together with the MLIR community and implemented our first

[1] analysing even earlier data has been prohibitively difficult

[2] Our data is based on commit 666accf283311c5110ae4e2e5e4c4b99078eed15 in the LLVM Project (https://github.com/llvm/llvm-project/) repository.

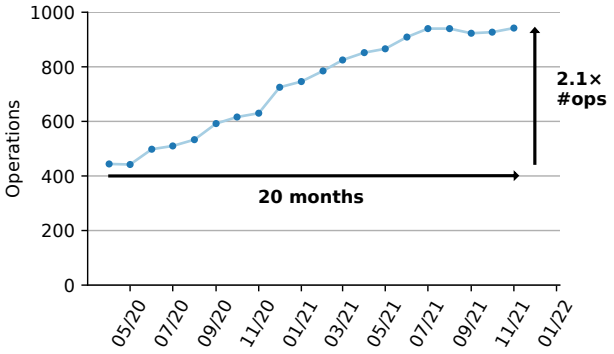


Figure 3. The number of operations defined in MLIR more than doubled in the 20 months since April 2020. The number of operations increased from initially 444 to 942 operations, which are today defined across 28 dialects. Increasingly more operations and dialects are nowadays defined in external projects, e.g. CIRCT, Flang, or Tensorflow.

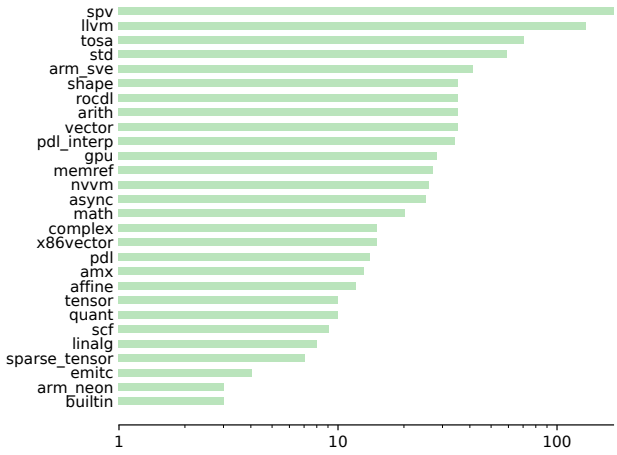


Figure 4. The number of operations in a given IR dialect ranges from 3 operations in the smallest dialects (arm_neon and builtin) to over hundered operations in large dialects such as LLVM and SPIR-V.

ideas by making the IR definitions in TableGen more declarative using the Operation Definition Specifications (ODS). As we reached the limitations of TableGen, we designed with IRDL a specialized domain-specific language for defining IRs. Thanks to this gradual development approach, the IR definitions expressed by the community were sufficiently structured for us to semi-automatically recover IRDL code from the generic, often TableGen-derived, C++ code that is today used in MLIR's production repositories to define IRs. We use the structured data obtained in the process of generating IRDL as the foundation for the following analysis, such that the data reported here is representative of the IR definitions in MLIR's production repositories.

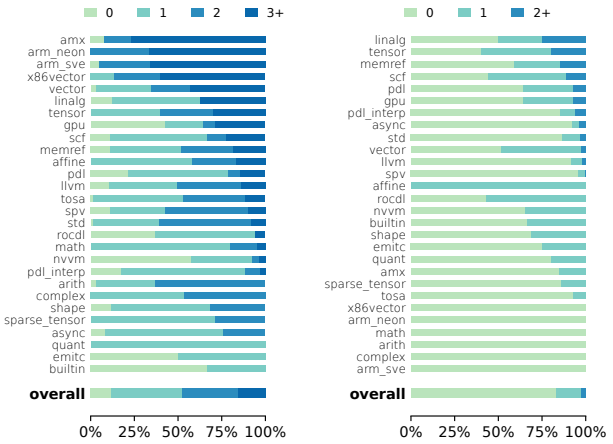
We implemented the majority of IRDL and are currently in the process of upstreaming this work to MLIR. The IRDL language is implemented with the exception of optional, variadic, and attribute constraints. Some minor syntax changes have also not yet been incorporated. We support the registration of dialects at runtime and are currently upstreaming the necessary support for dynamic dialects to MLIR. Support for generating C++ code has already been prototyped through the existance of TableGen and ODS. While not yet implemented, we expect that IRDL-C++ including support for generating C++ code and the corresponding parameter constraints will follow a similar software design and may even use some of the existing code. Overall, we aspire to contribute all IRDL into the public MLIR repositories.

When developing IRDL, we followed an open-source-first research model. In particular, we developed and upstreamed parts of our contributions into the public LLVM repositories even before submitting this publication for review. While researchers often perceive development activities that go beyond initial prototyping as costly and slow, we favor an open-source-first research approach. Detailed code reviews improve the quality and robustness of our implementation, yield better designed software, and ensure that the final software artifact satisfies community needs. We also believe that an open-source-first approach saves significant time as code is developed and adapted to community needs in a single step, instead of the two-step approach of prototyping followed by delayed upstreaming that typically requires complex and costly design updates. While the frontend parts of IRDL still need to go through broader reviews and community discussions that may further evolve our ideas, we expect that the code that is already available upstream will facilitate these discussions. With a variant of IRDL hopefully being available in MLIR in the near future, we hope for it to serve as a foundation for new and exciting open-source-first research projects.

6.2 Characteristics of IR Dialects

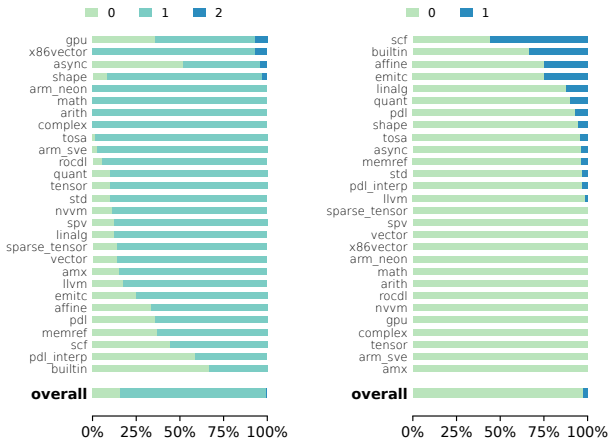
Our first objective is to understand the structure and characteristics of our IR dialects and the features required to express them. A detailed understanding of the requirements IR designers have has been critical to ensure IRDL covers the feature set required by a typical IR design.

Usage of operand definitions. We analyze the usage of operand definitions in the operations defined in MLIR (Figure 5). The majority of operations define between one to two operands (Figure 5a). Around 12% of operations have no operands, and 32% of operations have more than 3 operand definitions (up until 9 operand definitions). Interestingly, dialects defining a majority of operations with more than 3 operands, such as `amx`, `arm_neon`, `arm_sve`, and `x86vector`,



(a) Operands **(b)** Variadic Operands

Figure 5. Operations have typically zero (12%), one (41%), or two (32%) operands and rarely three or more (16%). The majority of operations are non-variadic (83%), but most (79%) dialects have at least one operation that uses variadic operands.



(a) Results **(b)** Variadic Results

Figure 6. Operations have either zero (16%) or one (84%) result definitions, and rarely two (1%). The large majority of operations are non-variadic (97%), but half of the dialects have at least one variadic result.

are all dialects targeting specific hardware for SIMD and matrix computations.

Furthermore, when looking at variadic definitions (Figure 5b), only 17% of operations define variadic operands. However, 79% of dialects have at least one operation with variadic operands, and almost half the dialects (46%) have more than 25% of their operations defining a variadic operand.

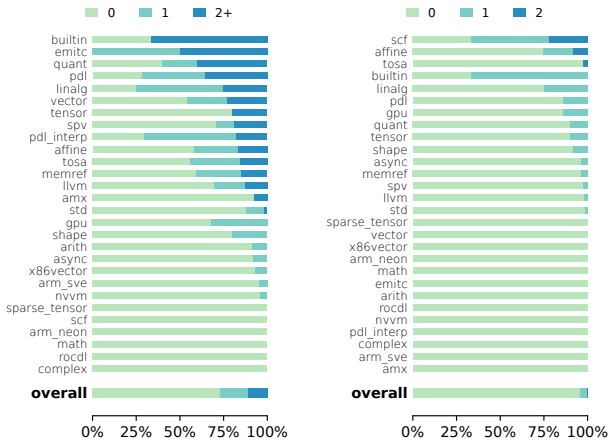
Usage of result definitions. We analyze the usage of result definitions in the operations defined in MLIR (Figure 6). Overall, 16% of operations define no results, and 84% of operations define 1 result. The remaining operations that define more than one result are defined by only 4 dialects: gpu, `x86vector`, `async`, and `shape`.

The usage of variadic result definitions (Figure 6b) yields that, contrary to variadic operands, no operations in MLIR define multiple variadic results. Only 3% of operations define a variadic result, though exactly half of the dialects define at least one operation with a variadic result.

Overall, this data shows that MLIR design choice of allowing multiple results, while not being used by many operations, is still used across multiple levels of abstractions.

Usage of attributes. We analyze the usage of attribute definitions in the operations defined in MLIR (Figure 7a). Around 73% of operation definitions do not define attributes, though around half the dialects (46%) have at least 25% of their operations defining an attribute.

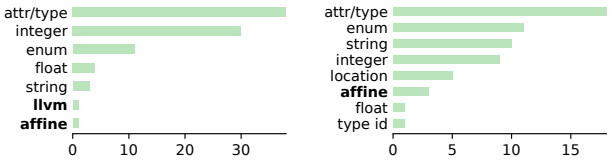
Usage of regions. We also provide data on the usage of region definitions in the operations defined in MLIR (Figure 6). Only 4% of operations define at least one region. However, around half of the dialects (54%) define at least one operation



(a) Attributes **(b)** Regions

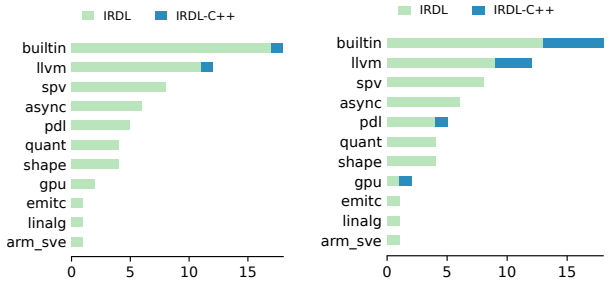
Figure 7. Operations have either zero (73%), one (16%), or more rarely more than two (11%) attributes. Most dialects (76%) define at least one operation using an attribute. Most operations define zero (96%), one (4%), or more rarely two (1%) regions. However, more than half the dialects (54%) have at least one operation that defines a region.

with a region definition. Note that the two dialects with more than half the operation defining a region are the `builtin` and `scf` dialects. In particular, `scf` represents structured control flow operations, explaining its high ratio of operation with regions.



(a) Types **(b)** Attributes

Figure 8. Only a few type and attribute parameters are domain-specific (3%). Domain-specific parameters (bold) are either from the LLVM or polyhedral (affine) dialect.



(a) Definitions **(b)** Verifiers

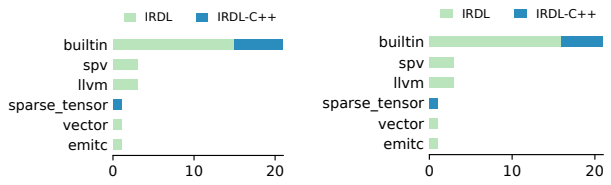
Figure 9. 97% of all type definitions exclusively use parameters defined in IRDL. Only a few types (16%) require an additional C++ verifier.

6.3 Expressiveness of Types and Attributes in IRDL

We gather data on type and attribute definitions in MLIR across all defined dialects, and summarise their expressiveness in IRDL. 14 out of the 28 dialects define either an attribute or a type, and in total, 62 types and 30 attributes are defined.

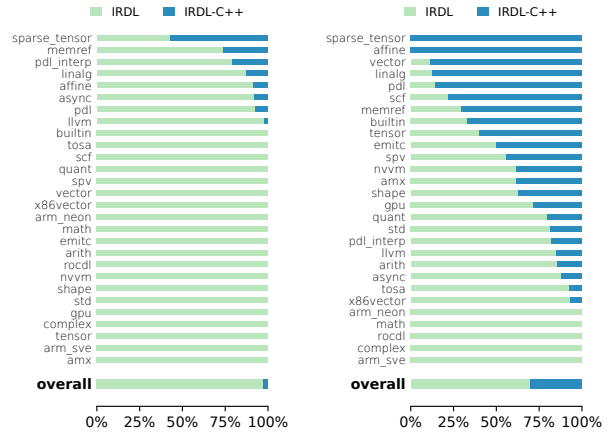
Type and attribute parameters. We first analyze the type and attribute parameters used in MLIR. In practice, only a limited set of parameters are used in types (Figure 9a) and attributes (Figure 10a), such as attributes, types, strings, integers, enums, locations representing a position in code, type ids used to uniquely identify C++ types. The remaining parameters are domain-specific and are only used in the `affine` and `llvm` dialect. Besides the domain-specific parameters, all these parameters are defined as builtins in IRDL. Thus, IRDL can represent the parameters of 97% of all types, respectively 77% of all attributes. Overall, all dialects, besides `llvm`, `builtin`, and `sparse_tensor`, can define the parameters of their types and attributes in IRDL, the aforementioned 3 dialects requiring the use of IRDL-C++ to define the parameters.

Verifiers. We provide data on the use of custom C++ verifiers on types (Figure 9b) and attributes (Figure 10b) defined



(a) Definitions **(b)** Verifiers

Figure 10. 77% of all attribute definitions exclusively use parameters defined in IRDL. Only a few atributes (20%) require an additional C++ verifier.



(a) Local constraints **(b)** Verifiers

Figure 11. The vast majority of operations (97%) can define their local constraints in IRDL, and only 30% of all operations require C++ for the verification of non-local constraints.

in MLIR. Overall, 16% of types and 20% of attributes define a C++ verifier, and over the 14 dialects defining a type or an attribute, only 5 of them require the use of IRDL-C++ for at least one type or attribute.

6.4 Expressiveness of Operations in IRDL

We inspect operations in MLIR and present data on the suitability of IRDL to represent them. Currently, MLIR defines in total 942 operations across its 28 dialects. Operation definitions in MLIR essentially consist of verifiers. Verifiers are commonly separated into local constraints and global constraints. Local constraints represent constraints over single operands, results, or attributes, e.g. restricting an operand to be of a `!IntegerType` type. On the other hand, global constraints represent constraints over multiple operands, constraints, or attributes. For instance, an operation may request that an attribute contains an integer with the same bitwidth as an operand. We separate our analysis between local constraints and global constraints.

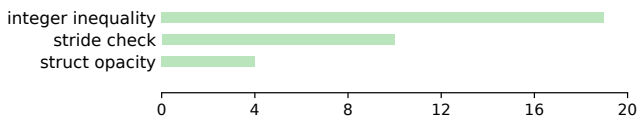


Figure 12. Only three kind of constraints used by operations require IRDL-C++ to be defined.

Local constraints on operands, results, and attributes. We provide data on local constraints in operation definitions (Figure 11a). IDRL defines local constraints with the `Operands`, `Result`, and `Attribute` directives. Overall, the vast majority of operations (97%) can represent all their local constraints in IRDL. If we look at the results per dialect, 20 out of the 28 dialects can represent all of their operation local constraints in IRDL.

We also provide data on the kind of local constraints present in MLIR operations that cannot be represented in IRDL. These constraints fall into 3 different categories (Figure 12). The first group constrains integer attributes to have values in a certain range, the second constrains memory accesses to be strided, and the third group constrains an LLVM struct to be opaque. Overall, in order to define all MLIR operations, only a small number of constraints have to be defined in IRDL-C++.

Global constraints. Finally, we analyze global constraints in MLIR operation definitions (Figure 11b). Global constraints are defined in IRDL-C++ with the `CppConstraint` directive, as well as the `ConstraintVars` directive in IRDL. Overall, 30% of operations in MLIR require a global constraint using IRDL-C++.

7 Related Work

Facilitating the design of compilers has long been recognized as an important objective. Previous work addressed this objective through several language-independent as well as embedded DSLs that facilitate compiler design. In comparison to previous work, IRDL permits the definition of IRs and their invariants using a language that is simultaneously (a) expressive enough to cover IRs ranging from high-level domain-specific concepts to machine-level instruction set descriptions, (b) sufficiently concise to ease development, and (c) structured enough to connect to external tools (Figure 13). In contrast, previous work either proposes DSLs that lack expressiveness to cater to the diverse needs of the MLIR community or have the required expressiveness but relies on general-purpose programming languages. As general-purpose languages are not tailored for IR and invariant definition, they are overly verbose, hard to connect to external tooling, and – from a purely practical perspective – unlikely to be used in a production toolchain such as LLVM.

Language-independent DSLs. Previous work on independent DSLs is compared to IRDL typically less expressive in the IRs that can be defined and rarely integrates support to define invariants on the defined IRs. Multiple projects have been developed with a focus on AST-based languages. For instance, Stratego/XT [1] provides a language to define ASTs, as well as strategies for traversing them, taking ideas from the ASF+SDF project [23]. JastAdd [8] follows similar ideas while relying on SableCC [6] to generate Java classes for each AST node, similarly to IRDL-C++. Nanopass [10], embedded in Scheme, additionally can reuse operations from other languages to define new languages. Finally, POET [27] focuses on manipulating generic ASTs fragments of other languages such as C. All those tools target an AST representation and have similar limitations in terms of expressiveness of invariants over the ASTs. For instance, they do not support parameterized types, and do not have builtin support for `AnyOf`, `And`, or `Not` constraints, or constraints with nested parameters. Instead, these tools expect compiler experts to write passes to define complex invariants, which is verbose and prevents a self-contained definition of such IRs.

Jetbrains MPS [24] also explored the definition of non-textual DSLs. MPS has similar expressiveness than the previously mentioned projects, but additionally can represent references to other operations (like in an SSA representation). MPS distinguishes itself by focusing on user experience in the Jetbrains IDE, by providing DSLs to define non-textual representations of programs, as well as DSLs to define actions that can be triggered in the Jetbrains IDE.

Embedded DSLs. Previous work also explored the definition of intermediate representations using DSLs embedded in other languages. Compared to language-independent DSLs, those languages often provide Turing-complete feature sets. While they provide a concise subset, their overall expressiveness comes from the use of a general-purpose programming language, like IRDL-C++. However, the concise subset of such DSLs is less expressive than IRDL.

Both Graal IR [4] and Delite [20] are extensible compilers that use the type system of their host languages (Java and Scala) to represent IR constraints. While Graal IR uses Java annotations to define IRs, Delite is built on top of Lightweight Modular Staging [18] and can be extended with Forge, a meta-DSL for auto-generating Delite DSL implementations [21] using a high-level specification. However, because of their deep integration in their respective language, the expressiveness of their DSL is limited by the type system of the host language, and thus have to rely more often on generic code compared to IRDL and IRDL-C++. While the defined IRs could be partially introspected via Java reflection, the complexity of the host language and its type system, and the lack of reflection support in Scala make introspection, from our perspective, impractical.

Framework	Representation	Embedding	Types: Singleton	Types: Parametric	Types: Values in Params.	Attributes	Builtin Constraints: Variadic	Builtin Constraints: Equality	Builtin Constraints: Nested Param.	Builtin Constraints: AnyOf	Builtin Constraints: And	Builtin Constraints: Not	Turing-completeness	Introspectable
IRDL	SSA + Regions	DSL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
IRDL-C++	SSA + Regions	DSL and C++	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗
Graal IR	Sea of nodes	Java	✓	✓	✗	✗	✓	✓	✓	✗	✗	✗	✓	?
Delite + Forge	Scala program	eDSL (Scala)	✓	✓	✗	✗	✓	✓	✓	✗	✗	✗	✓	?
Stratego/XT	AST	DSL	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓
JastAdd/SableCC	AST	DSL	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓
Jetbrains MPS	AST + References	DSL	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓
Nanopass	Scheme IR (AST)	eDSL (Scheme)	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓
Sham	Racket IR (AST)	eDSL (Racket)	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓
POET	AST	DSL	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Figure 13. IRDL, combined with IRDL-C++, has a better balance between expressiveness and conciseness than previous work.

Another approach to IR definition is explored by Racket [22]. Racket allows the extension of the host language syntax and semantics through the use of its macro system. Racket can be extended with Sham [25], which comes with a DSL to simplify the boilerplate need to define AST-based IRs. However, Sham is less expressive than IRDL and require the use of arbitrary Racket code to define complex invariants.

DSLs for compiler backends. Other related work also includes the description of IRs tailored for compiler target architectures. For example, GCC uses "Machine Description" [3], and LLVM uses TableGen [11] to specify multiple assembly target architectures. Other tools, such as ISDL [7], are not compiler-specific and aim at providing DSLs to define instruction sets, along with their timing information and usage of resources, that can easily be targeted by any compiler. Contrary to our work, these tools are tailored for low-level IRs and cannot express arbitrary high-level IRs. They are also deeply embedded into the respective tools and lack the verification and introspection capabilities of a DSL.

8 Conclusion

We present IRDL, a language to define IRs for SSA compilers with regions from a high-level description. We also present an extension of IRDL, IRDL-C++, for the description of IRs that require Turing-complete support. We demonstrate the suitability of IRDL to represent IRs by representing in IRDL and IRDL-C++ all dialects defined in the MLIR project and analyze various characteristics of these dialects. We expect that this work will fundamentally change compiler IR construction by facilitating the work of compiler designers as well as providing a well-defined interface on top of which powerful automation and external tooling can be developed.

Acknowledgements

Flexible and sustained industry support is key when aiming to evolve large open-source software projects in collaboration with their community. Hence, we would like to thank Xilinx, Inc. and Google LLC for supporting this research in parts. We also acknowledge the MLIR developer community, which offers with MLIR and all its dialects a powerful compilation framework and a supportive collaboration environment. We thank this community for fueling our research through inspiring ideas, open discussions, excellent code, and plenty of code reviews. We look forward to continuing to work with the MLIR community and broadening the discussion on how a variant of IRDL may provide concrete benefits to some workflows in the MLIR ecosystem.

References

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.* 72, 1–2 (June 2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

[2] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35. https://doi.org/10.1145/75277.75280

[3] Sameera Deshpande and Uday P Khedker. 2007. Incremental machine descriptions for GCC. In *GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems*.

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

[5] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as Hardware Compiler Infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*.

[6] Etienne M Gagnon and Laurie J Hendren. 1998. *SableCC, an object-oriented compiler framework*. IEEE.

[7] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. 1997. ISDL: An instruction set description language for retargetability. In *Proceedings of the 34th annual Design automation conference*. 299–302. https://doi.org/10.1145/266021.266108

[8] Görel Hedin and Eva Magnusson. 2001. JastAdd—a Java-based system for implementing front ends. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 59–78. https://doi.org/10.1016/S1571-0661(04)80920-4

[9] Paul Hudak. 1997. Domain-specific languages. *Handbook of programming languages* 3, 39-60 (1997), 21.

[10] Andrew W Keep and R Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 343–350. https://doi.org/10.1145/2544174.2500618

[11] Jozef Kolek, Zoran Jovanović, Nenad Šljivić, and Dragan Narančić. 2013. Adding microMIPS backend to the LLVM compiler infrastructure. In *2013 21st Telecommunications Forum Telfor (TELFOR)*. IEEE, 1015–1018. https://doi.org/10.1109/TELFOR.2013.6716404

[12] Anton Korobeynikov. 2009. Tutorial: Building a backend in 24 hours. In *LLVM Developer's Meeting, Cupertino, CA, USA*.

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[14] 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. https://doi.org/10.1109/CGO51591.2021.9370308

[15] Microsoft. 2021. *Official page for Language Server Protocol*. Retrieved March 18, 2022 from https://microsoft.github.io/language-server-protocol/

[16] Jacques Pienaar. 2020. MLIR in TensorFlow ecosystem. (2020).

[17] LLVM Project. 2017. Swift Documentation.

[18] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 127–136. https://doi.org/10.1145/1868294.1868314

[19] Yulei Sui. 2010. Open64 introduction.

[20] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25. https://doi.org/10.1145/2584665

[21] Arvind K Sujeeth, Austin Gibbons, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: generating a high performance DSL implementation from a declarative specification. *Acm Sigplan Notices* 49, 3 (2013), 145–154. https://doi.org/10.1145/2517208.2517220

[22] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 132–141. https://doi.org/10.1145/1993498.1993514

[23] Mark GJ van den Brand, Arie van Deursen, Jan Heering, Hayco A de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A Olivier, Jeroen Scheerder, et al. 2001. The ASF+ SDF meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 3–8.

[24] Markus Voelter and Vaclav Pech. 2012. Language modularity with the MPS language workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1449–1450. https://doi.org/10.1109/ICSE.2012.6227070

[25] Rajan Walia, Chung-chieh Shan, and Sam Tobin-Hochstadt. 2020. Sham: A DSL for Fast DSLs. *arXiv preprint arXiv:2005.09028* (2020). https://doi.org/10.22152/programming-journal.org/2022/6/4

[26] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011), 633–652. https://doi.org/10.1016/j.parco.2011.05.005

[27] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370637