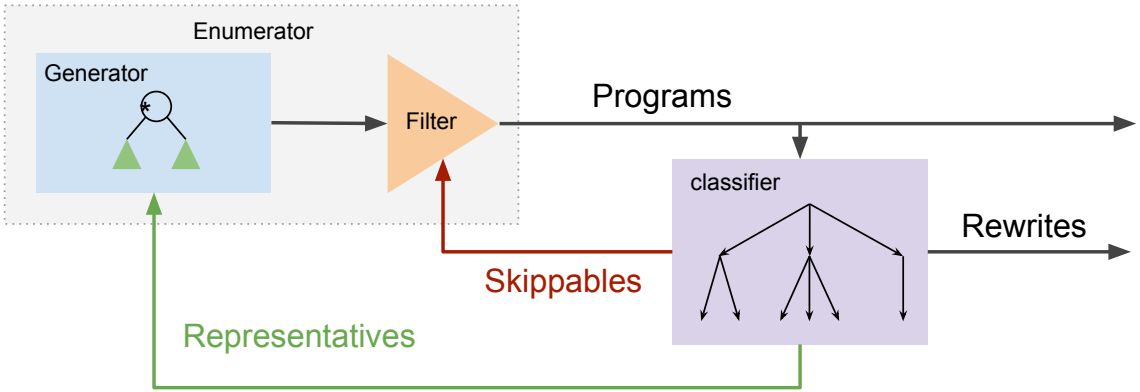
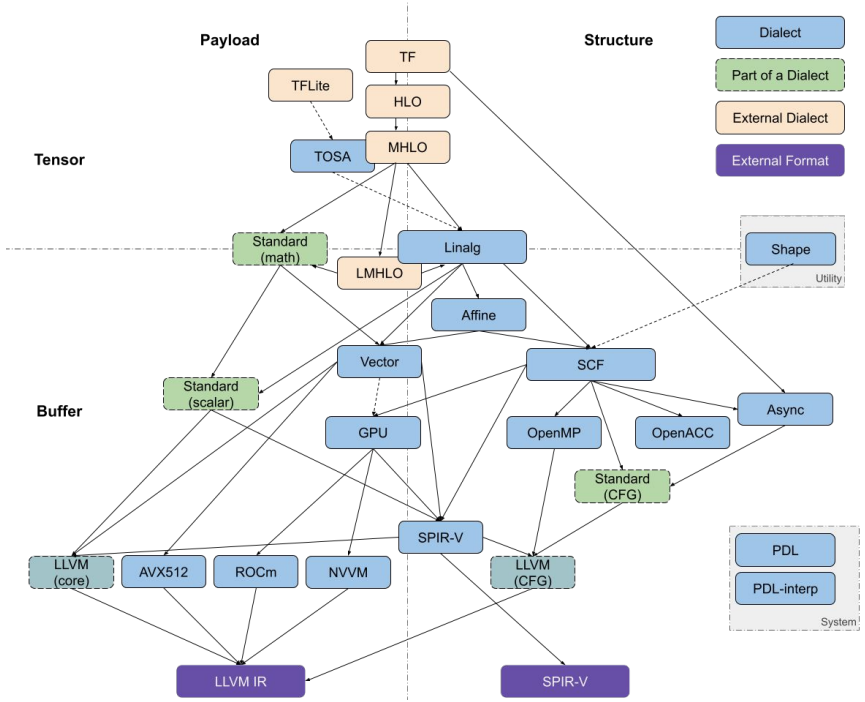
Automatically generating rewrite patterns in MLIR



Mathieu Fehr, Malo Monin

The promise of shared abstractions

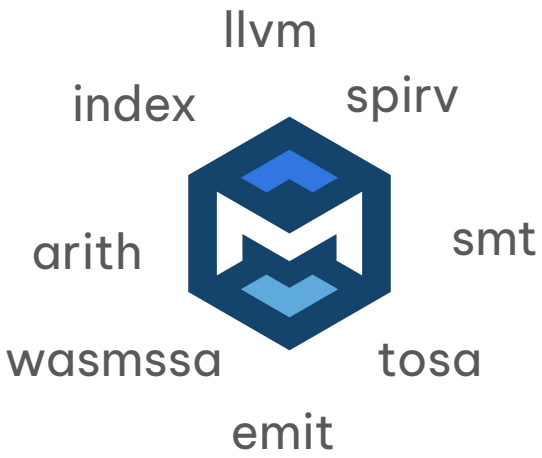


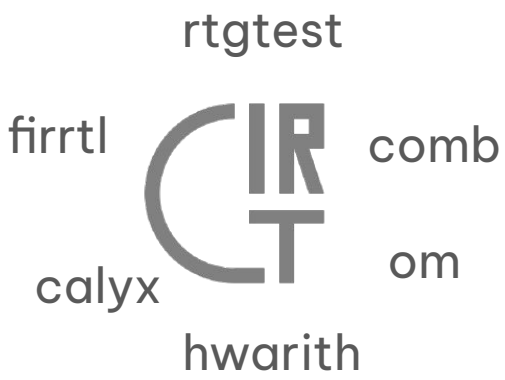
By Alex Zinenko

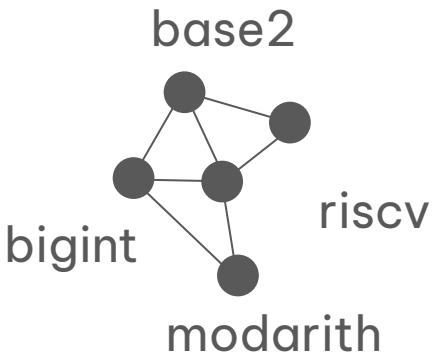
Each domain get one dialect

Define optimizations for that domain once

How many MLIR dialects redefine integer arithmetic?







How many MLIR dialects redefine integer arithmetic?

llvm

index

spirv

arith

smt

wasmssa

tosa

emit

rtgtest

firrtl

comb

calyx

om

hwarith

base2

bigint

riscv

modarith

Why do we need so many arithmetic dialects?

arith
index
comb
hwarith
modarith
bigint
smt
riscv

Why do we need so many arithmetic dialects?

arith	✅
index	✅
comb	❌
hwarith	❌
modarith	❌
bigint	❌
smt	❌
riscv	❌

Has poison semantics?

Why do we need so many arithmetic dialects?

arith	❌
index	❌
comb	✅
hwarith	✅
modarith	❌
bigint	❌
smt	❌
riscv	❌

Has 4 value logic

Why do we need so many arithmetic dialects?

arith	✅
index	✅
comb	❌
hwarith	❌
modarith	N/A
bigint	✅
smt	❌
riscv	❌

Has undefined behavior for division

Why do we need so many arithmetic dialects?

arith	✅
index	✅
comb	✅
hwarith	❌
modarith	✅
bigint	❌
smt	✅ (bv) ❌
riscv	(int) ✅

Can overflow happen?

Why do we need so many arithmetic dialects?

arith	❌
index	❌
comb	✅
hwarith	✅
modarith	❌
bigint	❌
smt	❌
riscv	❌

Is it cheap to add bits to the width?

Why do we need so many arithmetic dialects?

arith
index
comb
hwarith
modarith
bigint
smt
riscv

These dialects are very different from each others !

he width?

This result in different sound optimizations

arith	🤔
index	✅
comb	✅
hwarith	✅
modarith	✅
bigint	✅
smt	✅
riscv	✅

$$x + (y + z) = (x + y) + z$$

This result in different sound optimizations

arith	✅
index	❌
comb	✅
hwarith	✅
modarith	N/A
bigint	✅
smt	✅
riscv	✅

$x * 2 \hat{} y = x << y \quad \text{if } y < \text{bitwidth}$

This result in different sound optimizations

arith	🤔
index	❌
comb	❌
hwarith	✅
modarith	N/A
bigint	✅
smt	✅ (int) ❌
riscv	(bv) ❌

$(x * y) / x = y \quad \text{if} \quad x \mathrel{!=} 0$

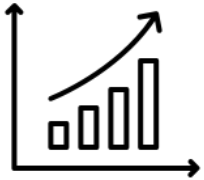
This result in different sound optimizations

arith	🤔
index	❌
comb	
hwarith	
modarith	
bigint	✅
smt	✅ (int) ❌
riscv	(bv) ❌

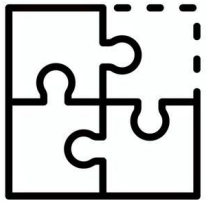
These dialects have widely different optimizations!

Defining arithmetic optimization passes is costly

Easy to get wrong



Large amount of dialects/ops



How to know if we missed optimizations?

Our vision

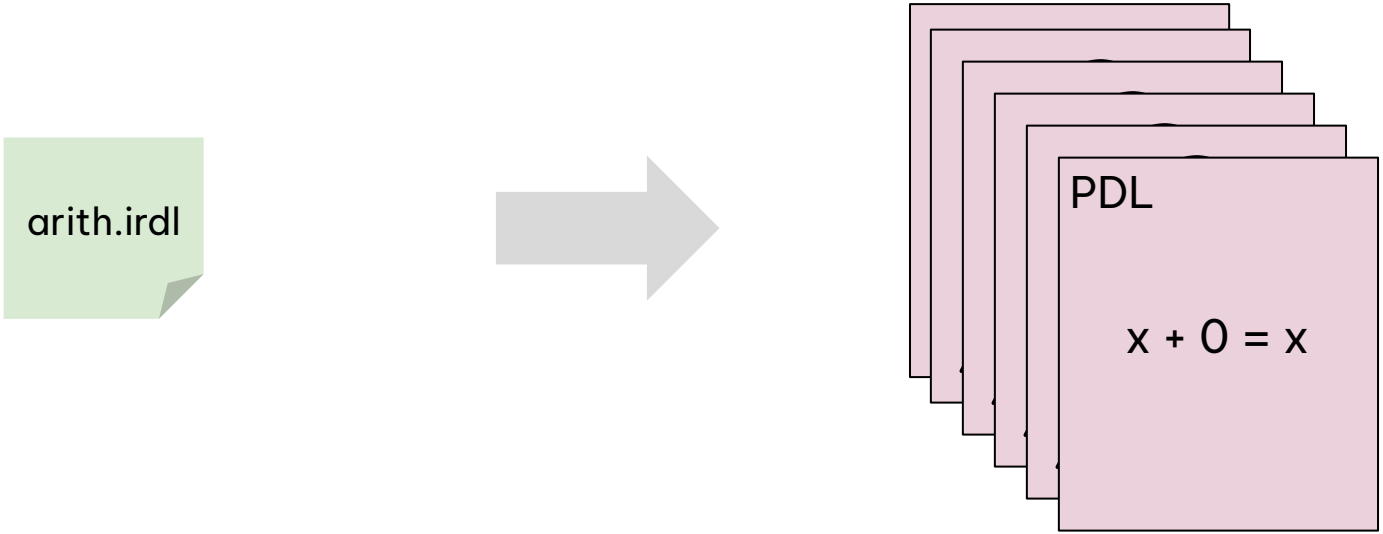
InstCombine should be synthesized for each dialect

Our vision

Synthesize a **base set** of optimizations and lowerings with guarantees of **completeness**

Use **superoptimization** for additional rewrite patterns

Tool 1 : Rewrite synthesizer



Tool 2 : Lowering synthesizer

smt.bv.ashr

arith.irdl


```
%r = smt.bv.ashr %arg0, %arg1 -> !smt.bv<8>
```

```
%0 = arith.constant 7 : i8
%1 = arith.minui %arg1, %0 : i8
%2 = arith.shrsi %arg0, %1 : i8
```

Tool 3 : Superoptimizer

```
%c2 = arith.constant 2 : i32
%r = arith.muli %arg0, %c2 : i32
```

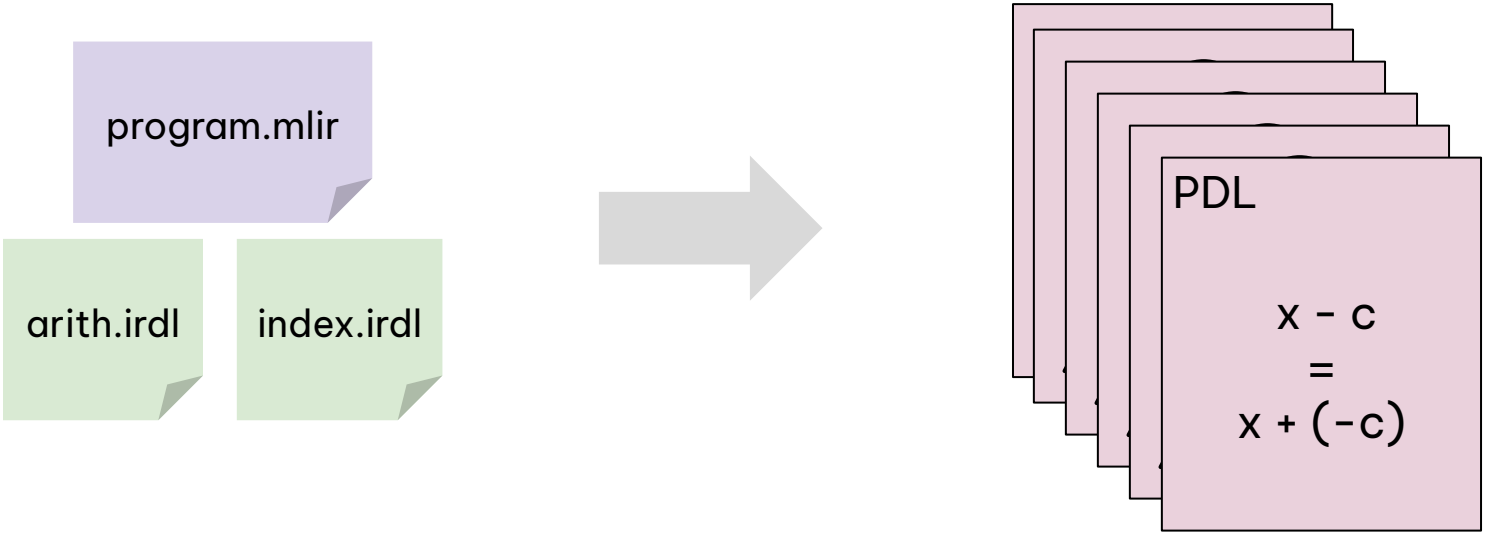
arith.irdl

```
%c2 = arith.constant 2 : i32
%r = arith.muli %arg0, %c2 : i32
```

⇩

```
%c = arith.constant 1 : i32
%r = arith.shli %arg0, %c : i32
```

Tool 3 : Superoptimizer



Enumerative synthesis

Spec: $2 \times x$ ▶

Enumerative synthesis

Spec: $2 \times x$ ▶

$$0$$
$$1$$
$$x$$
$$x + 0$$
$$x + 1$$
$$0 + x$$
$$1 + x$$
$$x + y$$
$$\vdots$$

Enumerative synthesis

Spec: $2 \times x$ ▸

0	$\times$
1	$\times$
x	$\times$
$x + 0$	$\times$
$x + 1$	Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Enumerative synthesis

Spec: $2 \times x$ ▸

0	×
1	×
x	×
$x + 0$	×
$x + 1$	Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Equivalence checker

Enumerator + Cost model

Enumerative synthesis

Spec: $2 \times x$ ▸

0	×
1	×
x	×
$x + 0$	×
$x + 1$	Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Equivalence checker

Program equivalence checking in MLIR

Program equivalence checking in MLIR

First-Class Verification Dialects for MLIR

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

MLIR is a toolkit supporting the development of extensible and composable intermediate representations (IRs) called *dialects*; it was created in response to rapid changes in hardware platforms, programming languages, and application domains such as machine learning. MLIR supports development teams creating compilers

Program equivalence checking in MLIR

```
func.func @foo(%x : i32) {
    ...
}
```

```
func.func @bar(%x : i32) {
    ...
}
```

Program equivalence checking in MLIR

```
func.func @foo(%x : i32) {
    ...
}
```

```
func.func @bar(%x : i32) {
    ...
}
```

--arith-to-smt

```
func.func @foo(%x : !smt.bv<32>) {
    ...
}

func.func @bar(%x : !smt.bv<32>) {
    ...
}
```

Program equivalence checking in MLIR

```
func.func @foo(%x : i32) {
    ...
}
```

```
func.func @bar(%x : i32) {
    ...
}
```

--arith-to-smt

```
func.func @foo(%x : !smt.bv<32>) {
    ...
}

func.func @bar(%x : !smt.bv<32>) {
    ...
}

%x = func.call @foo(%v)
%y = func.call @foo(%v)
%ne = smt.distinct %x, %y
smt.assert %ne
```

Enumerative synthesis

Spec: $2 \times x$ ▸

0	×
1	×
x	×
$x + 0$	×
$x + 1$	Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Enumerator

Enumerating MLIR programs

guided-tree-search Public

heuristically and dynamically sample (more) uniformly from large decision trees of unknown shape

C++ 13 4

Enumerating MLIR programs

guided-tree-search Public

heuristically and dynamically sample (more) uniformly from large decision trees of unknown shape

C++ 13 4

```
int a = chooser->chose(2);
int b = chooser->chose(2);
llvm::errs() << a << "," << b << "\n";
```

Enumerating MLIR programs

guided-tree-search Public

heuristically and dynamically sample (more) uniformly from large decision trees of unknown shape

C++ 13 4

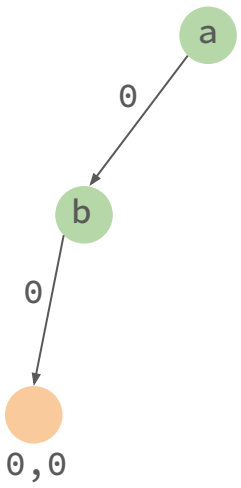
```
while (auto chooser = guide.chooser()) {
    int a = chooser->chose(2);
    int b = chooser->chose(2);
    llvm::errs() << a << "," << b << "\n";
}
```

Enumerating MLIR programs

```
while (auto chooser = guide.chooser()) {
    int a = chooser->chose(2);
    int b = chooser->chose(2);
    llvm::errs() << a << "," << b << "\n";
}
```

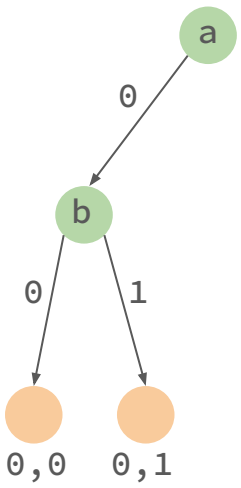
Enumerating MLIR programs

```
while (auto chooser = guide.chooser()) {
    int a = chooser->chose(2);
    int b = chooser->chose(2);
    llvm::errs() << a << "," << b << "\n";
}
```



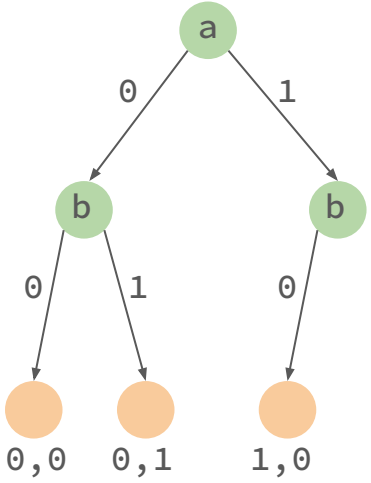
Enumerating MLIR programs

```
while (auto chooser = guide.chooser()) {
    int a = chooser->chose(2);
    int b = chooser->chose(2);
    llvm::errs() << a << "," << b << "\n";
}
```



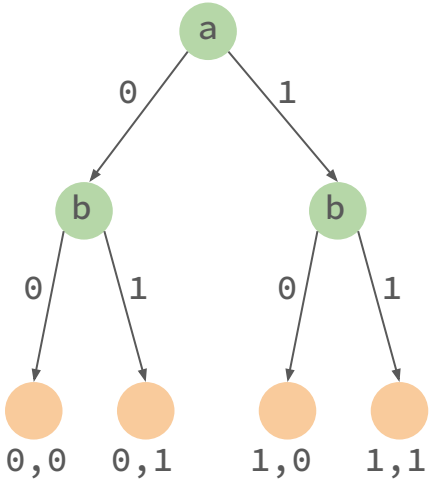
Enumerating MLIR programs

```
while (auto chooser = guide.chooser()) {
    int a = chooser->chose(2);
    int b = chooser->chose(2);
    llvm::errs() << a << "," << b << "\n";
}
```



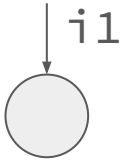
Enumerating MLIR programs

```
while (auto chooser = guide.chooser()) {
    int a = chooser->chose(2);
    int b = chooser->chose(2);
    llvm::errs() << a << "," << b << "\n";
}
```

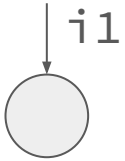


Generating a program

Generating a program

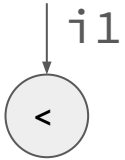


Generating a program



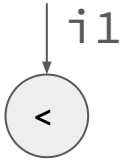
(1) Chose op with the result type

Generating a program



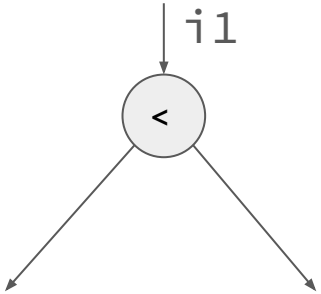
(1) Chose op with the result type

Generating a program



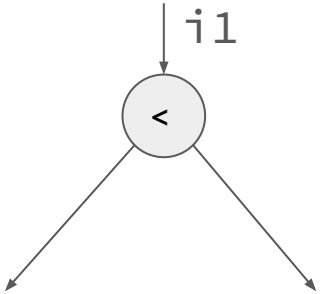
(1) Chose op with the result type
(2) Chose number of operands

Generating a program



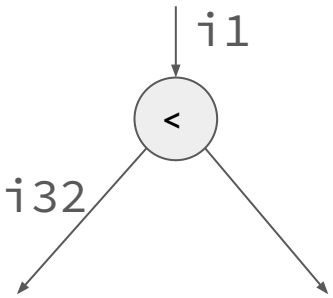
(1) Chose op with the result type
(2) Chose number of operands

Generating a program



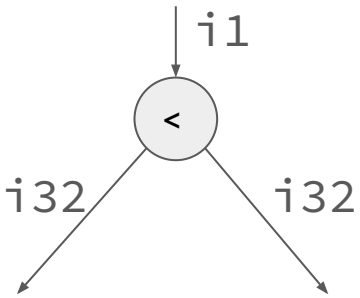
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand

Generating a program



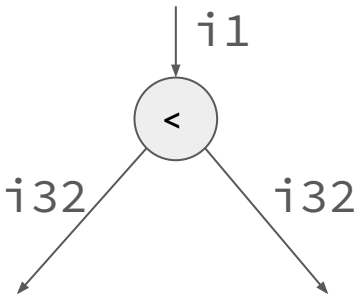
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand

Generating a program



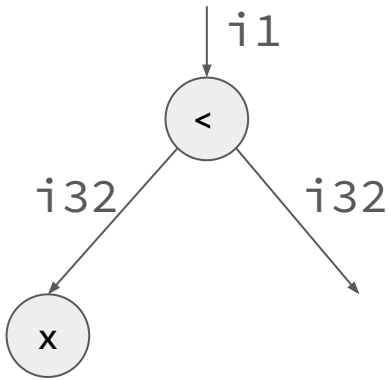
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand

Generating a program



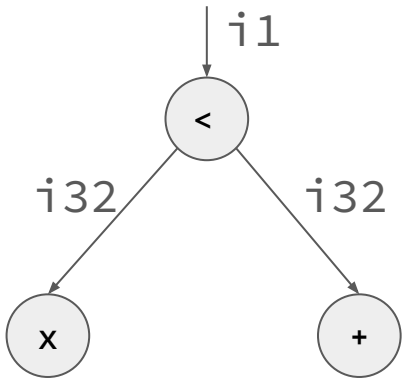
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



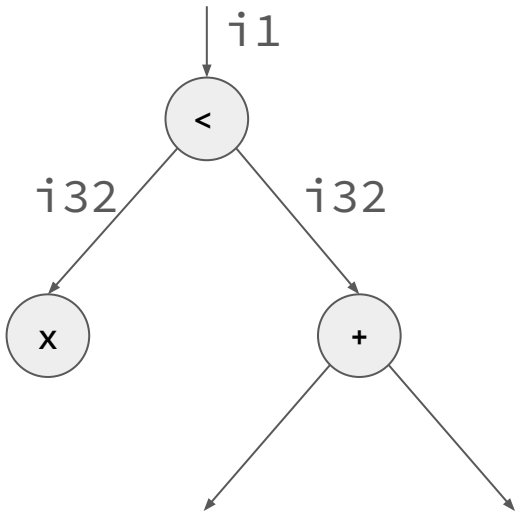
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



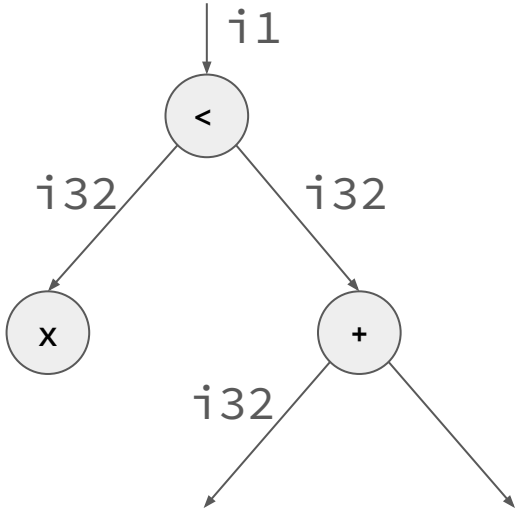
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



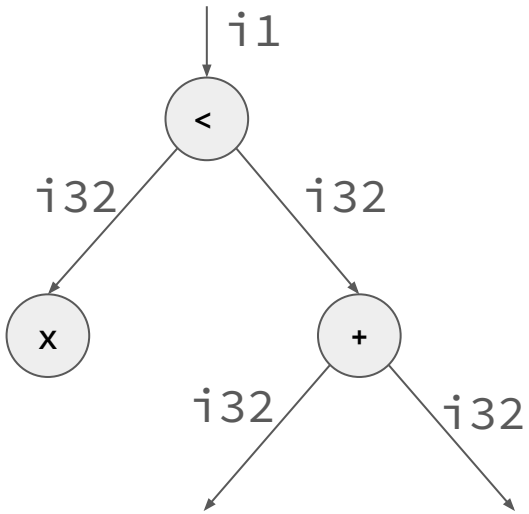
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



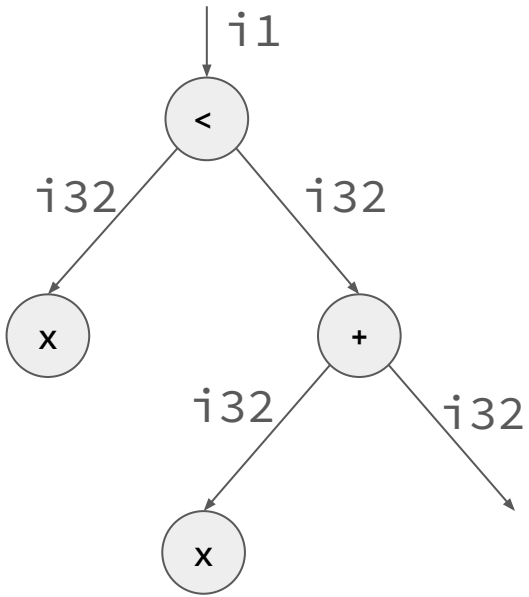
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



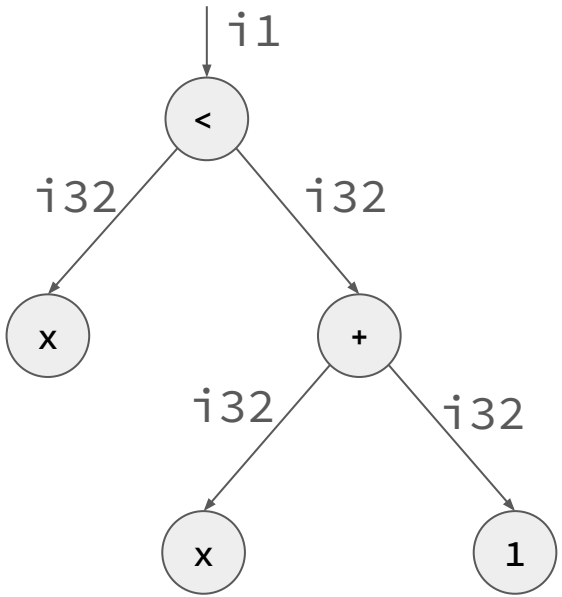
(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a program



(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a **verifying** program

(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a **verifying** program

(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Generating a **verifying** program

IRDL

```
irdl.dialect @arith {
  irdl.operation addi {
    %T = irdl.base "!builtin.integer"
    irdl.operands (lhs: %T, rhs: %T)
    irdl.results (res: %T)
  }
  ...
}
```

(1) Chose op with the result type
(2) Chose number of operands
(3) Chose type for each operand
(4) Recurse

Enumerative synthesis

📜 Spec: $2 \times x$ ▸

0	×
1	×
x	×
$x + 0$	×
$x + 1$	⚙️ Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Enumerative synthesis

📜 Spec: $2 \times x$ ▸

0	×
1	×
x	×
$x + 0$	×
$x + 1$	⚙️ Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Optimizations:

- Take input program
- Find equivalent program

Enumerative synthesis

Spec: $2 \times x$ ▸

0	×
1	×
x	×
$x + 0$	×
$x + 1$	Check
$0 + x$	
$1 + x$	
$x + y$	
$\vdots$	

Optimizations:
- Take input program
- Find equivalent program

Lowerings:
- Take input program
- Find refined program

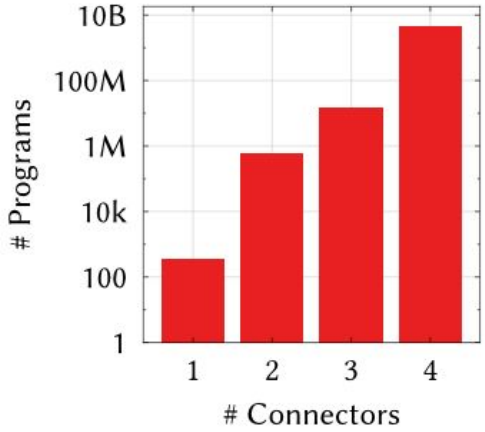
Synthesizing all rewrites?

Synthesizing all rewrites?

- Enumerate LHS programs
- Synthesize RHS program

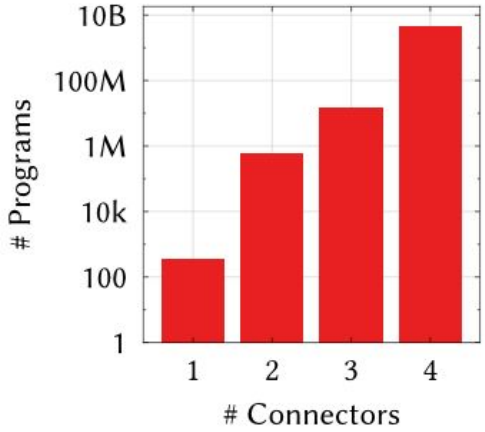
Synthesizing all rewrites?

- Enumerate LHS programs
- Synthesize RHS program



Synthesizing all rewrites?

- Enumerate LHS programs
- Synthesize RHS program



~100 000 000 000 000 candidates 😱

Intuition: A lot of candidates are redundant

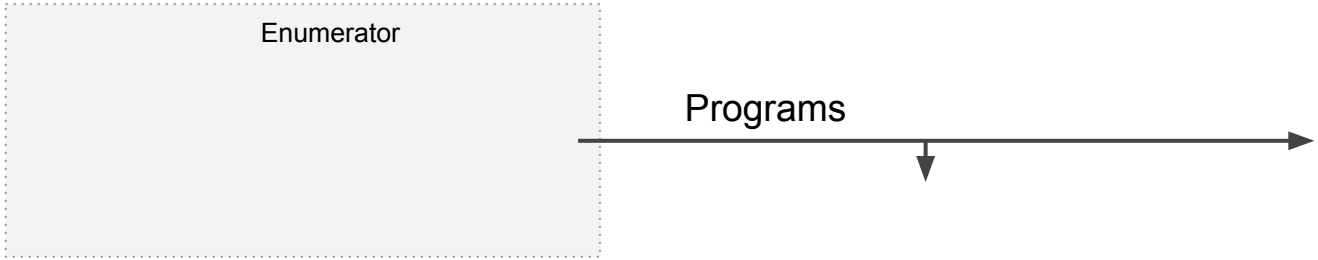
	0	$\times$
	1	$\times$
	x	$\times$
	$x + 0$	$\times$
Spec: $2 \times x$ ▸	$x + 1$	Check
	$0 + x$	
	$1 + x$	
	$x + y$	
	$\vdots$	

Intuition: A lot of candidates are redundant

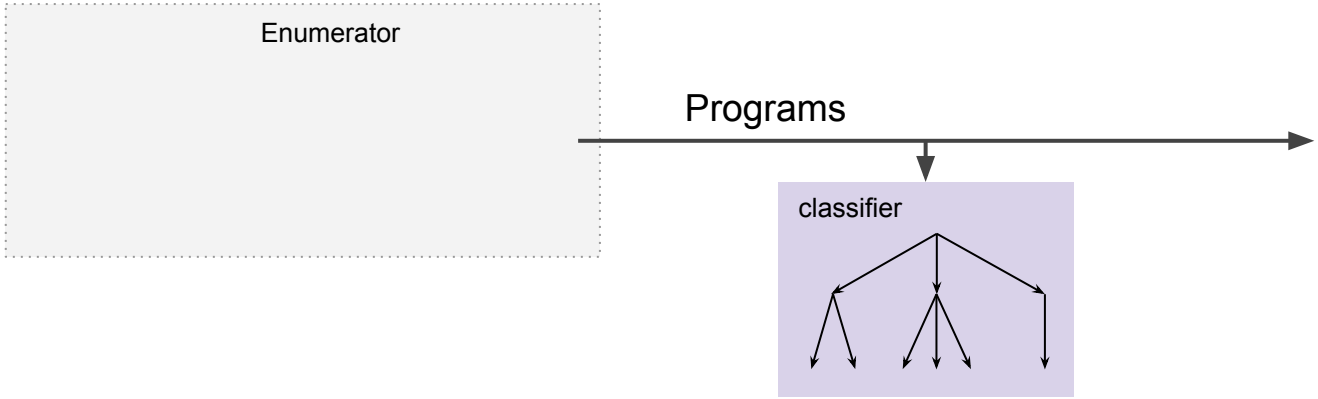
Spec: $2 \times x$ ▸

0	×
1	×
x	×
$\boxed{x + 0}$	×
$x + 1$	Check
$\boxed{0 + x}$	
$\boxed{1 + x}$	
$x + y$	
$\vdots$	

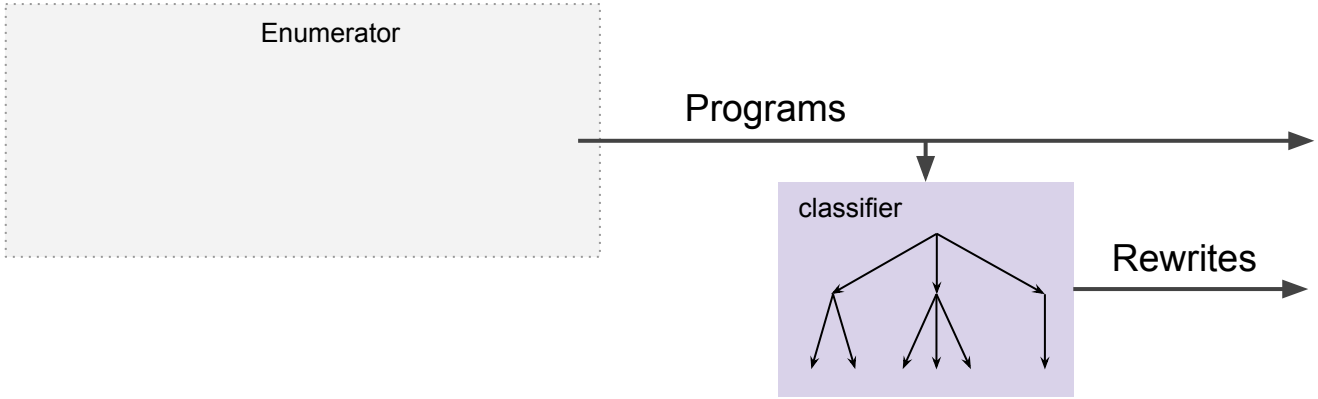
Our algorithm



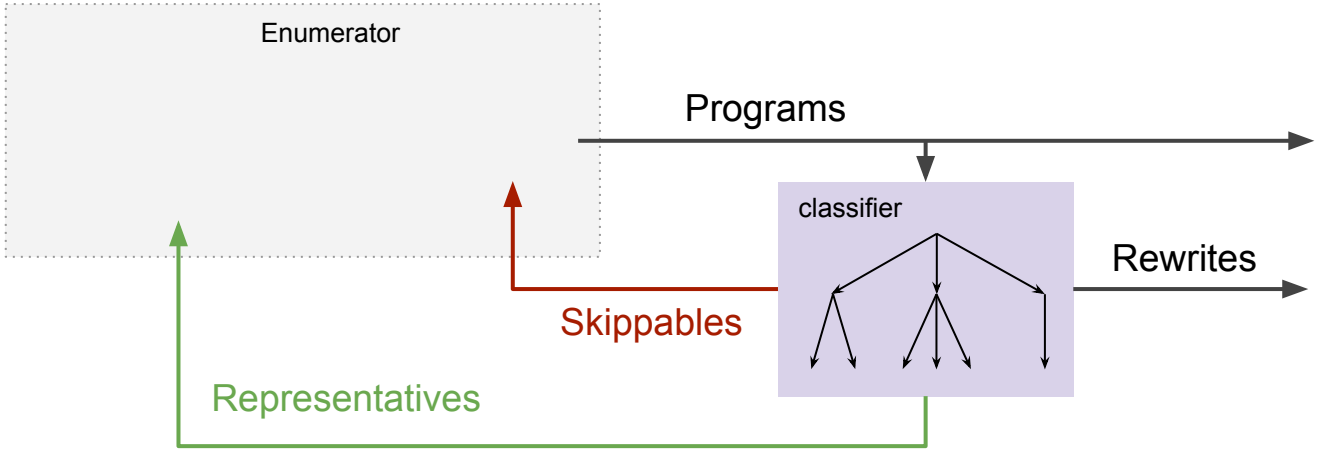
Our algorithm



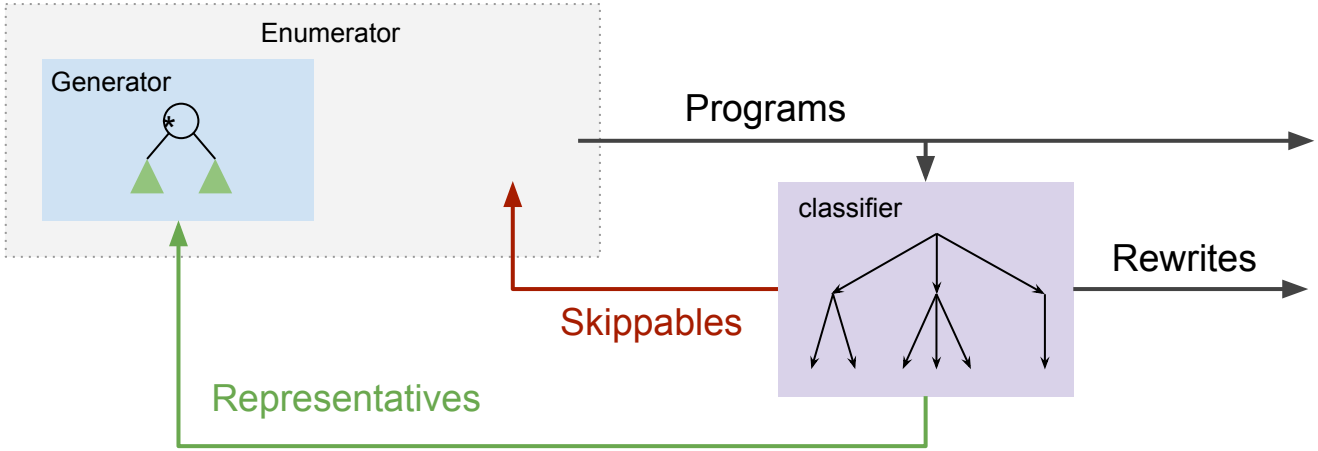
Our algorithm



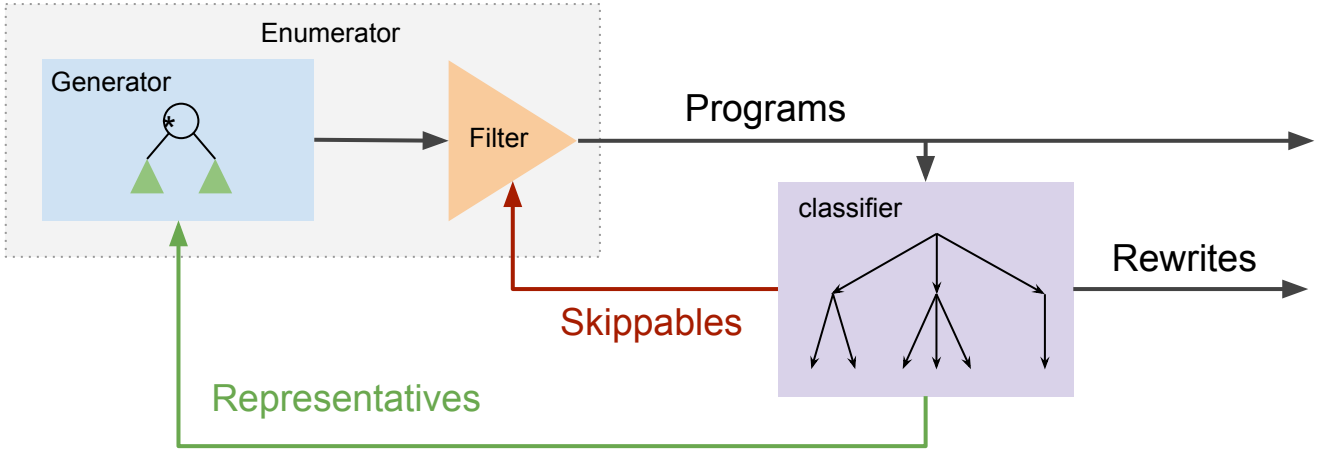
Our algorithm



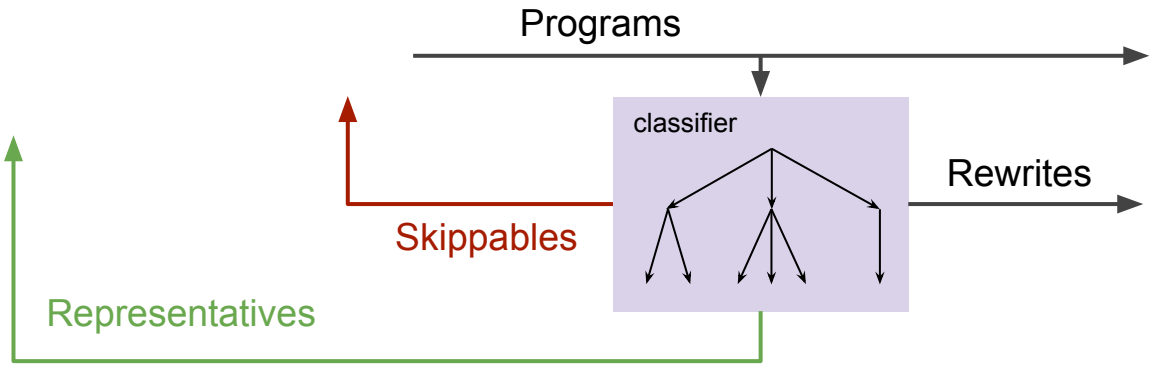
Our algorithm



Our algorithm



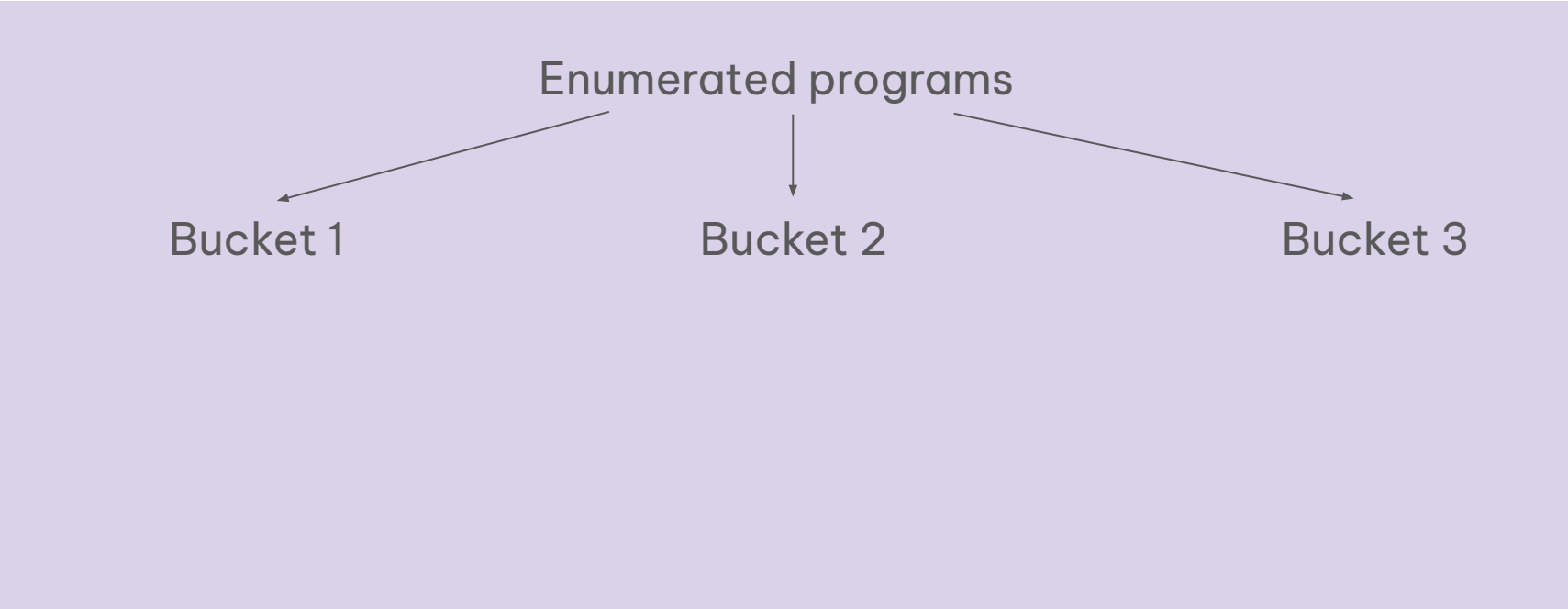
The classifier



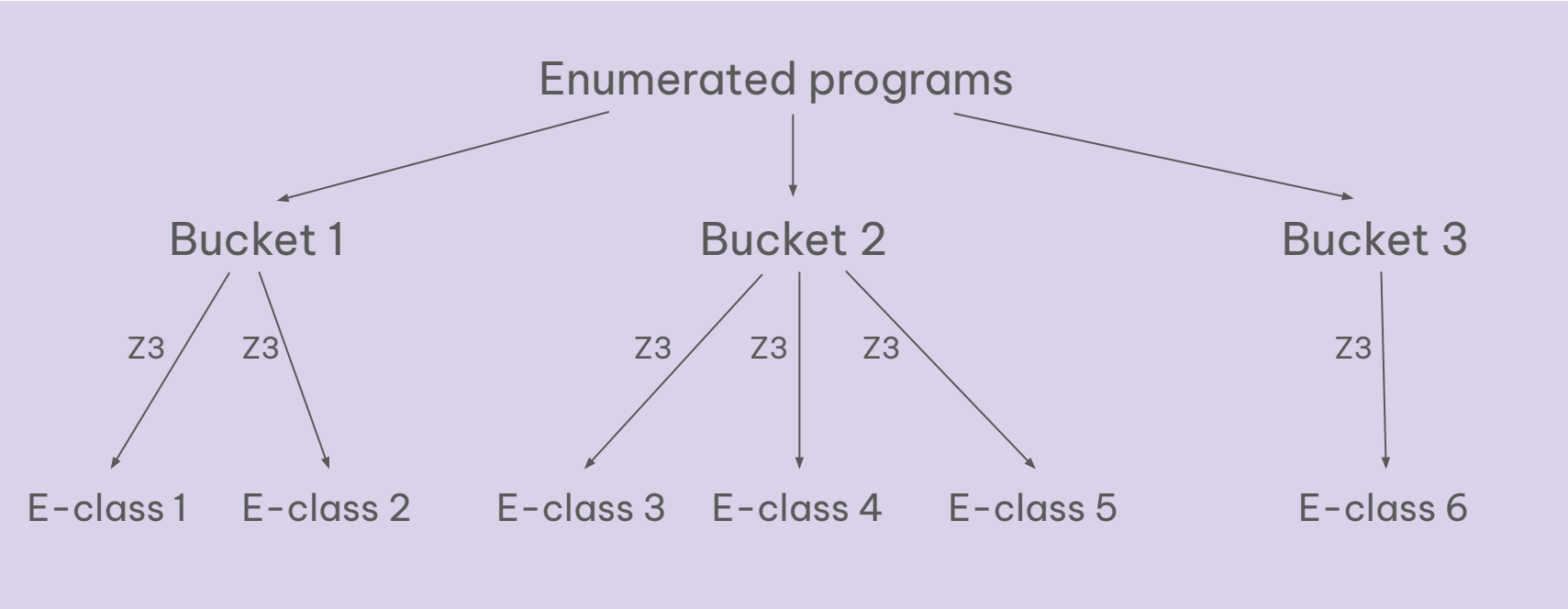
Classifying programs progressively

Enumerated programs

Classifying programs progressively



Classifying programs progressively



Choosing candidates and skippables

$$C_1 = \begin{cases} 0 \\ 0 + 0 \end{cases} \qquad C_2 = \begin{cases} x \\ x + 0 \\ 0 + x \end{cases} \qquad C_3 = \begin{cases} x + y \\ y + x \end{cases}$$

Choosing candidates and skippables

$$C_1 = \begin{cases} \boxed{0} \\ 0 + 0 \end{cases} \qquad C_2 = \begin{cases} \boxed{x} \\ x + 0 \\ 0 + x \end{cases} \qquad C_3 = \begin{cases} \boxed{x + y} \\ y + x \end{cases}$$

Choosing candidates and skippables

$$C_1 = \begin{cases} \boxed{0} \\ \boxed{0+0} \end{cases} \qquad C_2 = \begin{cases} \boxed{x} \\ \boxed{x+0} \\ \boxed{0+x} \end{cases} \qquad C_3 = \begin{cases} \boxed{x+y} \\ y+x \end{cases}$$

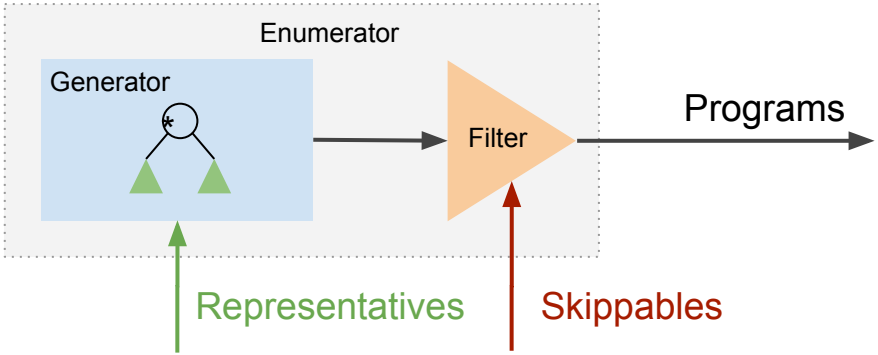
Choosing candidates and skippables

$$C_1 = \begin{cases} \boxed{0} \\ \boxed{0 + 0} \end{cases} \qquad C_2 = \begin{cases} \boxed{x} \\ \boxed{x + 0} \\ \boxed{0 + x} \end{cases} \qquad C_3 = \begin{cases} \boxed{x + y} \\ y + x \end{cases}$$

Creating the rewrites

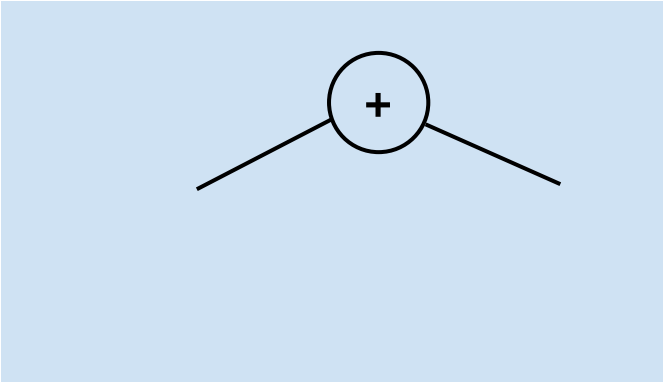
$$C_1 = \begin{cases} \boxed{0} \\ \boxed{0 + 0} \end{cases} \qquad C_2 = \begin{cases} \boxed{x} \\ \boxed{x + 0} \\ \boxed{0 + x} \end{cases} \qquad C_3 = \begin{cases} \boxed{x + y} \\ y + x \end{cases}$$

A better enumerator



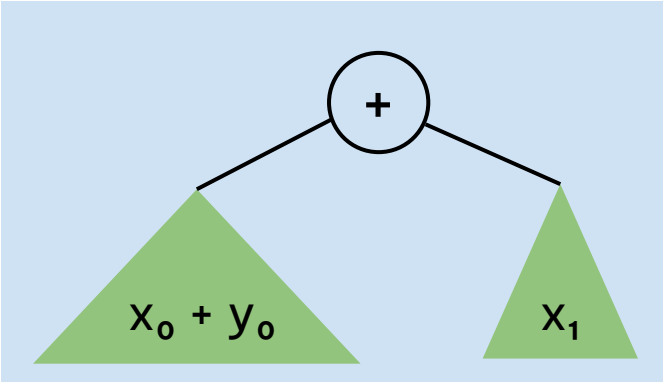
A generator using previous candidates

1) Choose an op with k operands



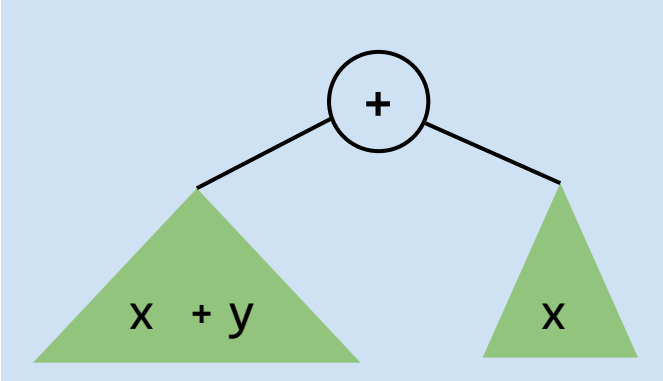
A generator using previous candidates

1) Choose an op with k operands
2) Choose k representative

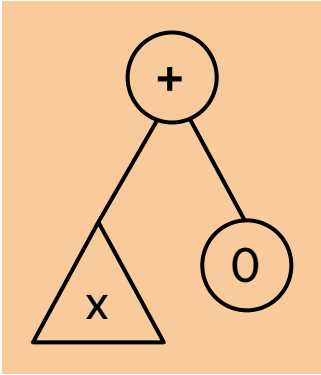


A generator using previous candidates

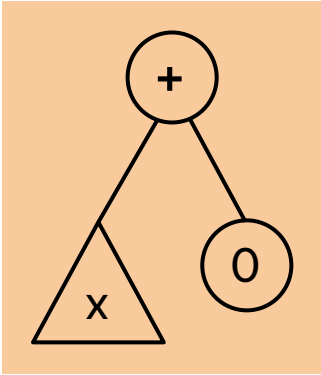
1) Choose an op with k operands
2) Choose k representative
3) Unify their parameters

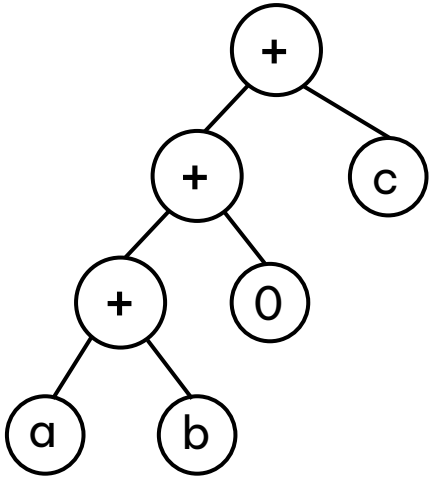


Removing programs with skippable subprograms

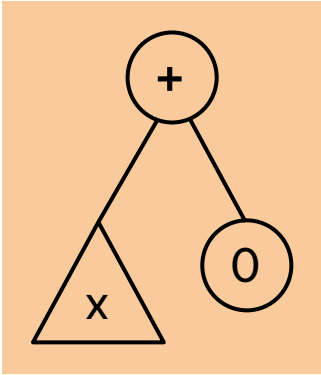


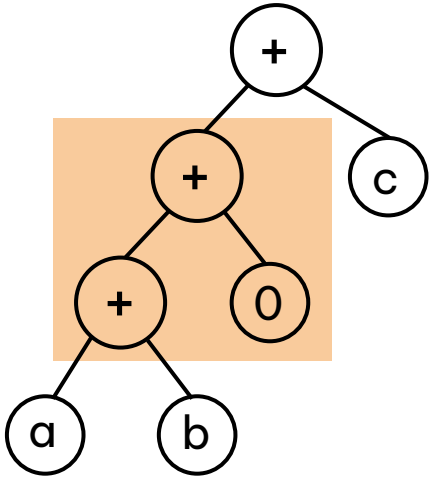
Removing programs with skippable subprograms



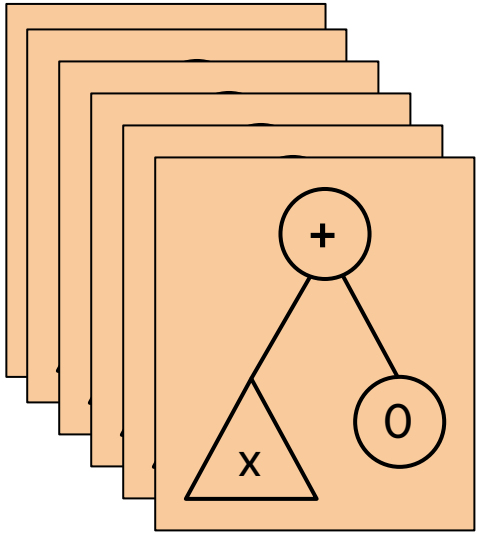


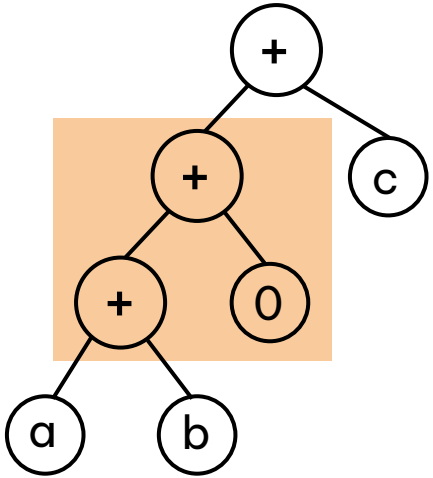
Removing programs with skippable subprograms



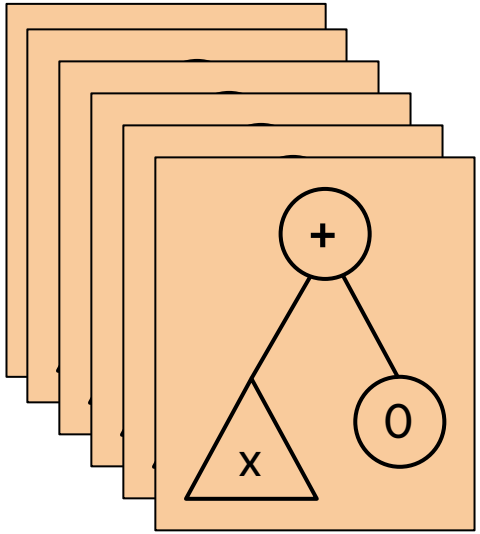


Removing programs with skippable subprograms

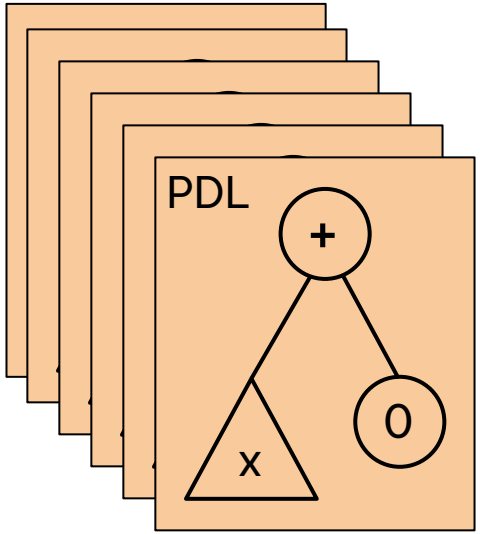




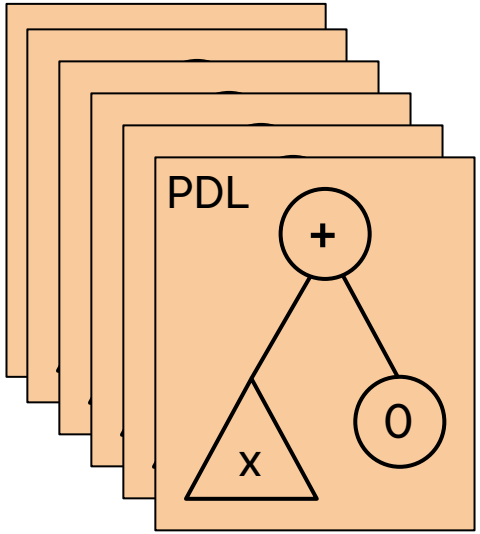
Removing programs with skippable subprograms

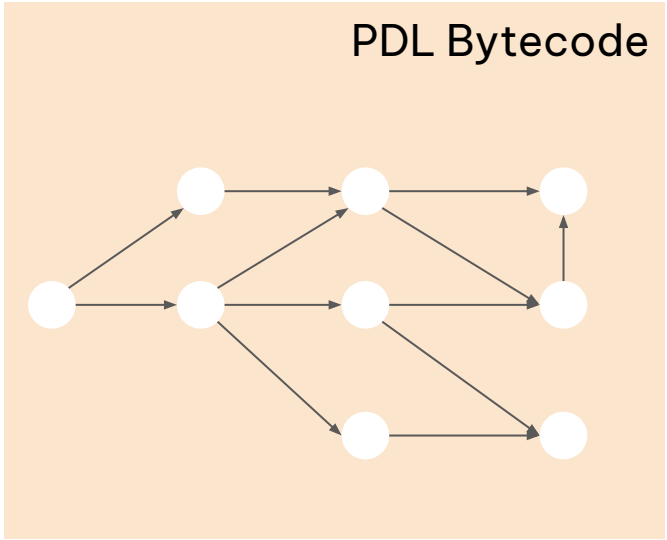


Removing programs with skippable subprograms

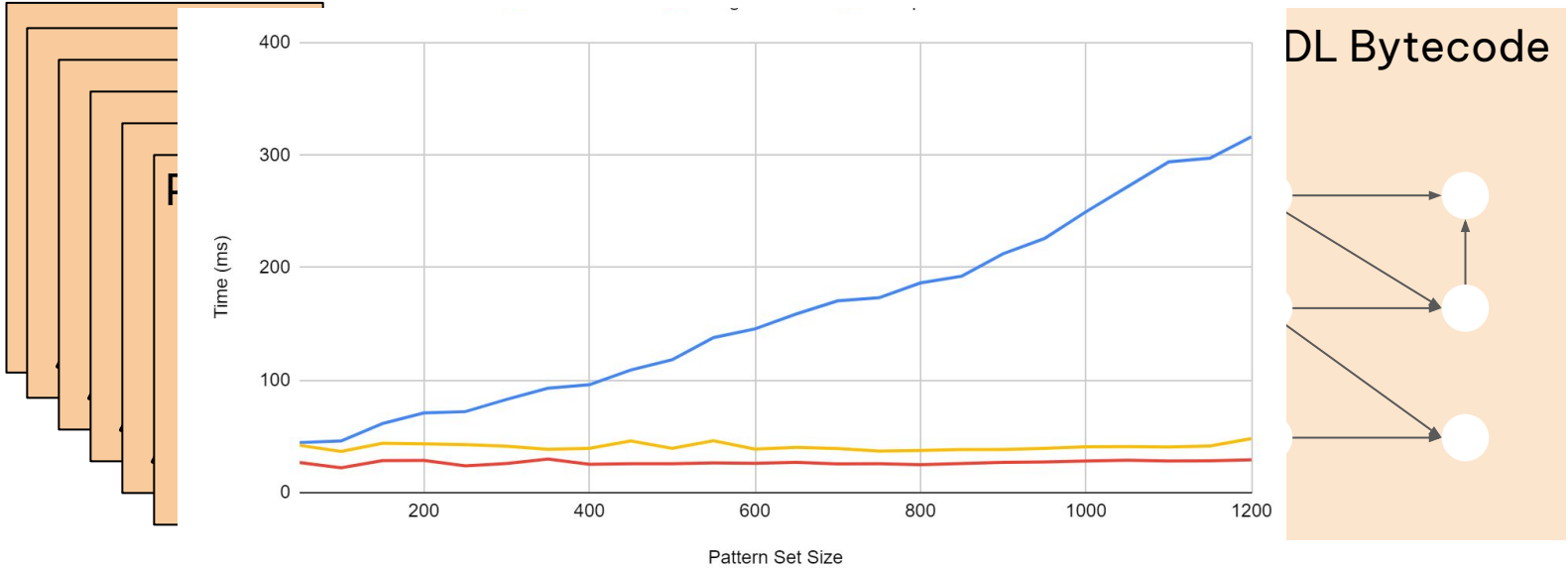


Removing programs with skippable subprograms

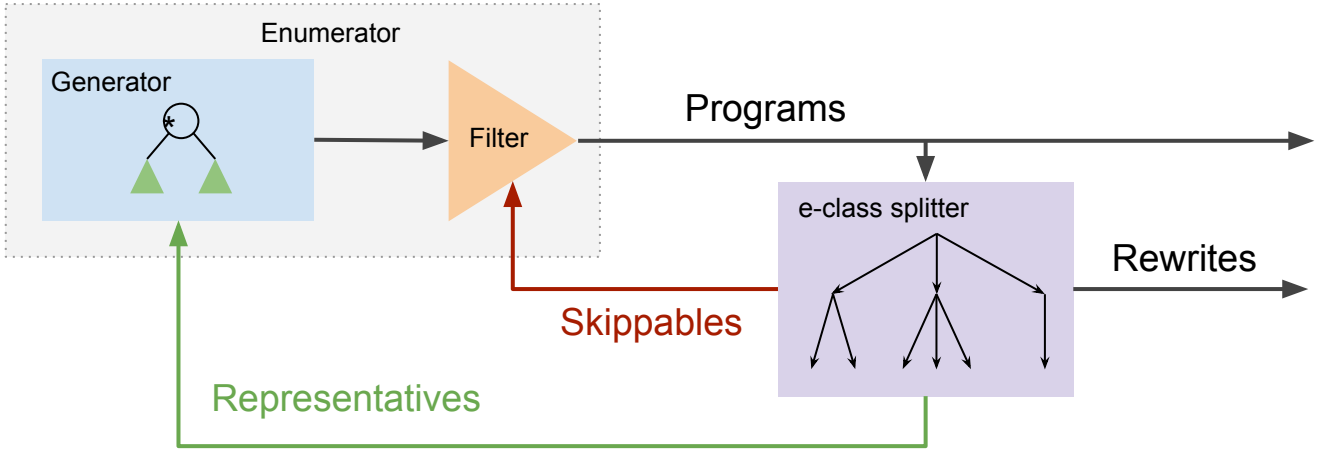




Removing programs with skippable subprograms



Our algorithm



Evaluating it on the SMT dialect

Types Booleans and bit-vectors of width 4.

Constants $\bot$, $\top$, 0, and 1.

Operations All 30 operations from MLIR.

- Booleans algebra: $\neg$, $\wedge$, $\vee$, $\oplus$, $\rightarrow$
- Bitwise operations: $\sim$, &, $|$, ^, $\gg_a$, $\gg_l$, $\ll$
- Bit-vector arithmetic: $+$, $\cdot$, $\div_s$, mod_s, rem_s, $\div_u$, rem_u
- Bit-vector comparisons: $<_s$, $<_u$, $\leq_s$, $\leq_u$, $\geq_s$, $\geq_u$, $>_s$, $>_u$
- Other: **if-then-else**, $\neq$, $=$

Evaluating it on the SMT dialect

	Enumerator			Result		
Size	**S. Space**	**Gen^{ed}**	**Enum^{ed}**	**Repr^{ves}**	**Skip^{bles}**	**Time**
0	6	100 %	100 %	100 %	0	0.96 s

Evaluating it on the SMT dialect

	Enumerator			Result		
Size	**S. Space**	**Gen^{ed}**	**Enum^{ed}**	**Repr^{ves}**	**Skip^{bles}**	**Time**
0	6	100 %	100 %	100 %	0	0.96 s
1	372	100 %	100 %	20.7 %	115	2.22 s

Evaluating it on the SMT dialect

	Enumerator			Result		
Size	**S. Space**	**Gen^{ed}**	**Enum^{ed}**	**Repr^{ves}**	**Skip^{bles}**	**Time**
0	6	100 %	100 %	100 %	0	0.96 s
1	372	100 %	100 %	20.7 %	115	2.22 s
2	59.7k	16 %	8.9 %	4.5 %	1.3k	36.14 s

Evaluating it on the SMT dialect

	Enumerator			Result		
Size	**S. Space**	**Gen^{ed}**	**Enum^{ed}**	**Repr^{ves}**	**Skip^{bles}**	**Time**
0	6	100 %	100 %	100 %	0	0.96 s
1	372	100 %	100 %	20.7 %	115	2.22 s
2	59.7k	16 %	8.9 %	4.5 %	1.3k	36.14 s
3	14.7M	3.8 %	1.8 %	1.3 %	43.1k	11.7 h

Speeding up our synthesis algorithm

Abstraction	#Programs	#Representatives	#Repr. with refinements	Time (s)
smt	6810	1367	N/A	73 s
arith	6487	2115	1720 (only phase 1)	2770 s
comb	2107	867	N/A	562 s

This is a 3-5x speedup for superoptimization!

Synthesizing smt -> arith lowering

34 patterns total

Synthesizing smt -> arith lowering

34 patterns total

- 17 patterns lowers to 1 operations
  - At most 1 minute each

Synthesizing smt -> arith lowering

34 patterns total

- 17 patterns lowers to 1 operations
  - At most 1 minute each

- 2 patterns lowers to 2 operations (ashr and lhsr)
  - Between 2 and 10 minutes each

Synthesizing smt -> arith lowering

```
%r = smt.bv.udiv(%arg0, %arg1) : !smt.bv<8>
```

```
%c0 = arith.constant 0 : i8
%zdiv = arith.cmpi eq, %arg1, %c0 : i8
%c-1 = arith.constant -1 : i8
%one = arith.constant 1 : i8
%lhs = arith.select %zdiv, %c-1, %arg0 : i8
%rhs = arith.select %zdiv, %one, %arg1 : i8
%r = arith.divui %lhs, %rhs : i8
```

Conclusion

- Simple peephole rewrites/lowerings should not be manually written
- This is a first step towards synthesizing instcombine for MLIR
- Still a lot of things to build (Generalization, Dataflow analysis)