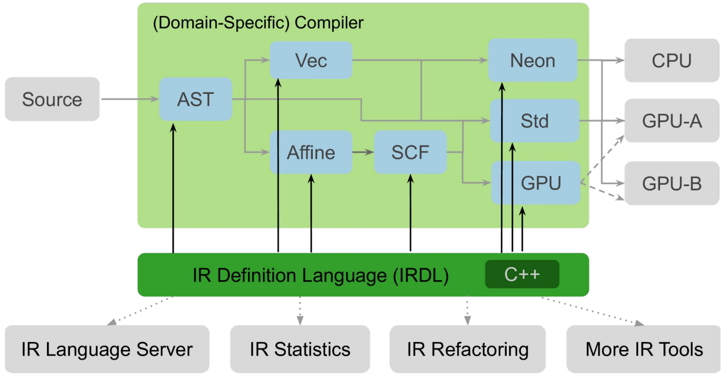


IRDL: An IR Definition Language for SSA Compilers

Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, ***Tobias Grosser***

University of Edinburgh, *Google, Modular.AI, ETH Zurich*

IRs: The New Gold of Computer Systems

IR: Intermediate Representation

$$|p \cdot q|$$

IR: Intermediate Representation

$$|p \cdot q|$$

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

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

    return %conorm : !std.f32
}
```

IR: Intermediate Representation

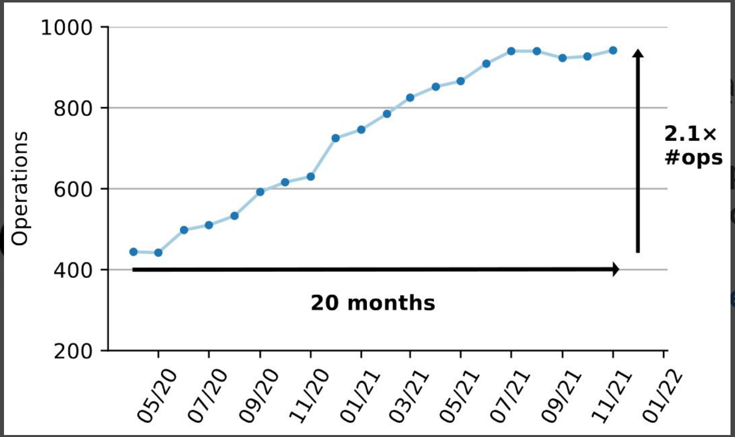
$$|p \cdot q|$$

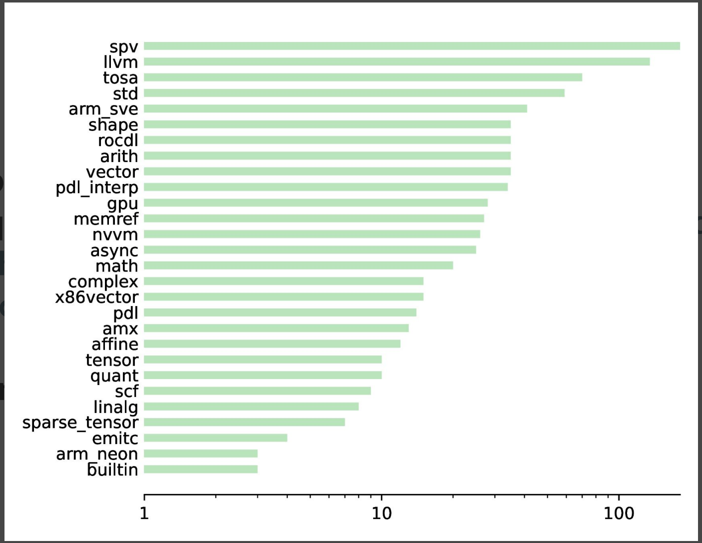
```
func @conorm(%p: !cmath.complex<std.f32>,
             %q: !cmath.complex<std.f32>) -> !std.f32 {

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

    return %conorm : !std.f32
}
```

easy to analyze, transform and output

Intermediate Representation

```
conorm(%p
        %q                                d.f32 {
pq = cmat
conorm = c

eturn %co
```





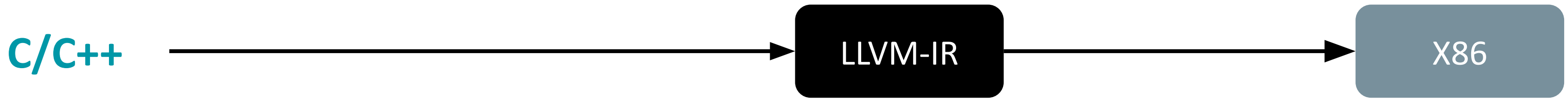
in 2-3 years, ~900 operations belonging to ~30 abstractions

MLIR

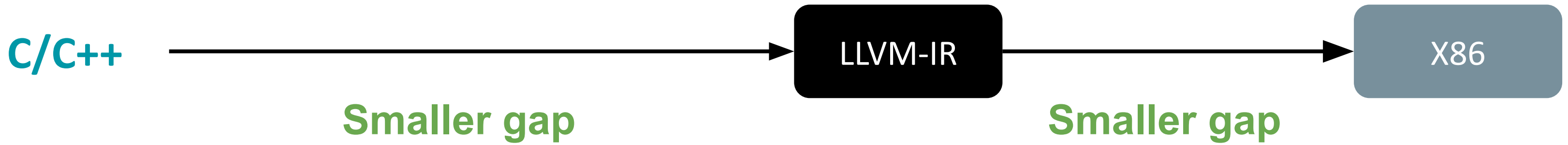
Old Compiler Pipelines

Old Compiler Pipelines

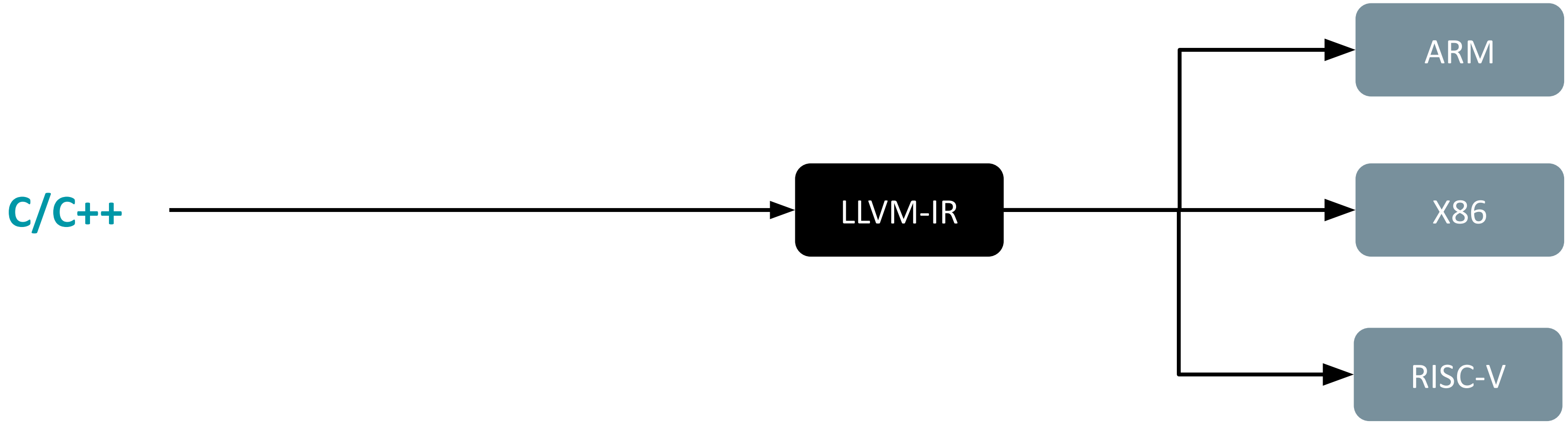
Traditional Compiler Pipelines



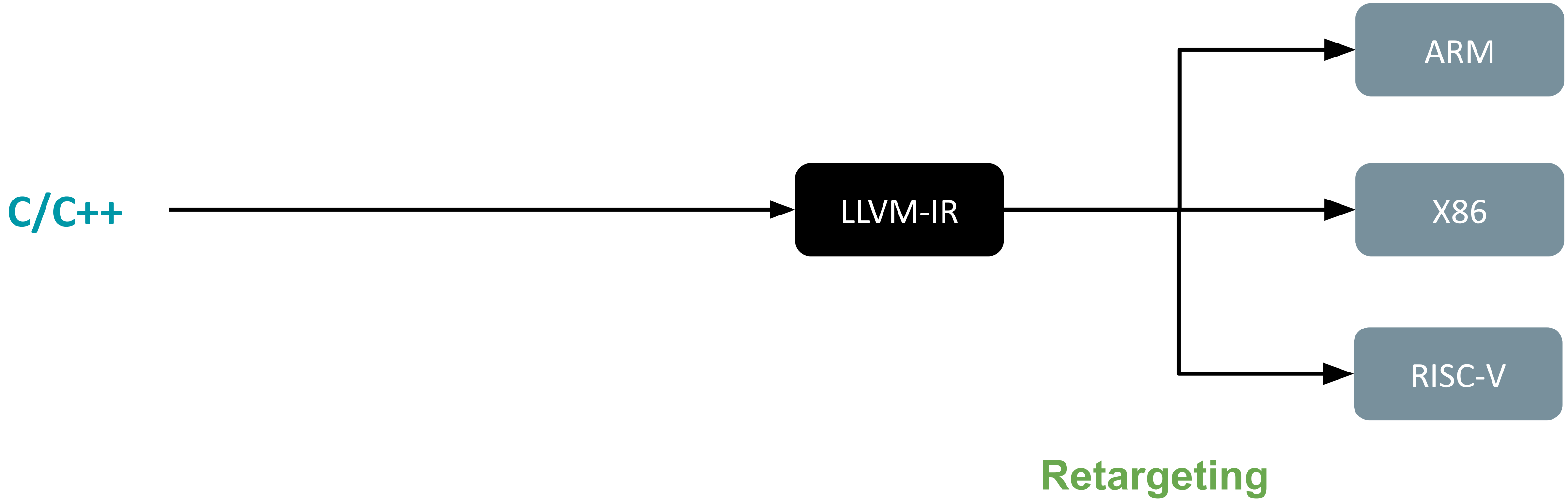
Traditional Compiler Pipelines



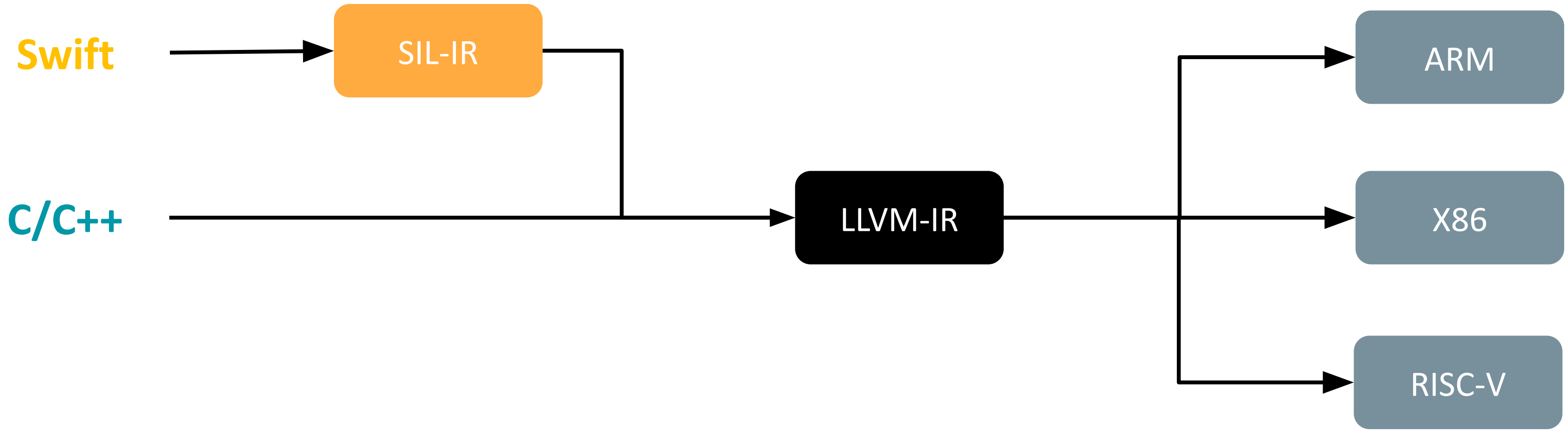
Traditional Compiler Pipelines



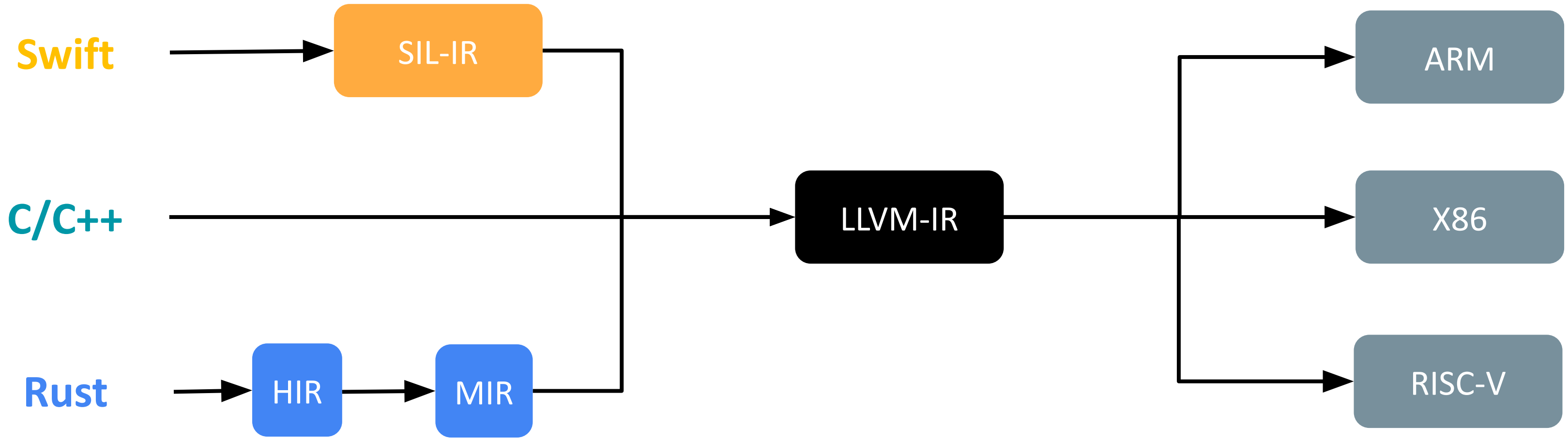
Traditional Compiler Pipelines



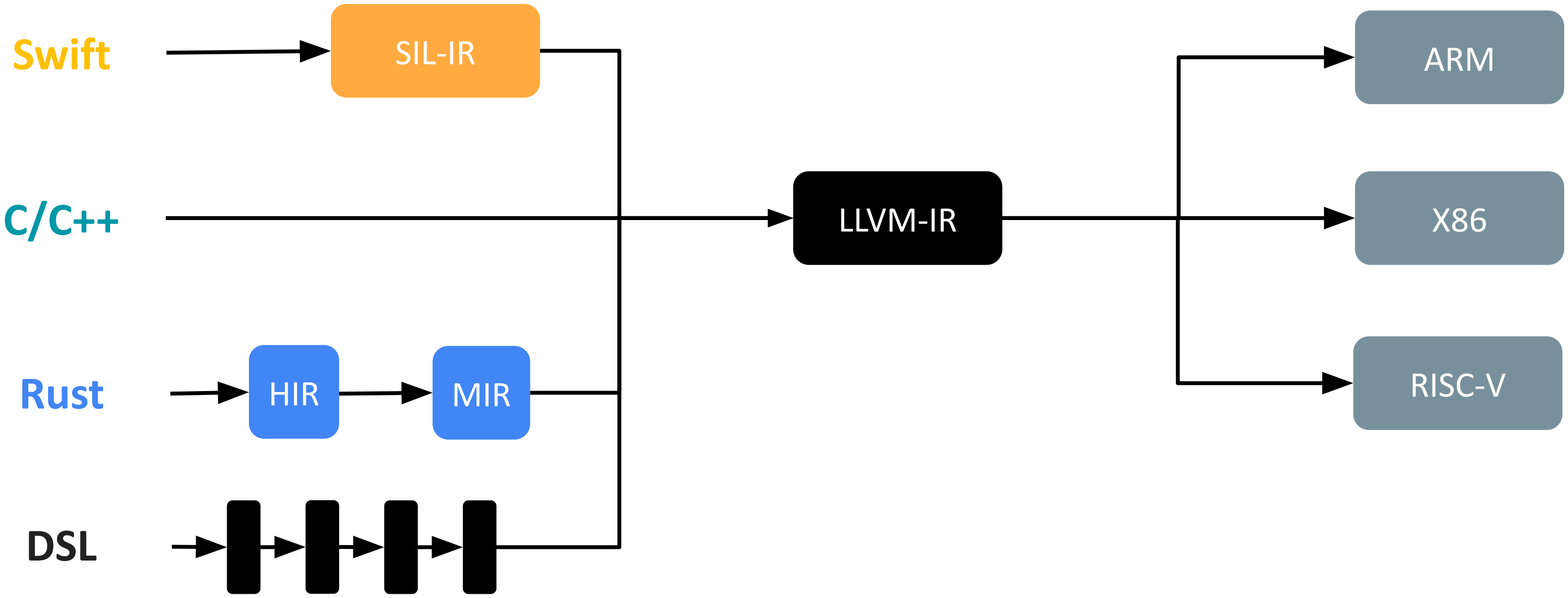
Recent Compiler Pipelines



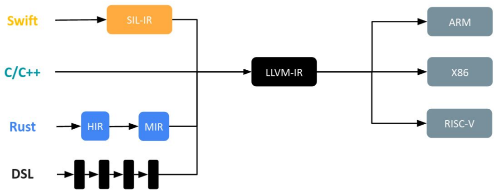
Recent Compiler Pipelines



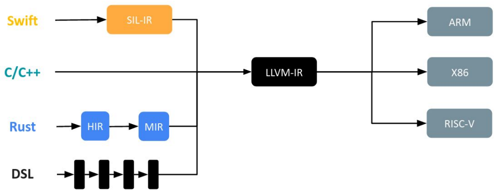
Recent Compiler Pipelines

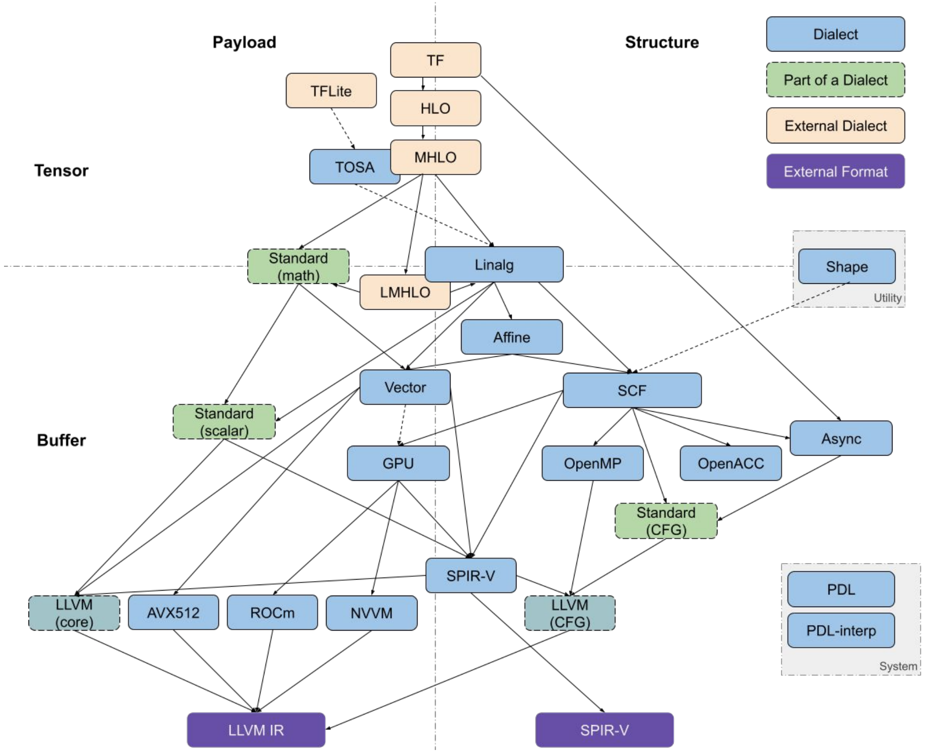


MLIR Compiler Pipeline



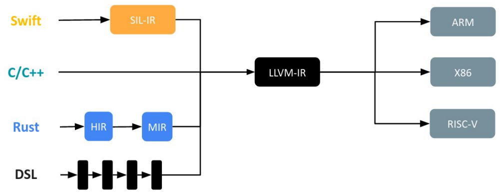
MLIR Compiler Pipeline

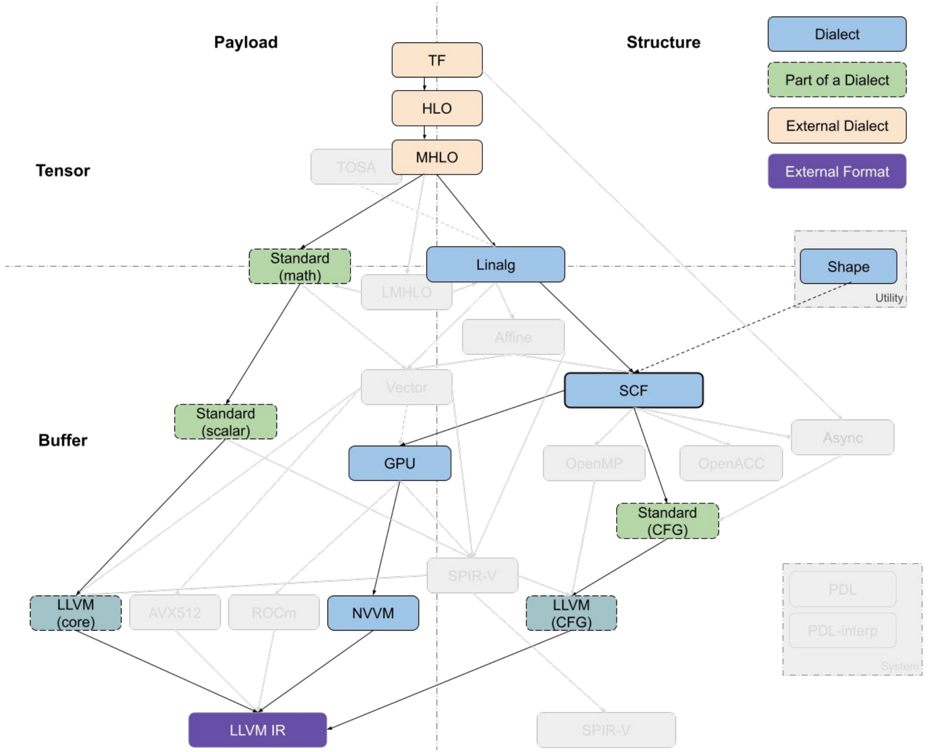




from Alex Zinenko

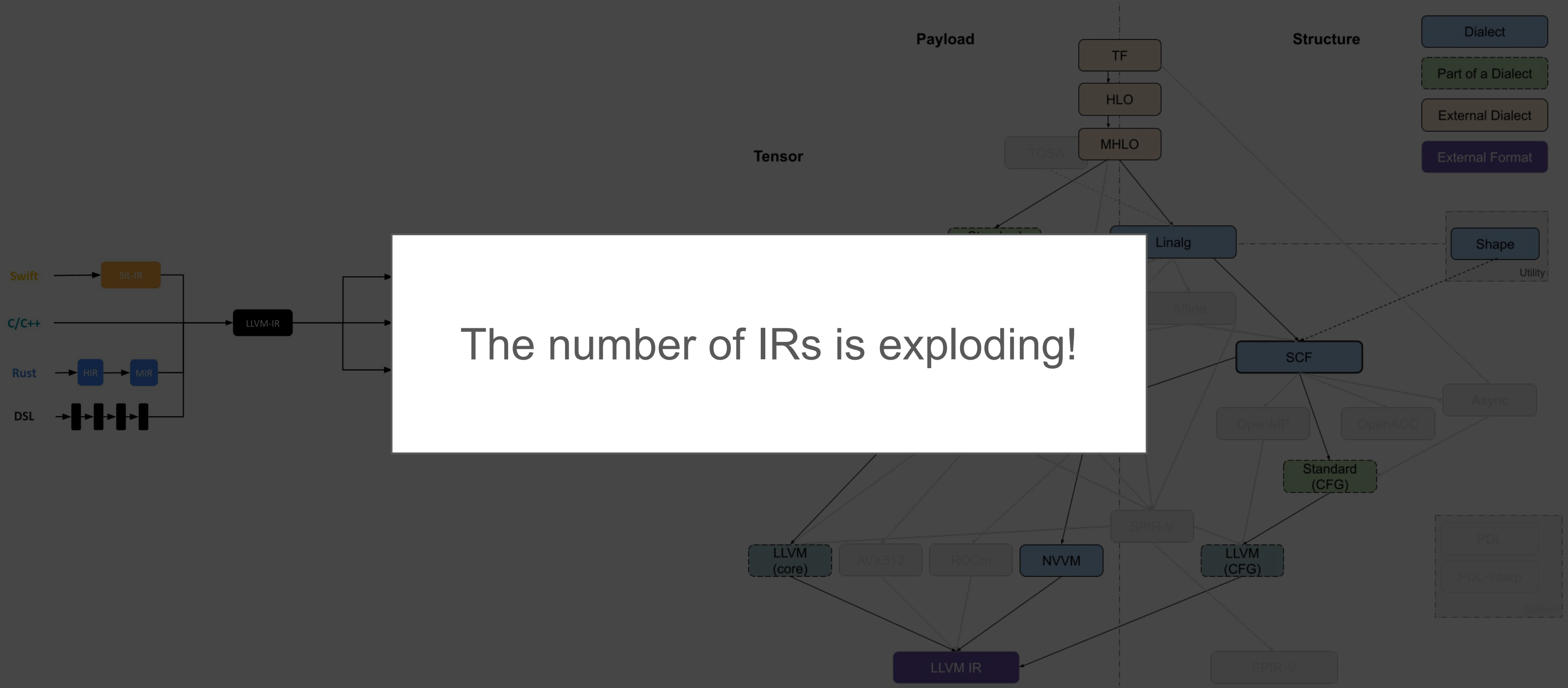
MLIR Compiler Pipeline





from Alex Zinenko

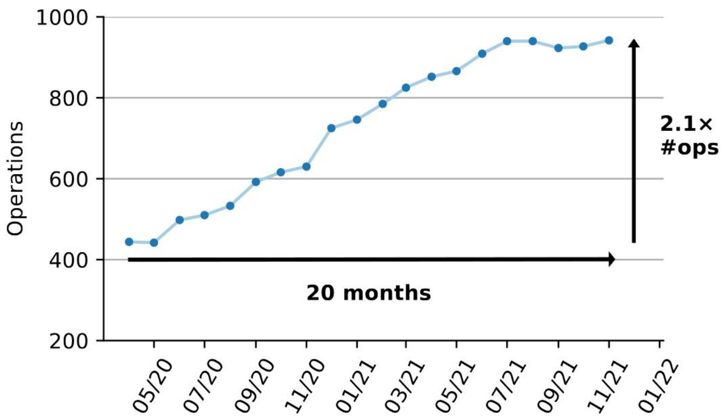
MLIR Compiler Pipeline

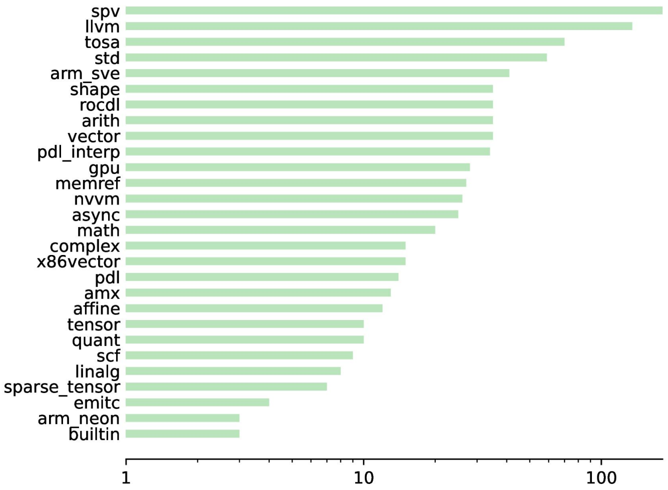


The number of IRs is exploding!

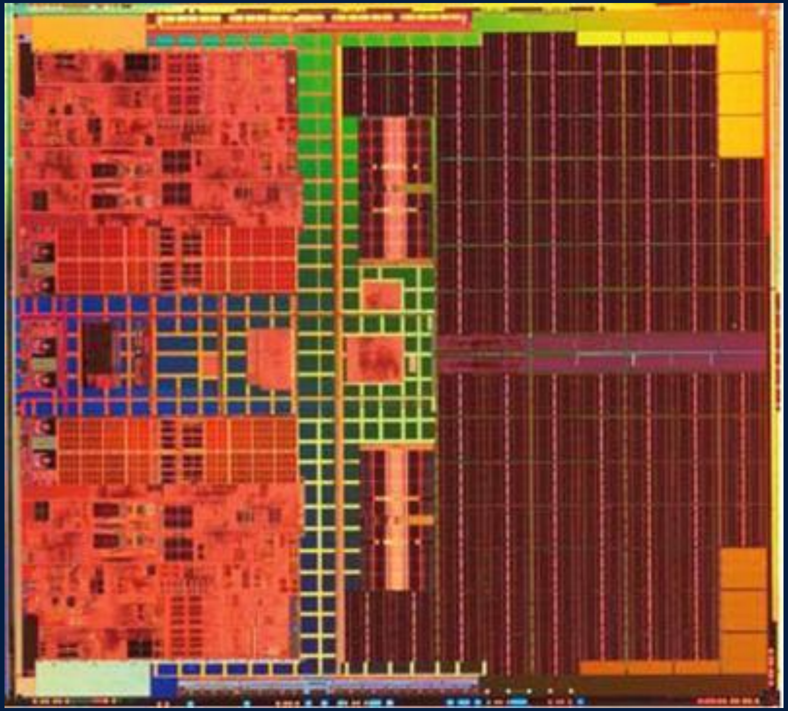
from Alex Zinenko

IRs in MLIR



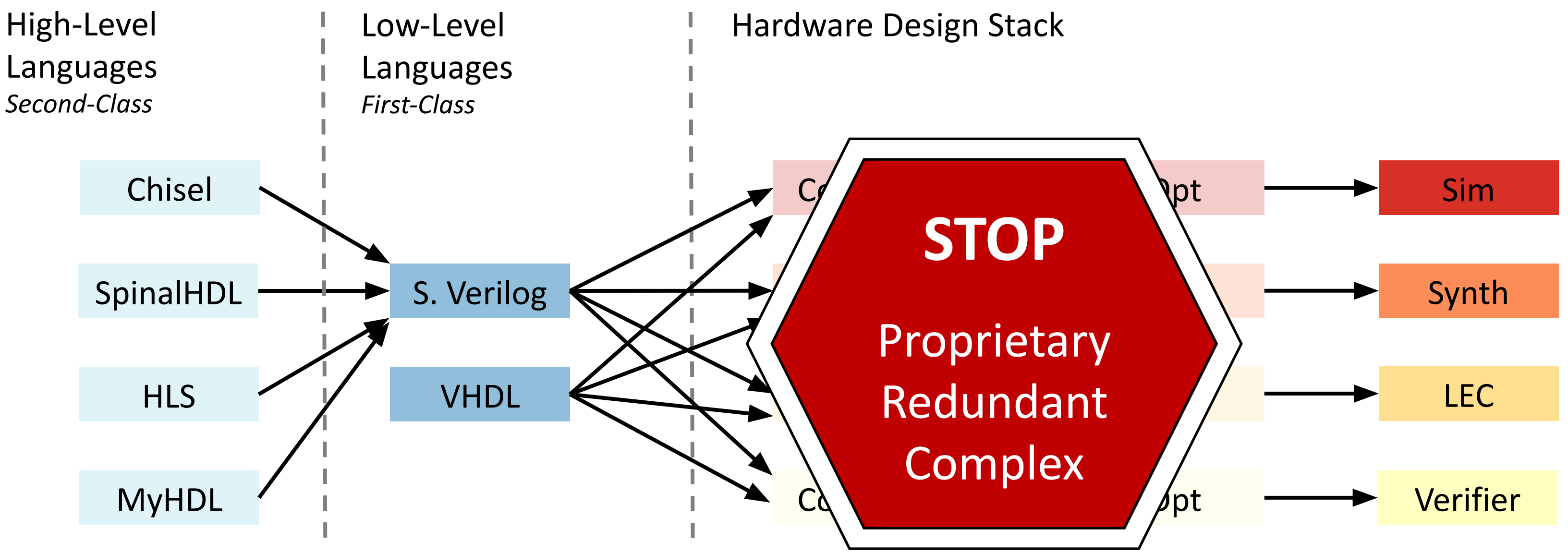


in 2-3 years, ~900 operations belonging to ~30 abstractions

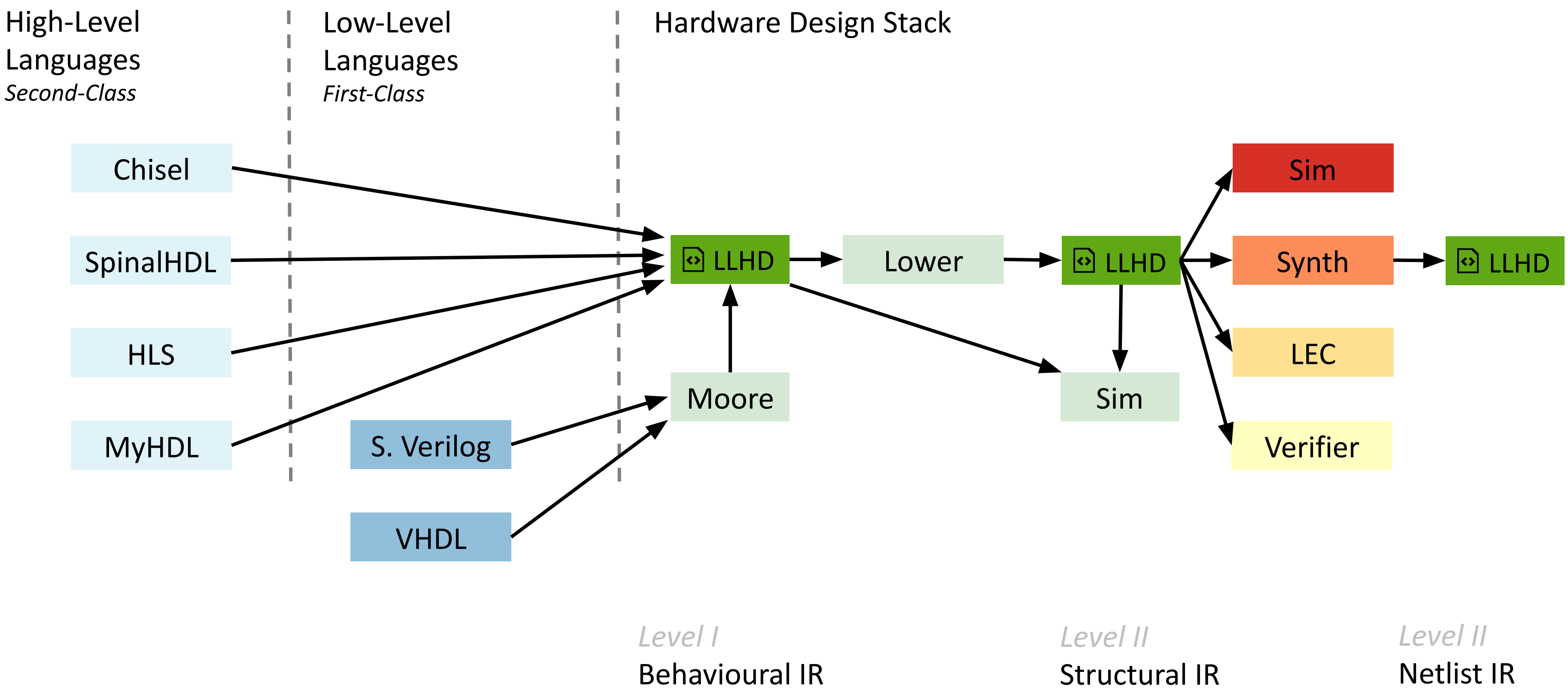


LLHD: A Multi-Level IR for Hardware Design

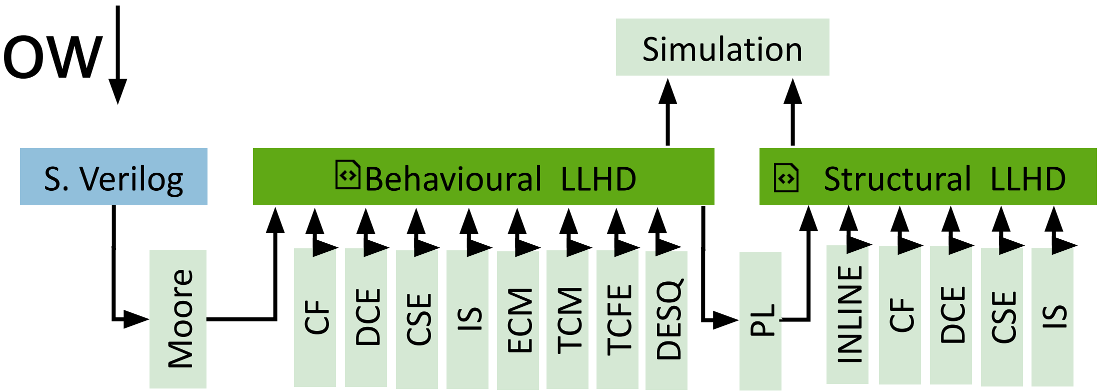
Redundancy in Hardware Design Languages



LLHD: A Multi-Level IR for Hardware Design



LLHD: Compilation Flow




```
module acc (input clk, input [31:0] x, input en, output [31:0] q);
  bit [31:0] d, q;
  always_ff @(posedge clk) q <= #1ns d;
  always_comb begin
    d <= #2ns q;
    if (en) d <= #2ns q+x;
  end
endmodule
```

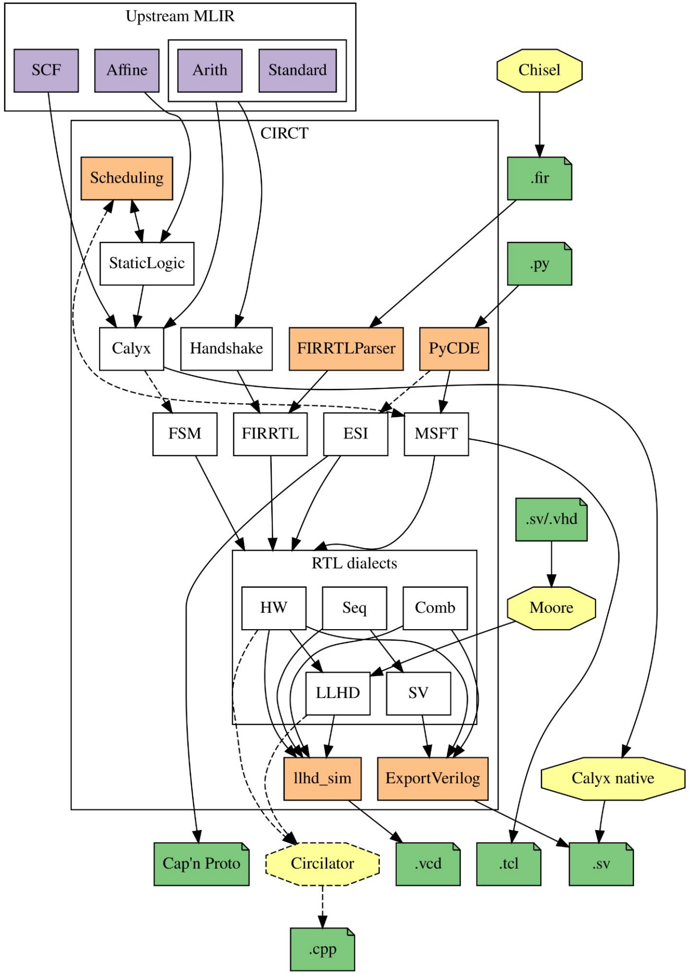
```
entity @acc (i1$ %clk, i32$ %x, i1$ %en)
    -> (i32$ %q) {
   %clkp = prb i1$ %clk
   %qp = prb i32$ %q
   %xp = prb i32$ %x
   %enp = prb i1$ %en
   %sum = add i32 %qp, %xp
   reg i32$ %q, %sum rise %clkp if %enp
}
```

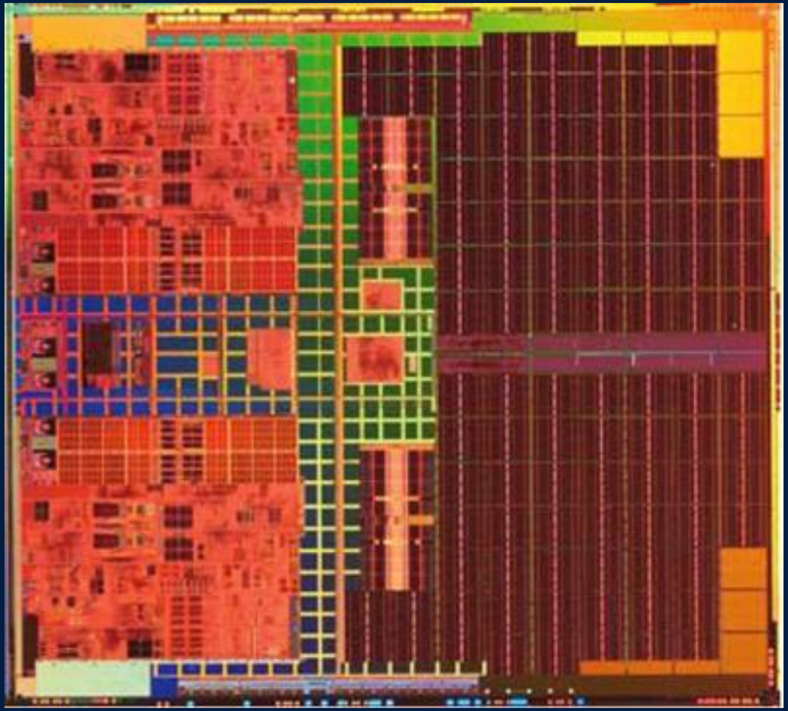
```
entity @acc_ff (i1$ %clk, i32$ %d) -> (i32$ %q)
{
   %delay = const time 1ns
   %clkp = prb i1$ %clk
   %dp = prb i32$ %d
   reg i32$ %q, %dp rise %clkp after %delay
}
```

```
entity @acc_comb (i32$ %q, i32$ %x, i1$ %en) -> (i32$
%d) {
   %qp = prb i32$ %q
   %xp = prb i32$ %x
   %enp = prb i1$ %en
   %sum = add i32 %qp, %xp
   %delay = const time 2ns
   %dns = [%gp, %sum]
   %dn = mux i32 %dns, %enp
   drv i32$ %d, %dn after %delay
}
```

```
    wait %entry for %q, %x, %en
  }
```

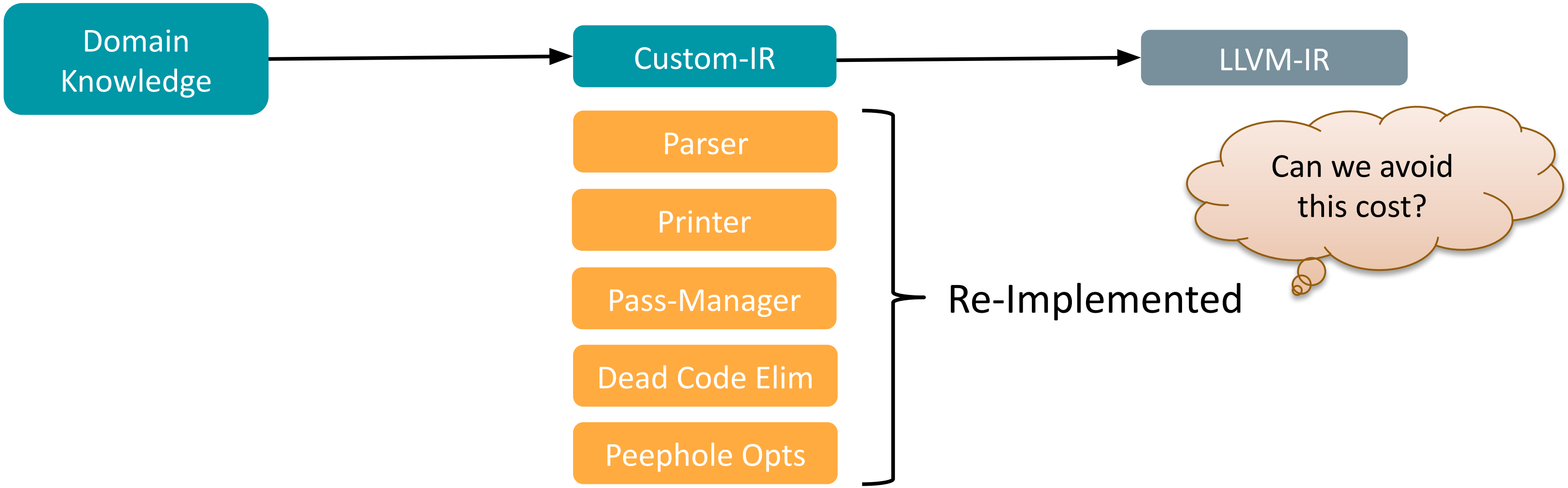
CIRCT: More Dialects



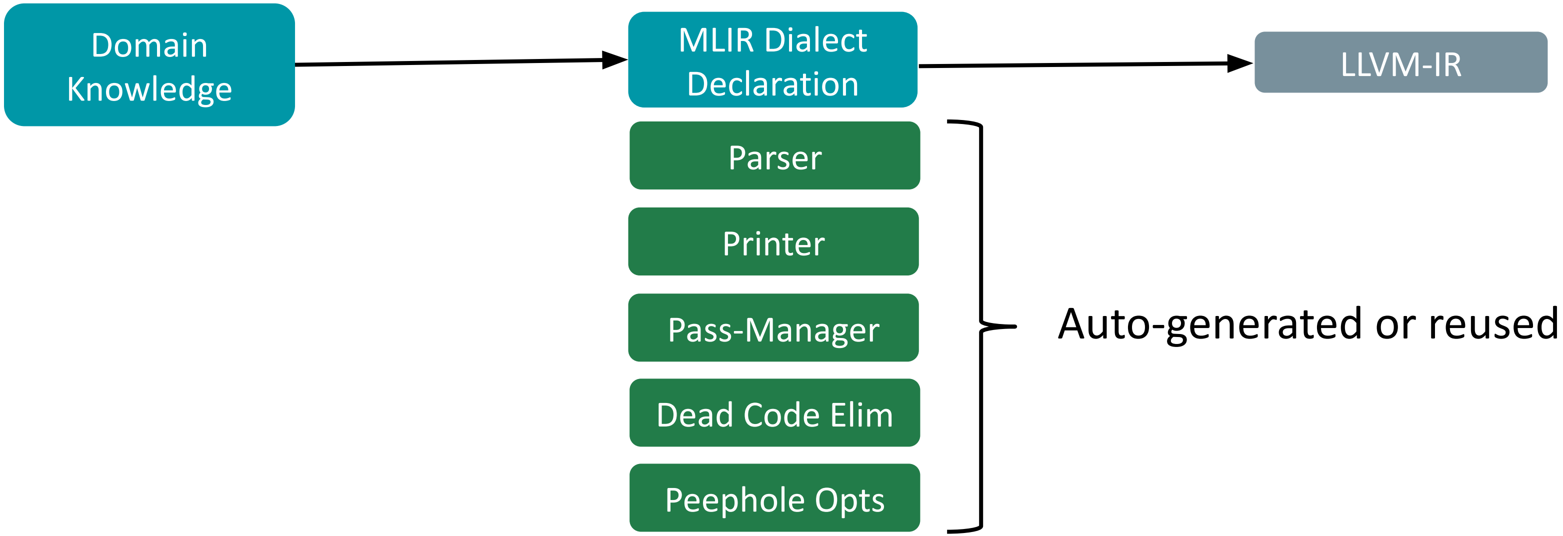


How Can I Build IRs?

The Usual Cost of Building a New IR



Building a Modern Compiler IR – using MLIR



Defining an IR in MLIR: Easy or Painful?

C++ Template Meta Programming

TableGen

Build Systems

LLVM Terminology

Compiler Experts:
Defining IRs in MLIR is easy!
vs. gcc, LLVM backends, …

Everybody Else:
Defining IRs in MLIR is confusing!
vs. Python, Domain-Specific Languages, …

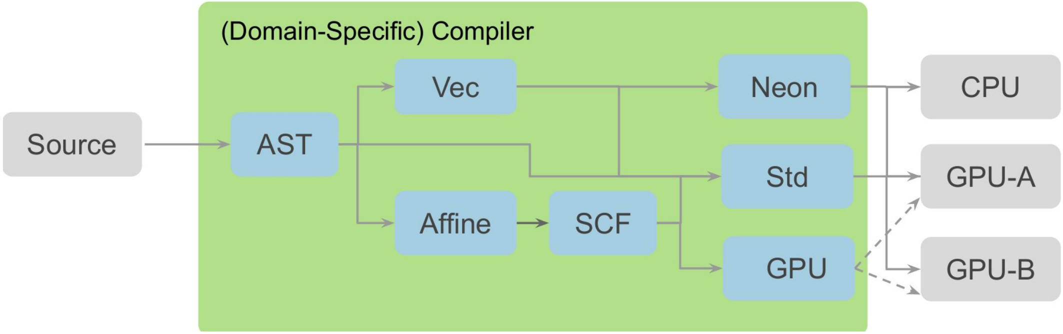
Defining an IR in MLIR: Easy or Painful?

C++ Template Meta Programming

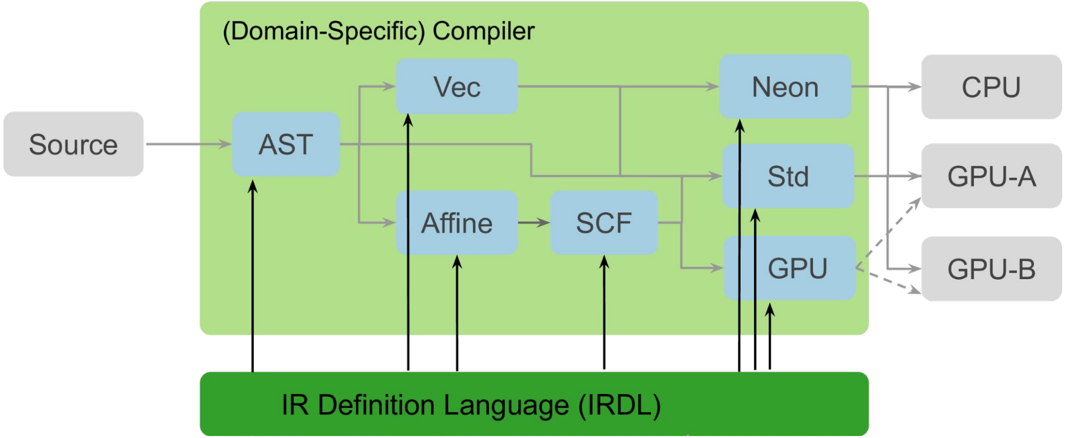
```
class XOrIOpAdaptor {
public:
  XOrIOpAdaptor(::mlir::ValueRange values, ::mlir::DictionaryAttr attrs = nullptr, ::mlir::RegionRange regions = {});
  XOrIOpAdaptor(XOrIOp&op);
  ::mlir::ValueRange getOperands();
  std::pair<unsigned, unsigned> getODSOperandIndexAndLength(unsigned index);
  ::mlir::ValueRange getODSOperands(unsigned index);
  ::mlir::Value lhs();
  ::mlir::Value rhs();
  ::mlir::DictionaryAttr getAttributes();
  ::mlir::LogicalResult verify(::mlir::Location loc);

private:
  ::mlir::ValueRange odsOperands;
  ::mlir::DictionaryAttr odsAttrs;
  ::mlir::RegionRange odsRegions;
};
class XOrIOp : public ::mlir::Op<XOrIOp, ::mlir::OpTrait::ZeroRegion, ::mlir::OpTrait::OneResult, ::mlir::OpTrait::OneTypedResult<::mlir::Type>::Impl, ::mlir::OpTrait::ZeroSuccessor, ::mlir::OpTrait::NOperands<2>::Impl, ::
::mlir::OpTrait::SameOperandsAndResultType, ::mlir::MemoryEffectOpInterface::Trait, ::mlir::VectorUnrollOpInterface::Trait, ::mlir::OpTrait::Elementwise, ::mlir::OpTrait::Scalarizable, ::mlir::OpTrait::Vectorizable, ::mlir
public:
  using Op::Op;
  using Op::print;
  using Adaptor = XOrIOpAdaptor;
  static ::llvm::ArrayRef<::llvm::StringRef> getAttributeNames() {
		return {};
  }
  static constexpr ::llvm::StringLiteral getOperationName() {
		return ::llvm::StringLiteral("arith.xori");
  }
  std::pair<unsigned, unsigned> getODSOperandIndexAndLength(unsigned index);
  ::mlir::Operation::operand_range getODSOperands(unsigned index);
  ::mlir::Value lhs();
  ::mlir::Value rhs();
  ::mlir::MutableOperandRange lhsMutable();
  ::mlir::MutableOperandRange rhsMutable();
  std::pair<unsigned, unsigned> getODSResultIndexAndLength(unsigned index);
  ::mlir::Operation::result_range getODSResults(unsigned index);
  ::mlir::Value result();
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::Type result, ::mlir::Value lhs, ::mlir::Value rhs);
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::TypeRange resultTypes, ::mlir::Value lhs, ::mlir::Value rhs);
  static void build(::mlir::OpBuilder &, ::mlir::OperationState &odsState, ::mlir::TypeRange resultTypes, ::mlir::ValueRange operands, ::llvm::ArrayRef<::mlir::NamedAttribute> attributes = {});
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::Value lhs, ::mlir::Value rhs);
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::ValueRange operands, ::llvm::ArrayRef<::mlir::NamedAttribute> attributes = {});
  ::mlir::LogicalResult verify();
  static void getCanonicalizationPatterns(::mlir::RewritePatternSet &results, ::mlir::MLIRContext *context);
  ::mlir::OpFoldResult fold(::llvm::ArrayRef<::mlir::Attribute> operands);
  static ::mlir::ParseResult parse(::mlir::OpAsmParser &parser, ::mlir::OperationState &result);
  void print(::mlir::OpAsmPrinter &p);
  void getEffects(::mlir::SmallVectorImpl<::mlir::SideEffects::EffectInstance<::mlir::MemoryEffects::Effect>> &effects);
};
```

IRDL: An IR Definition Language

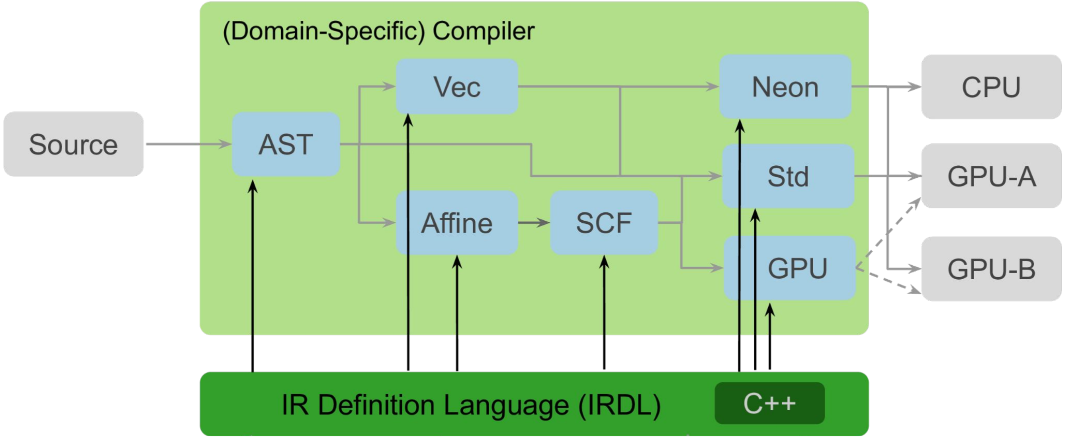
The IR Definition Language



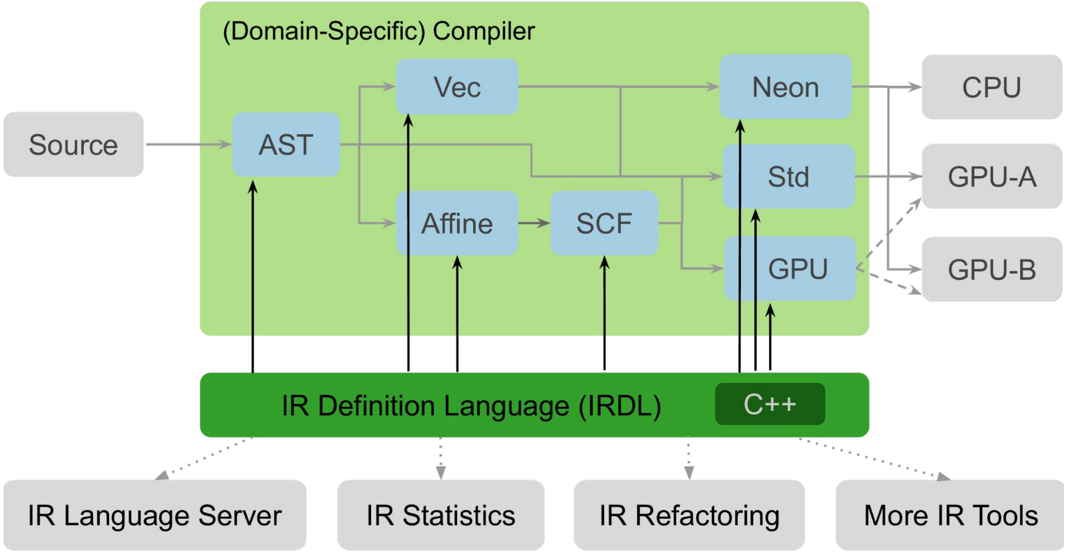
The IR Definition Language



The IR Definition Language



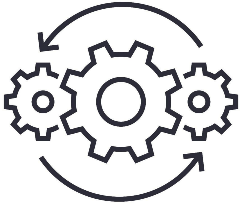
The IR Definition Language



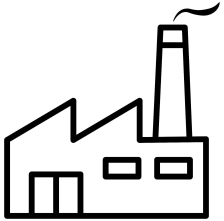
IRDL Objectives

Concise

Introspectable



Dynamic



Generable

MLIR: A unified SSA representation of IRs

```
%res = std.add(%lhs, %rhs) : (!std.i32, !std.i32) -> !std.i32
```

- Results
- Operation and dialect name
- Operands
- Operation type

MLIR: A unified SSA representation of IRs

```
%res = std.constant() { value = 42 } : () -> !std.i32
```

Attributes

MLIR: A unified SSA representation of IRs

```
%res = std.constant() : () -> !std.i32
```

```
std.constant verifier:
Missing attribute "value"
```

MLIR: A unified SSA representation of IRs

```
%res = std.constant() : () -> !std.i32
```

Verifiers check IR Invariants!

```
std.constant verifier:
Missing attribute "value"
```

Example: A Complex-Number Dialect

Before optimization:

$$|p| \cdot |q|$$

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

After optimization:

$$|p \cdot q|$$

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

Example: A Complex-Number Dialect

Before optimization:

$$|p| \cdot |q|$$

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

After optimization:

$$|p \cdot q|$$

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

Example: A Complex-Number Dialect

Before optimization:

$$|p| \cdot |q|$$

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

After optimization:

$$|p \cdot q|$$

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

Defining cmath in IRDL

```
Dialect cmath {



}
```

A ***Dialect*** is a single self-contained text.

Defining cmath in IRDL

```
Dialect cmath {
  Type complex {
    Parameters (elementType: !FloatType)
  }




}
```

A ***Dialect*** is a single self-contained text.

A ***Type*** can be parametric.

Defining cmath in IRDL

```
Dialect cmath {
  Type complex {
    Parameters (elementType: !FloatType)
  }

  Alias AnyComplex = !complex<!FloatType>






}
```

A ***Dialect*** is a single self-contained text.

A ***Type*** can be parametric.

An ***Alias*** abbreviates lengthy types.

Defining cmath in IRDL

```
Dialect cmath {
  Type complex {
    Parameters (elementType: !FloatType)
  }

  Alias AnyComplex = !complex<!FloatType>

  Operation mul {
    ConstraintVar (T: !AnyComplex)
    Operands (lhs: !T, rhs: !T)
    Results (res: !T)

    Format "$lhs, $rhs : $T.elementType"
  }
}
```

A ***Dialect*** is a single self-contained text.

A ***Type*** can be parametric.

An ***Alias*** abbreviates lengthy types.

An ***Operation*** defines the actual IR.

Making IRDL complete

How to define the `sized_vector` type?

```
Type sized_vector {
  Parameters (typ: !Any, size: ???)
  Summary "A vector with known size"
}
```

New constraints

NO SCALABILITY

Turing-Completeness

NO INTROSPECTION

IRDL-C++: Augmenting IRDL with C++

```
Constraint non_neg_int : #AnyInteger {
    CppConstraint "$_self.value() >= 0"
}
```

Local Constraints defined in C++

IRDL-C++: Augmenting IRDL with C++

```
Constraint non_neg_int : #AnyInteger {
    CppConstraint "$_self.value() >= 0"
}

Type sized_vector {
    Parameters (typ: !Any, size: #non_neg_int)
}
```

→ **Local Constraints** defined in C++

IRDL-C++: Augmenting IRDL with C++

```
Constraint non_neg_int : #AnyInteger {
    CppConstraint "$_self.value() >= 0"
}

Type sized_vector {
    Parameters (typ: !Any, size: #non_neg_int)
}

Operation append_vector {
  ConstraintVars (T: !Any)
  Operands (lhs: Vector<T, #non_neg_int>,
            rhs: Vector<T, #non_neg_int>)
  Results  (res: Vector<T, #non_neg_int>)




}
```

Local Constraints defined in C++

IRDL-C++: Augmenting IRDL with C++

```
Constraint non_neg_int : #AnyInteger {
    CppConstraint "$_self.value() >= 0"
}

Type sized_vector {
    Parameters (typ: !Any, size: #non_neg_int)
}

Operation append_vector {
  ConstraintVars (T: !Any)
  Operands (lhs: Vector<T, #non_neg_int>,
            rhs: Vector<T, #non_neg_int>)
  Results  (res: Vector<T, #non_neg_int>)

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

Local Constraints defined in C++

Global Constraints defined in C++

IRDL-C++: Augmenting IRDL with C++

```
Constraint non_neg_integer : #AnyInteger {
    CppConstraint "$_self.value() >= 0"
}

Type sized_vector {
    Parameters (typ: !Any,
}

Operation append_vector {
  ConstraintVars (T: !Any)
  Operands (lhs: Vector<T, #AnyInteger>,
            rhs: Vector<T, #AnyInteger>)
  Results  (res: Vector<T, #AnyInteger>)

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

Local Constraints defined in C++

Still mostly introspectable :-)
But we lose dynamicity :-(

Global Constraints defined in C++

IRDL-C++: Augmenting IRDL with C++

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

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

Extensible with additional C++ structs

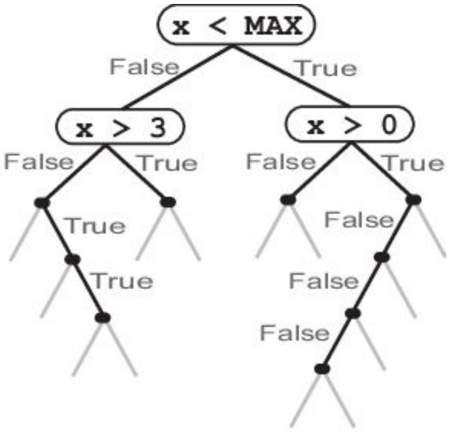
A Game of Abstractions

How do we build …?

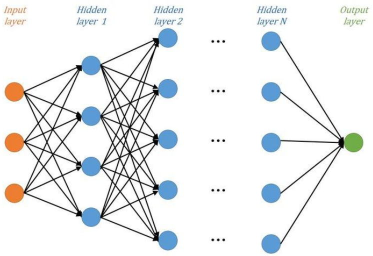
Compilers

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (= (+ (* 6 x) (* 2 y) (* 12 z)) 30))
(assert (= (+ (* 3 x) (* 6 y) (* 3 z)) 12))
(check-sat)
```

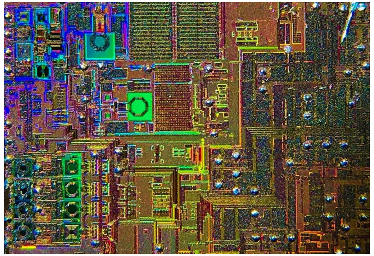
SMT Solvers



Verification



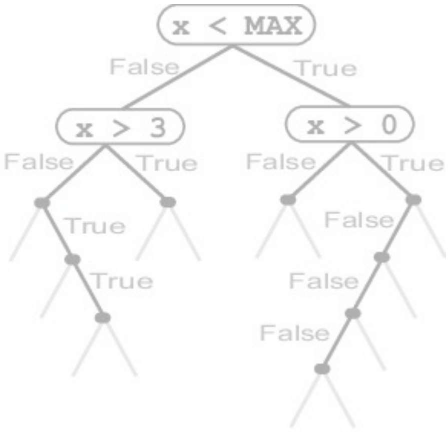
DSL Code Generators



EDA Tools

How do we build …?

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (= (+ (* 6 x) (* 2 y) (* 12 z)) 30))
(assert (= (+ (* 3 x) (* 6 y) (* 3 z)) 12))
(check-sat)
```

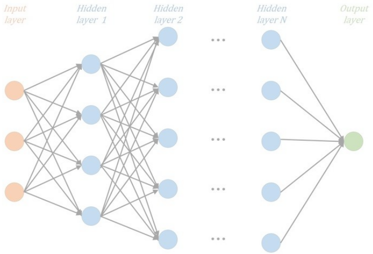


Compilers

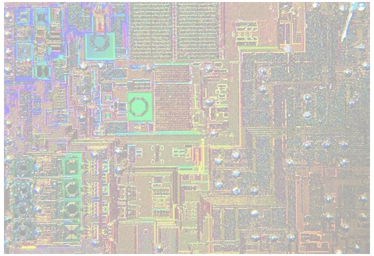
SMT Solvers

Verification

Complex C++ interface to the IR



DSL Code Generators



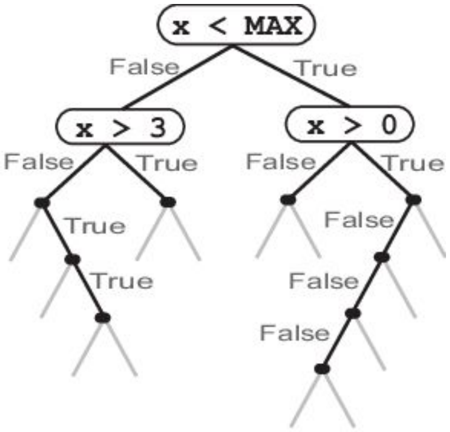
EDA Tools

How do we build …?

Compilers

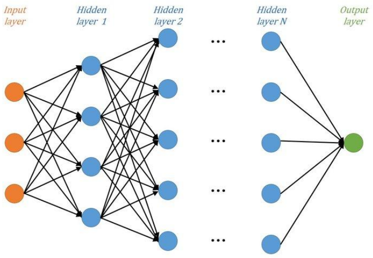
```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (= (+ (* 6 x) (* 2 y) (* 12 z)) 30))
(assert (= (+ (* 3 x) (* 6 y) (* 3 z)) 12))
(check-sat)
```

SMT Solvers

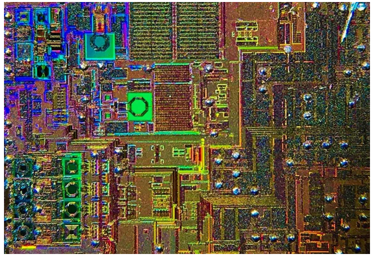


Verification

IRDL interface

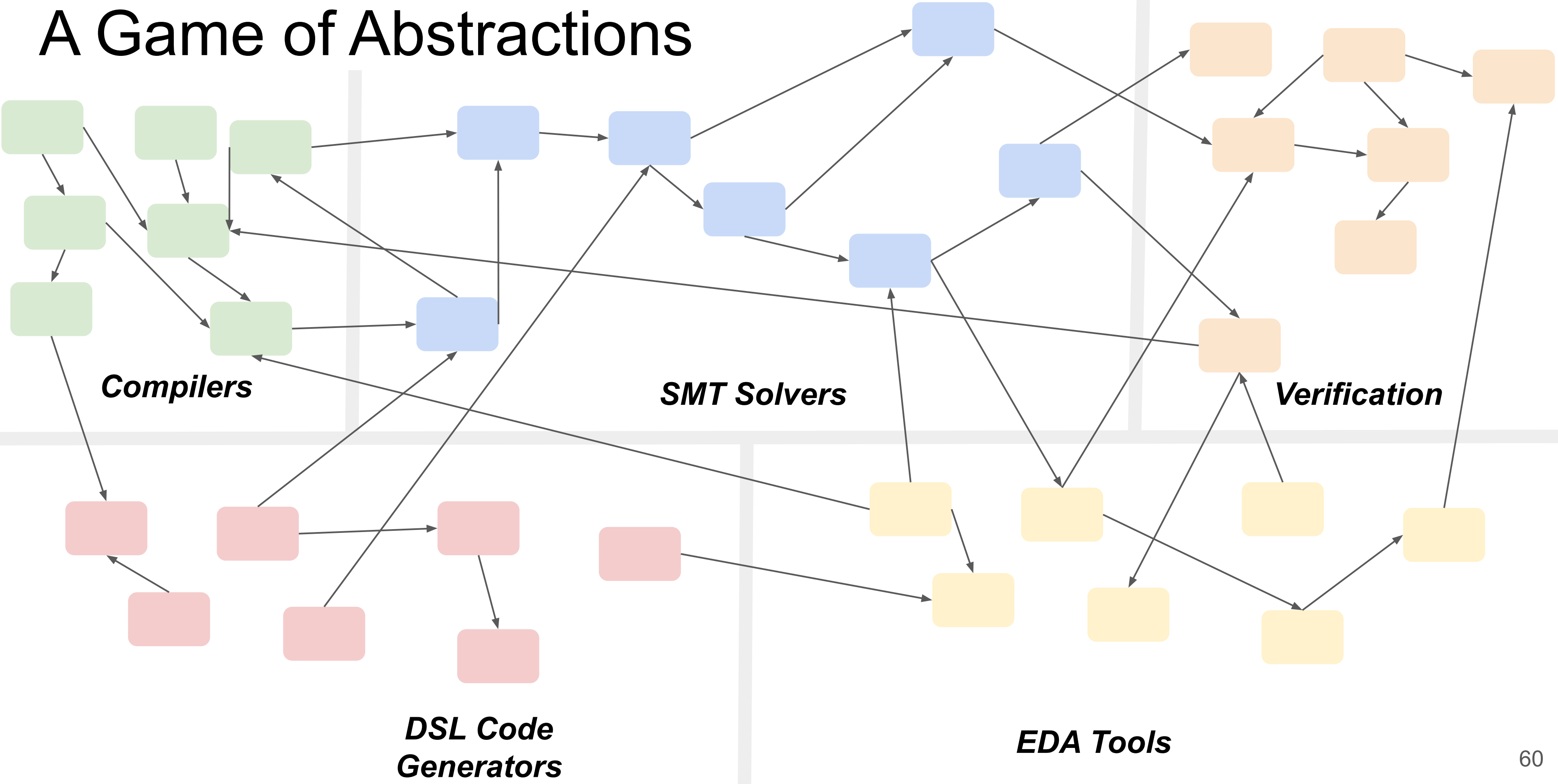


DSL Code Generators

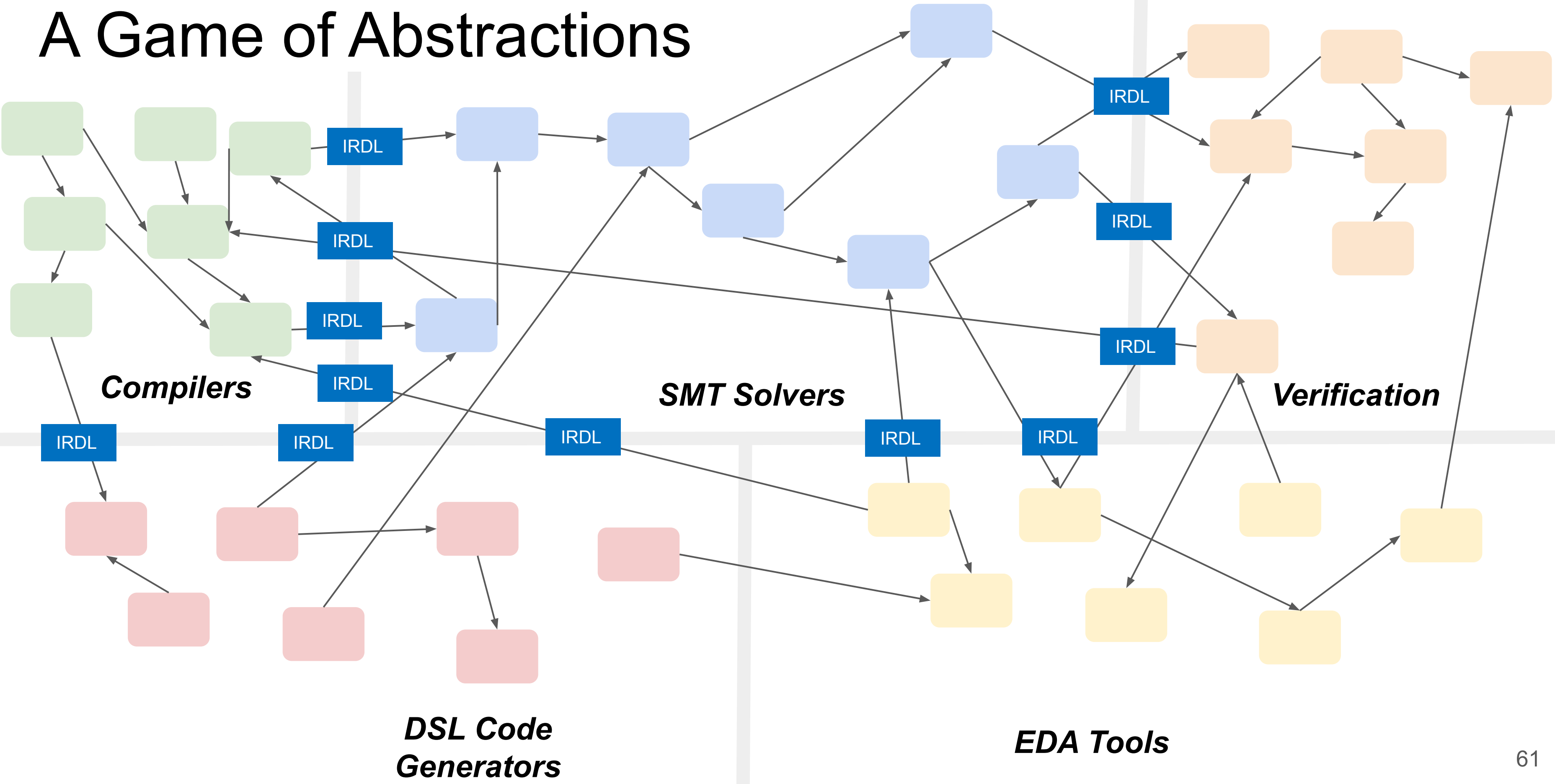


EDA Tools

A Game of Abstractions



A Game of Abstractions



Use case 1: Analysis of MLIR dialects

Translation of MLIR dialects to IRDL

MLIR

Semi-automatic translation

affine.irdl

```
irdl.dialect affine {
  irdl.operation affine.apply {
```

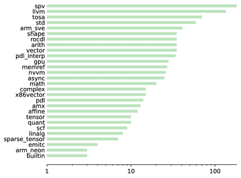
arith.irdl

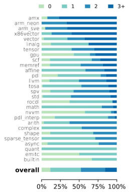
```
irdl.dialect arith {
  irdl.operation arith.addf {
```

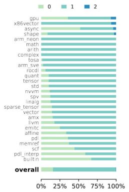
linalg.irdl

```
irdl.dialect linalg {
  irdl.type range {
                irdl.parameters()
  }
  irdl.operation linalg.index {
                irdl.results(result: CppClass<::mlir::IndexType>)
  }

  irdl.operation linalg.init_tensor {
                irdl.operands(sizes: Variadic<CppClass<::mlir::IndexType>>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.pad_tensor {
                irdl.operands(source: And<CppClass<::mlir::TensorType>, Any>,
                low: Variadic<CppClass<::mlir::IndexType>>,
                high: Variadic<CppClass<::mlir::IndexType>>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.range {
                irdl.operands(min: CppClass<::mlir::IndexType>,
                max: CppClass<::mlir::IndexType>,
                step: CppClass<::mlir::IndexType>)
                irdl.results(__empty__: CppClass<RangeType>)
  }

  irdl.operation linalg.tensor_collapse_shape {
                irdl.operands(src: And<CppClass<::mlir::TensorType>, Any>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.tensor_expand_shape {
                irdl.operands(src: And<CppClass<::mlir::TensorType>, Any>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.tiled_loop {
                irdl.operands(lowerBound: Variadic<CppClass<::mlir::IndexType>>,
                upperBound: Variadic<CppClass<::mlir::IndexType>>,
                step: Variadic<CppClass<::mlir::IndexType>>,
                inputs: Variadic<Any>,
                outputs: Variadic<And<CppClass<::mlir::ShapedType>, Any>>)
                irdl.results(results:
Variadic<And<And<CppClass<::mlir::TensorType>,
CPPPredicate<"$_self.cast<::mlir::ShapedType>().hasRank()">>, Any>>)
  }
```

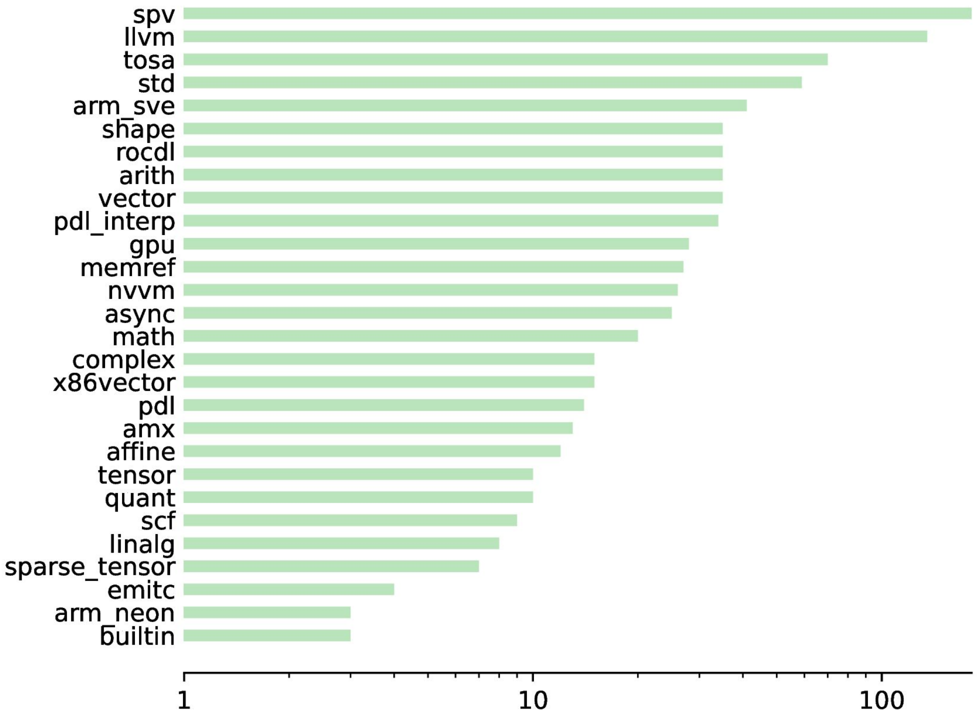
Automatic analysis



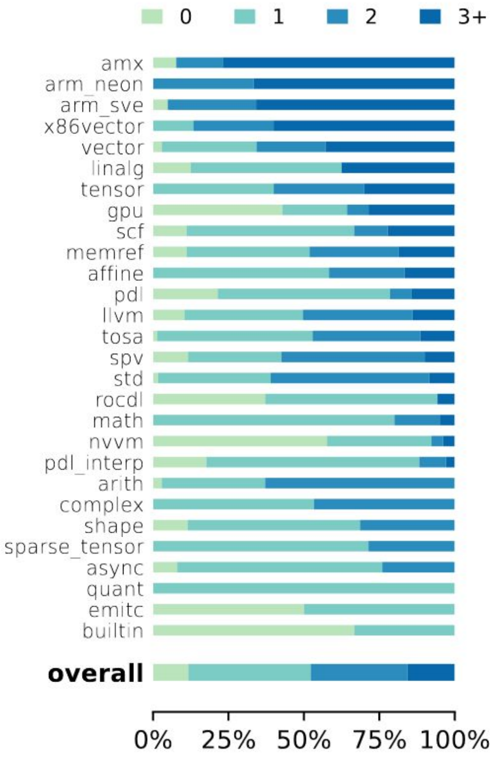




MLIR: The # of Operations in default dialects

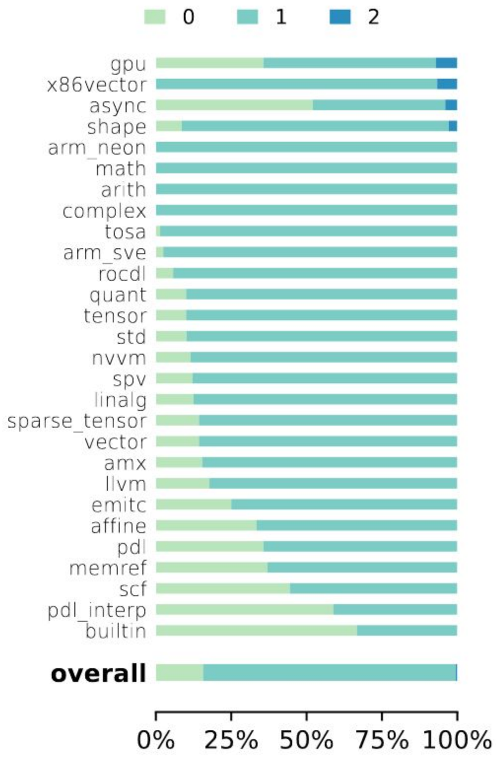


Operand use in operations across MLIR dialects



Many three-operand-ops in arm_neon and arm_sve.

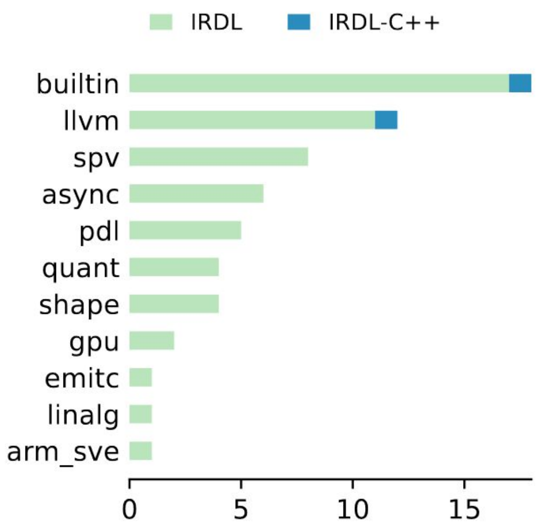
MLIR IR Operations do not require many return types



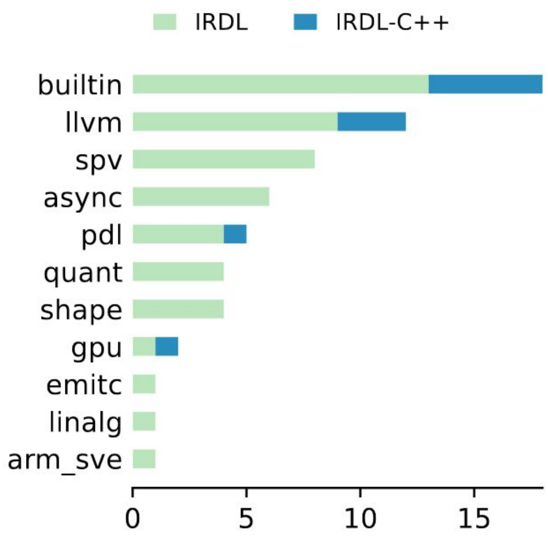
We rarely use more than two return values!

Most types do not require C++

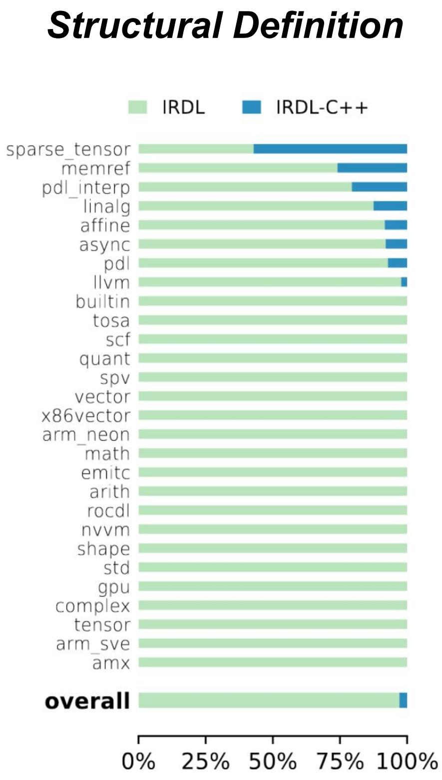
Structural Definition

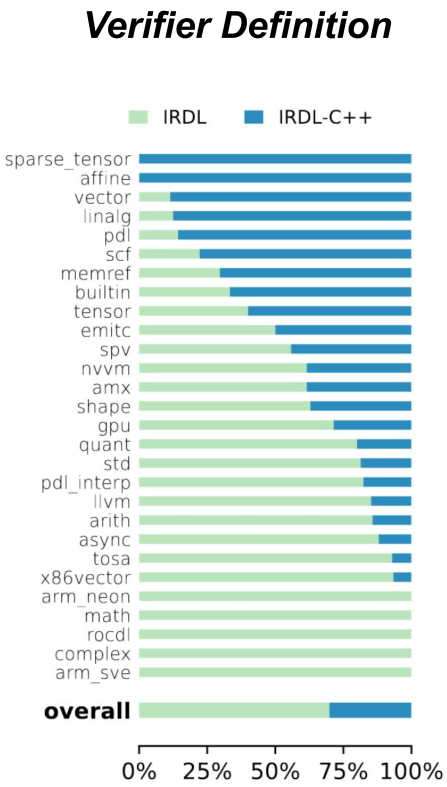


Verifier Definition



Most operations do not require C++





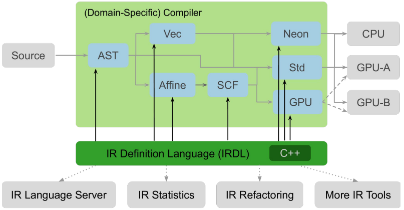
Use case 2: Bridging xDSL to MLIR

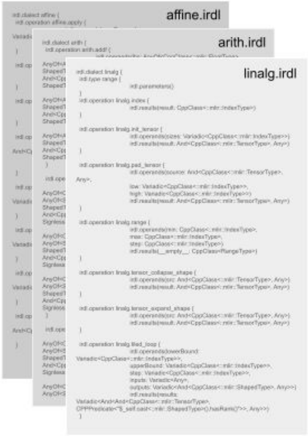
IRDL in xDSL

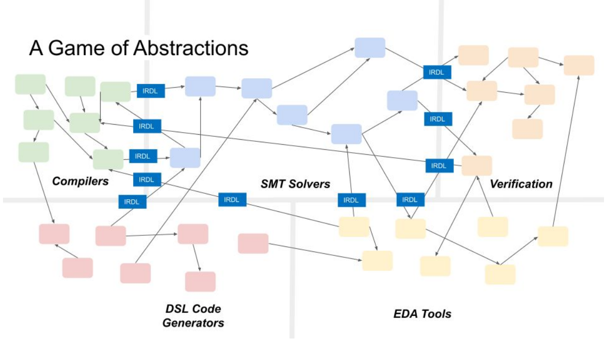
```python
@irdl_attr_definition
class VectorType(ParametrizedAttribute):
    name = "vector"

    shape: ParameterDef[ArrayAttr[IntegerAttr]]
    element_type: ParameterDef[Attribute]
```

Conclusion

Conclusion


(Domain-Specific) Compiler
Source
AST
Vec
Affine
SCF
Neon
Std
GPU
CPU
GPU-A
GPU-B
IR Definition Language (IRDL)
C++
IR Language Server
IR Statistics
IR Refactoring
More IR Tools


affine.irdl
arith.irdl
linalg.irdl


A Game of Abstractions
Compilers
SMT Solvers
Verification
DSL Code Generators
EDA Tools
IRDL