Nice to Meet You:
Synthesizing Practical MLIR Abstract Transformers

Xuanyu Peng Dominic Kennedy Yuyou Fan

Ben Greenman John Regehr Loris D'Antoni

THE UNIVERSITY OF UTAH

UC San Diego

Problem: Finding
Abstract Transformers $f^{\#}$

Abstract

Concrete

$$n1,\ n2 \xrightarrow{+} n3$$

Problem: Finding Abstract Transformers $f^{\#}$

Abstract: Interval Analysis

$[l1, u1], [l2, u2] \xrightarrow{+^{\#}} [l3, h3]$

Concrete

$n1, n2 \xrightarrow{+} n3$

Problem: Finding

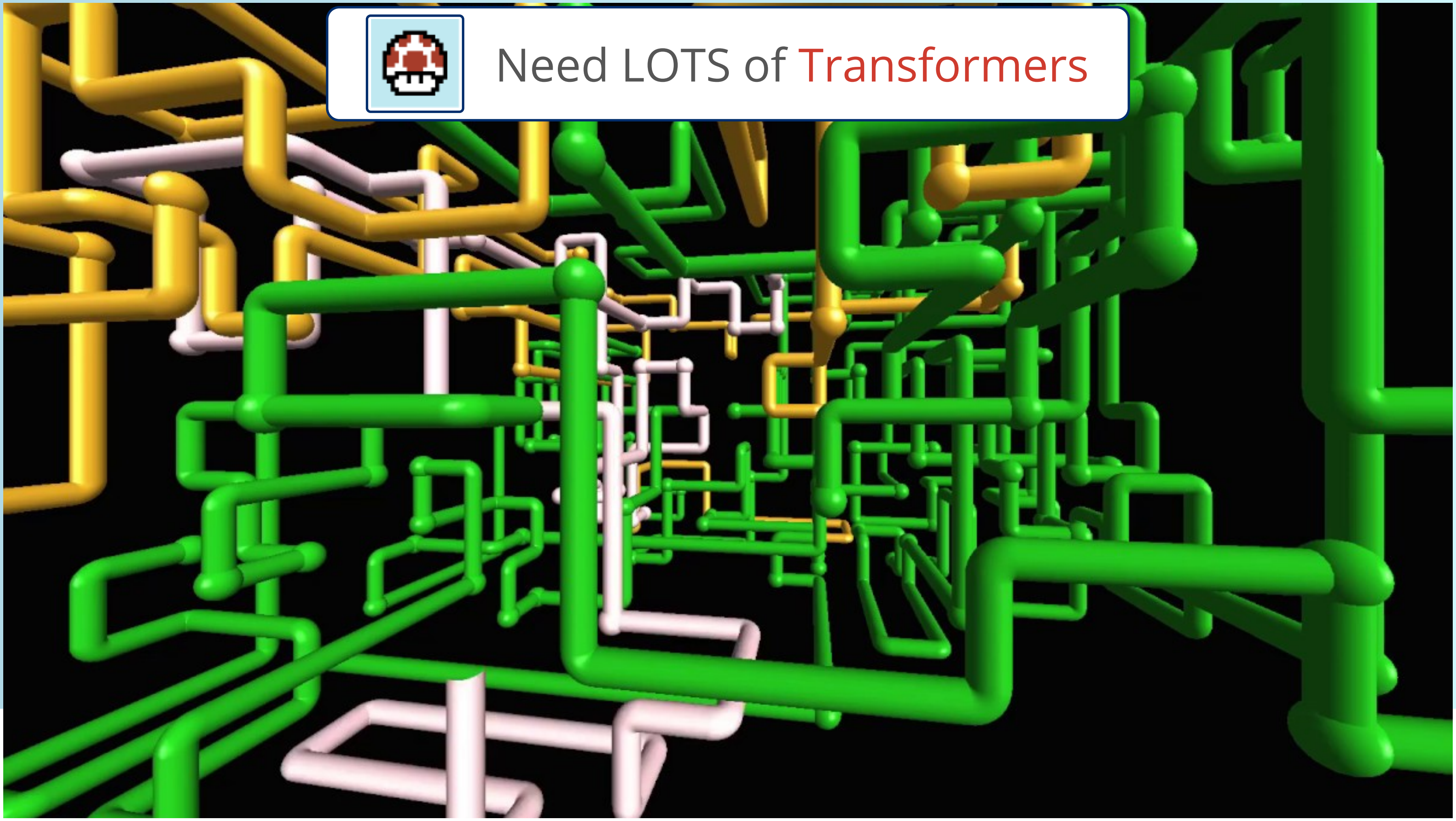
Abstract Transformers $f^{\#}$

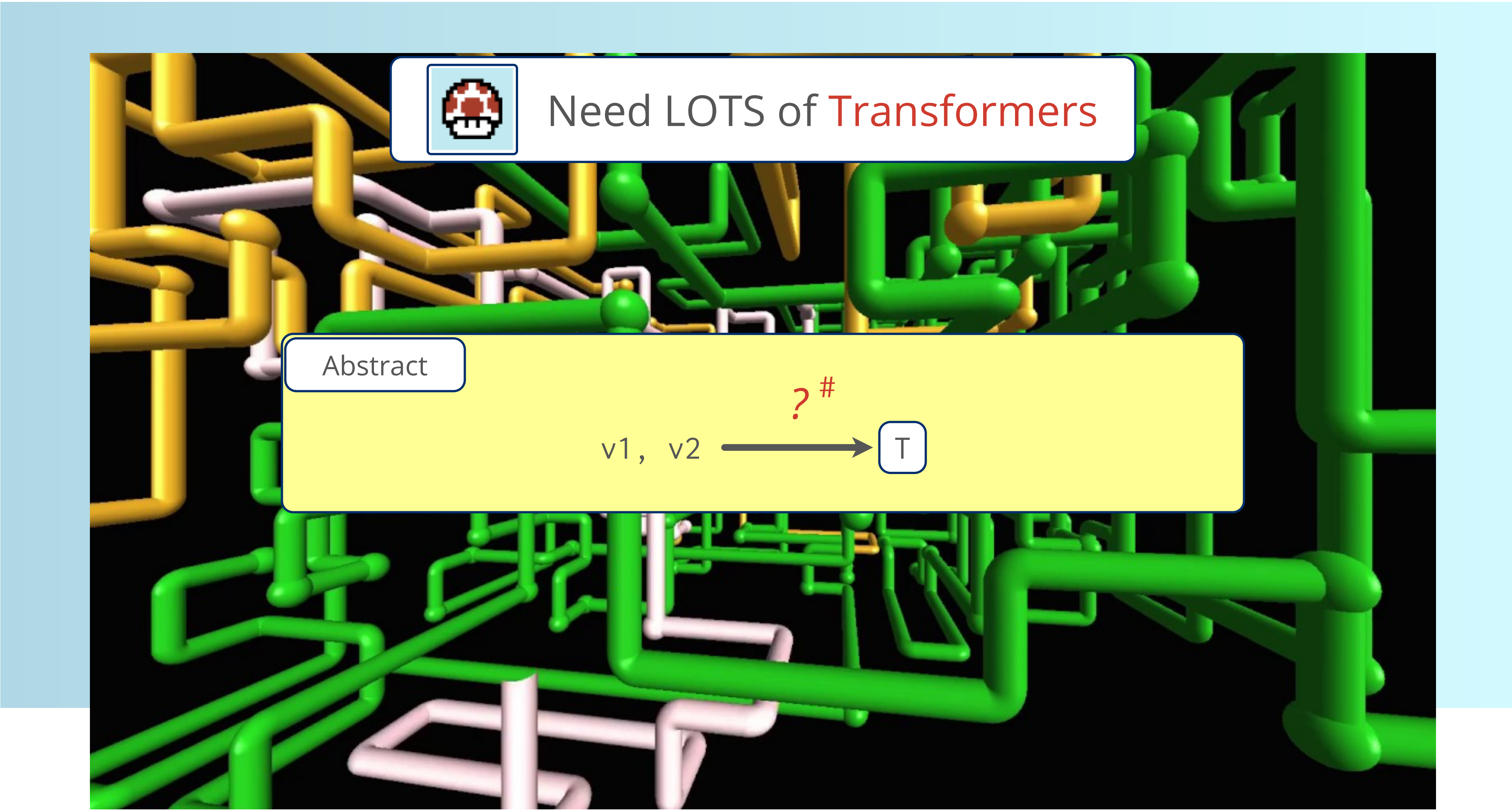
Abstract: Interval Analysis

$(+, \text{div}, \text{max}, \ldots)^{\#}$

`[l1, u1], [l2, u2]` → `[l3, h3]`

Need LOTS of Transformers




Need LOTS of Transformers
Abstract
?#
v1, v2
T

Nick Lewycky 2015-03-24 18:46:31 PDT Description

```
$ clang++ -v
clang version 3.7.0 (trunk 233044)
Target: x86_64-unknown-linux-gnu

Testcase:
#include <stdio.h>
#include <stdlib.h>
#include <string>

using namespace std;

int main(int argc, char **argv) {
  int r = 2;
  bool ok = true;
  while (ok) {
    string ab;
    for (int i = 0; i < r % 3; i++) {
      ab += "ab";
    }
    printf("%d %s\n", r, ab.c_str());
    r++;
    ok = (r < 3);
  }
}

nlewycky@ducttape:~$ clang++ -O2 a.cc -o a
nlewycky@ducttape:~$ ./a
2 ab
nlewycky@ducttape:~$ clang++ a.cc -o a
nlewycky@ducttape:~$ ./a
2 abab

The -O0 result is correct.
```

Bug 23011 - miscompile of % in loop

Status: RESOLVED FIXED

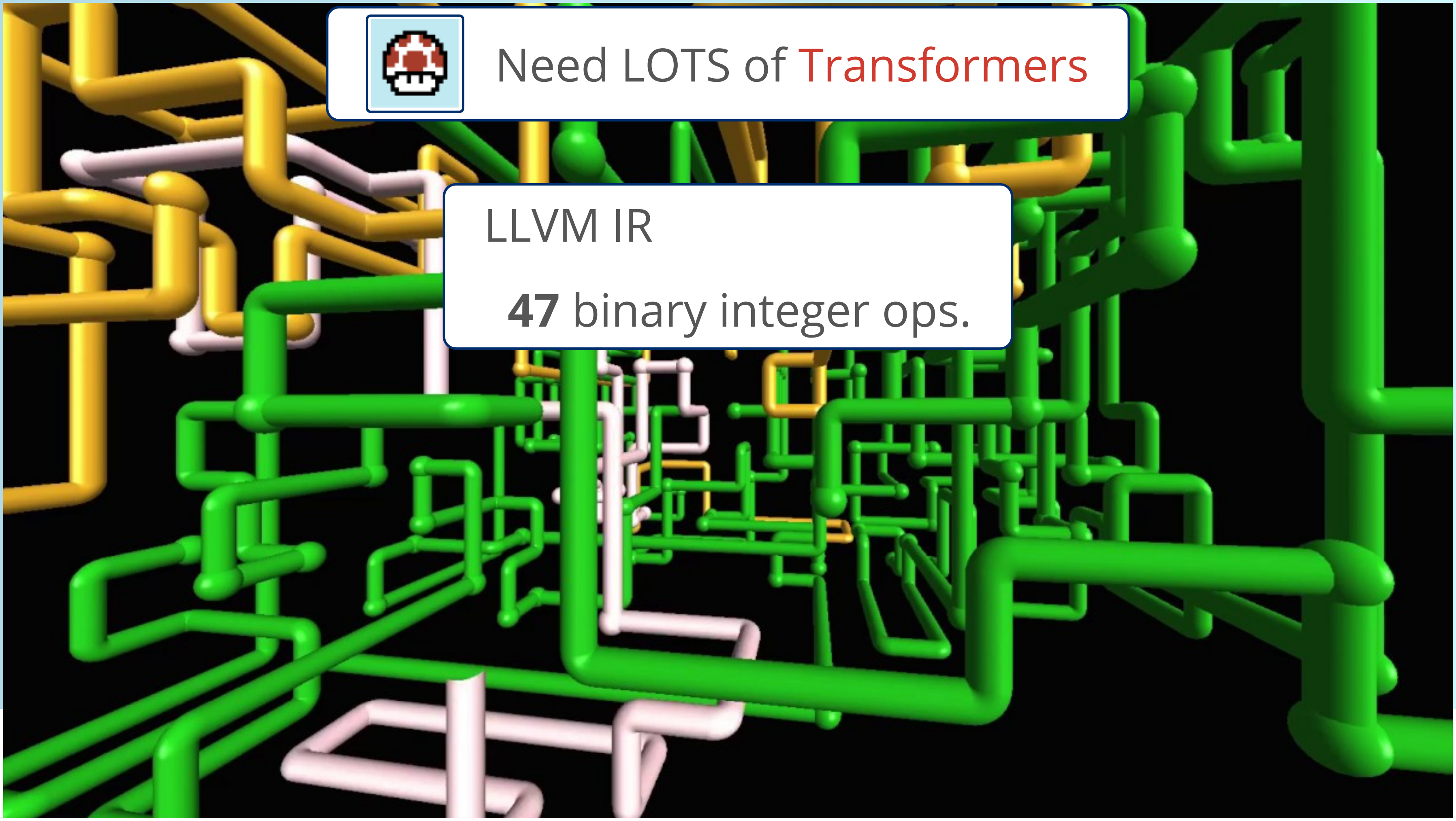
Sanjoy Das 2015-03-24 21:49:17 PDT Comment 5

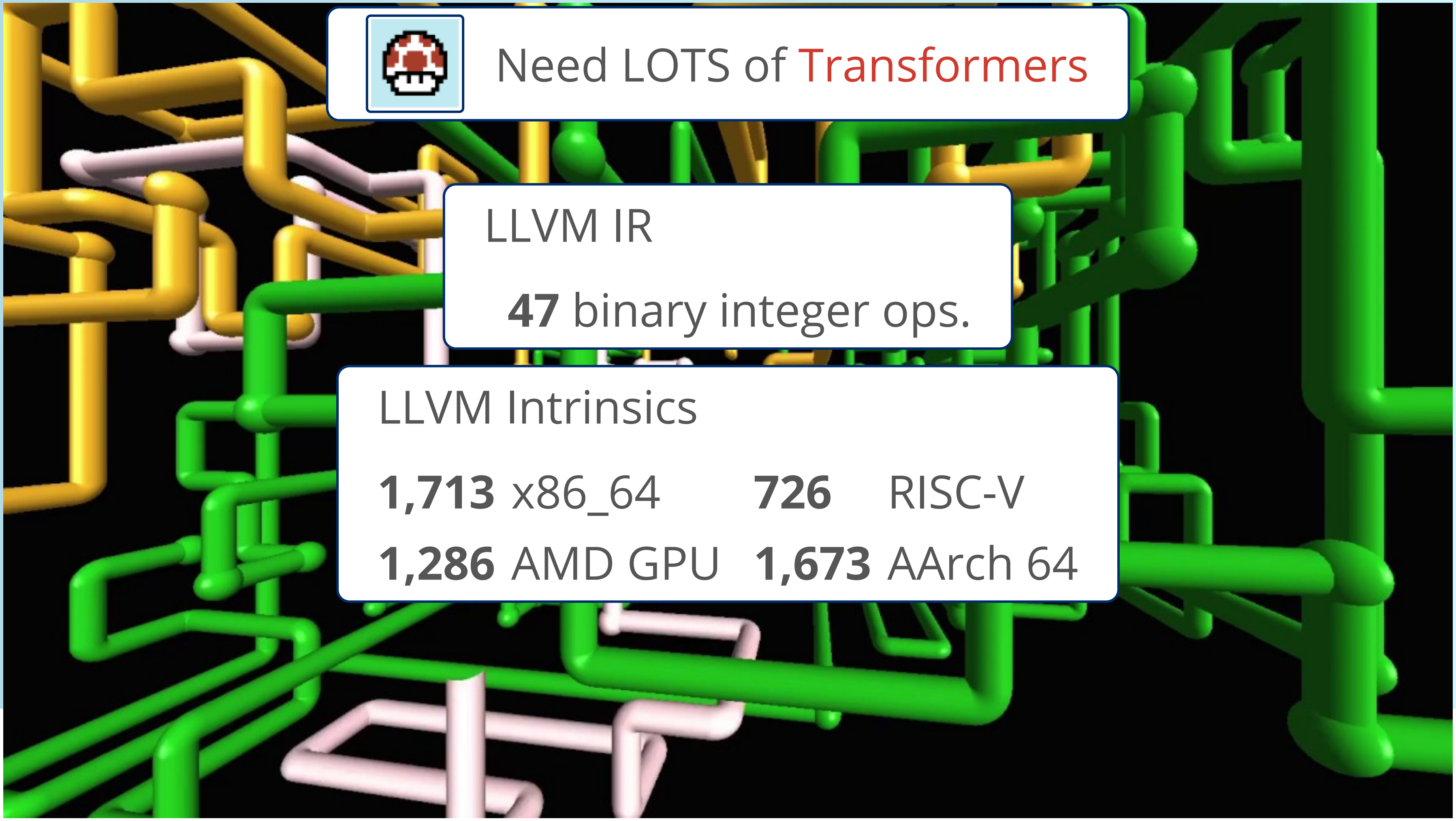
```
ComputeNumSignBits(%rem) returns 31, should be 30.
AFAICT, the bug is in ValueTracking:

      // Calculate the leading sign bit constraints by examining the
      // denominator. The remainder is in the range 0..C-1, which is
      // calculated by the log2(denominator). The sign bits are the bit-width
      // minus this value. The result of this subtraction has to be positive.
      unsigned ResBits = TyBits - Denominator->logBase2();


Denominator->logBase2() computes the floor of the log, IIUC we should be computing
the ceil.
```

Need LOTS of Transformers

LLVM IR

47 binary integer ops.




Need LOTS of Transformers
LLVM IR
47 binary integer ops.
LLVM Intrinsics
1,713 x86_64
726 RISC-V
1,286 AMD GPU
1,673 AArch 64

[l1, u1], [l2, u2] $\xrightarrow{+^{\#}}$ [l3, h3]

```
ConstantRange
ConstantRange::add(const ConstantRange &Other) const {
  if (isEmptySet() || Other.isEmptySet())
    return getEmpty();
  if (isFullSet() || Other.isFullSet())
    return getFull();

  APInt NewLower = getLower() + Other.getLower();
  APInt NewUpper = getUpper() + Other.getUpper() - 1;
  if (NewLower == NewUpper)
    return getFull();

  ConstantRange X = ConstantRange(std::move(NewLower), std::move(NewUpper));
  if (X.isSizeStrictlySmallerThan(*this) ||
      X.isSizeStrictlySmallerThan(Other))
    // We've wrapped, therefore, full set.
    return getFull();
  return X;
}
```

$[l1, u1], [l2, u2] \xrightarrow{+^{\#}} [l3, h3]$

Overflow

Top

Bot

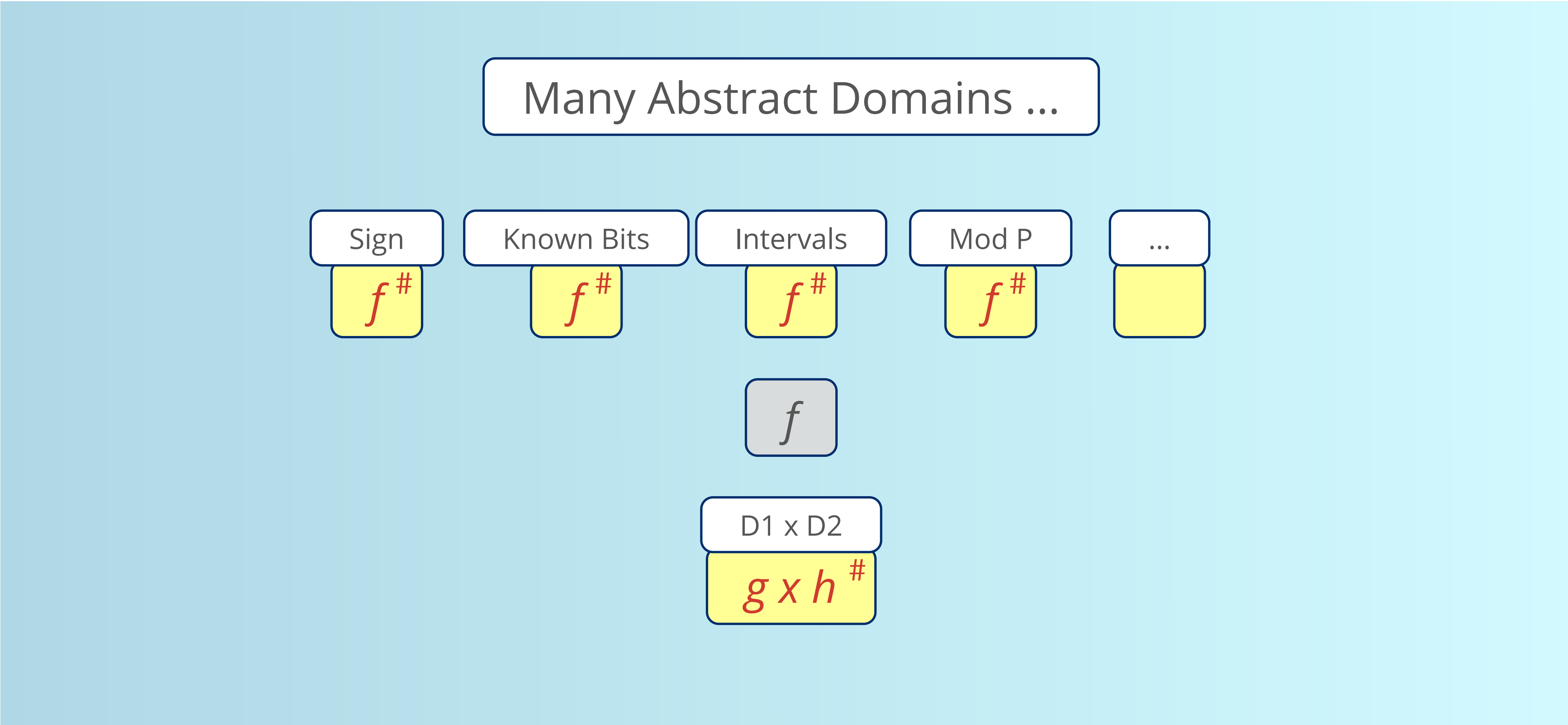
```
ConstantRange
ConstantRange::add(const ConstantRange &Other) const {
  if (isEmptySet() || Other.isEmptySet())
    return getEmpty();
  if (isFullSet() || Other.isFullSet())
    return getFull();

  APInt NewLower = getLower() + Other.getLower();
  APInt NewUpper = getUpper() + Other.getUpper() - 1;
  if (NewLower == NewUpper)
    return getFull();

  ConstantRange X = ConstantRange(std::move(NewLower), std::move(NewUpper));
  if (X.isSizeStrictlySmallerThan(*this) ||
      X.isSizeStrictlySmallerThan(Other))
    // We've wrapped, therefore, full set.
    return getFull();
  return X;
}
```

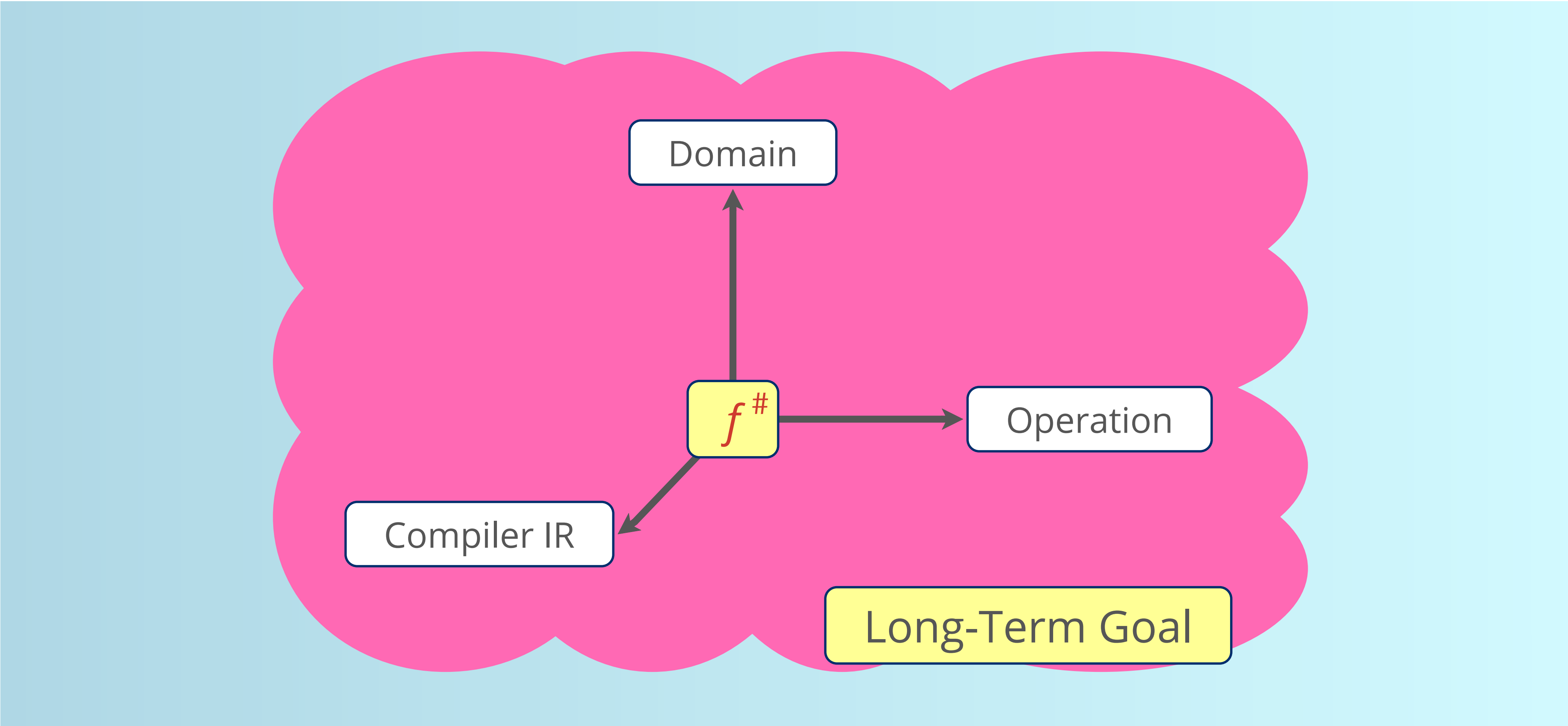
Many Abstract Domains ...

Intervals

$f^{\#}$

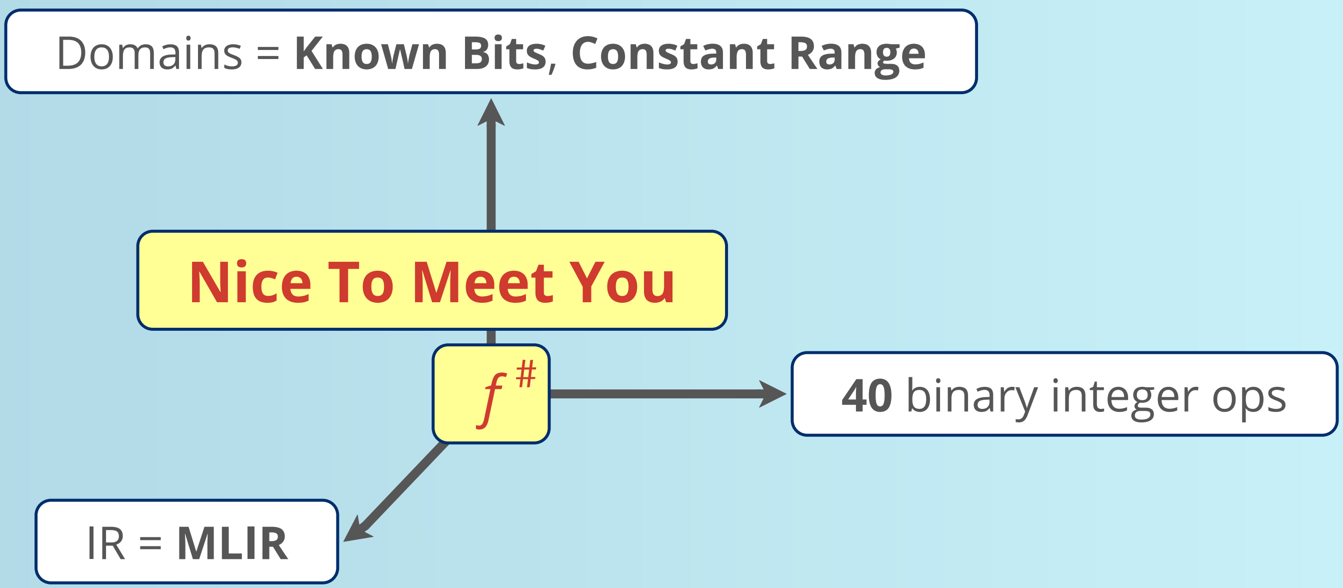
f


Many Abstract Domains ...
Sign
$f^{\#}$
Known Bits
$f^{\#}$
Intervals
$f^{\#}$
Mod P
$f^{\#}$
...
f
D1 x D2
$g \times h^{\#}$


Domain
$f^{\#}$
Operation
Compiler IR
Long-Term Goal

Design point: **Bulk Automation**

- Minimize user input
- Coverage over precision 


Domains = Known Bits, Constant Range
Nice To Meet You
$f^{\#}$
40 binary integer ops
IR = MLIR

Domains = **Known Bits**, **Constant Range**

Nice To Meet You

$f^{\#}$

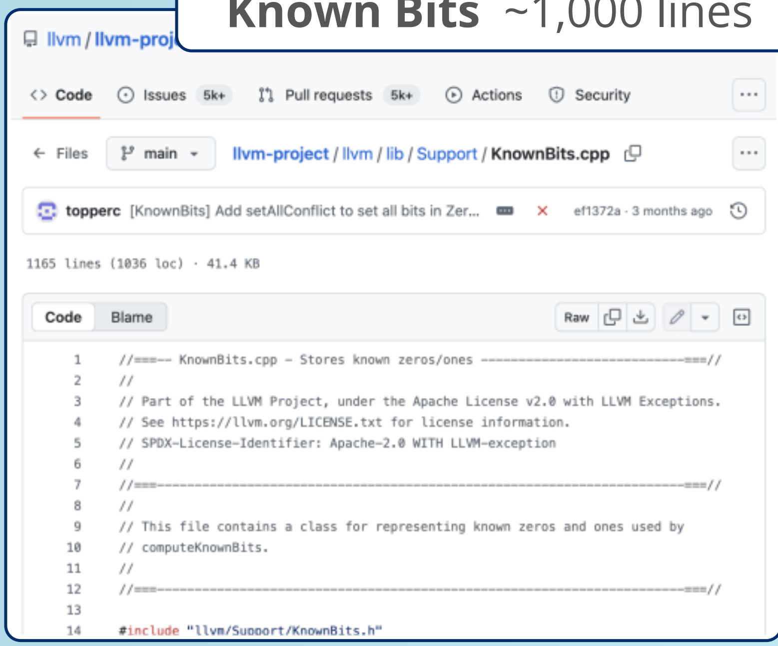
40 binary integer ops

26 ops ~ **complementary** to LLVM
14 ops ~ more precise

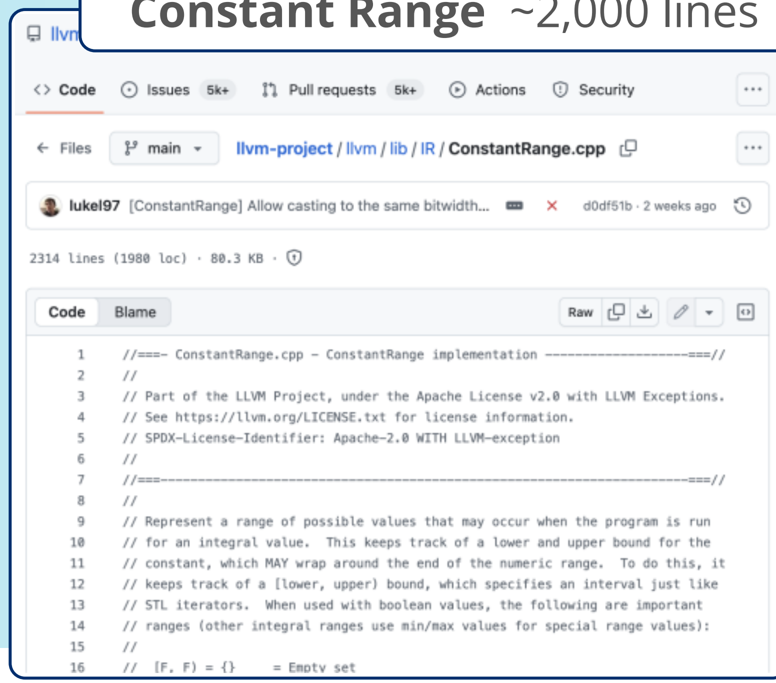
IR = **MLIR**

Nice To Meet You filled edge-cases in 10+ yr old transformers

Known Bits ~1,000 lines



Constant Range ~2,000 lines



Known Bits

Constant Range

Known Bits

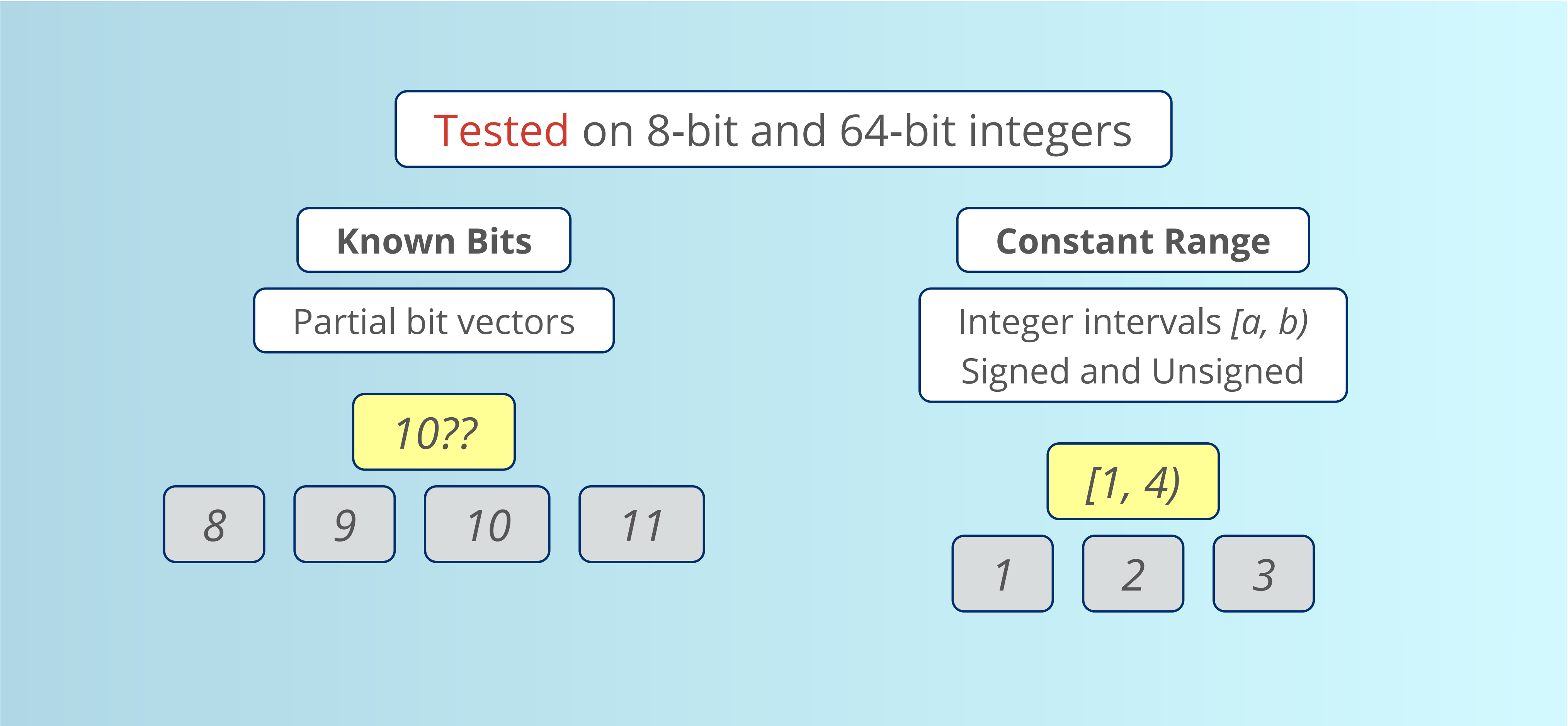
Constant Range

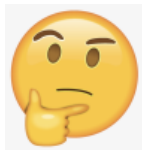
Integer intervals *[a, b)*

Signed and Unsigned

[1, 4)

1 *2* *3*


Tested on 8-bit and 64-bit integers
Known Bits
Partial bit vectors
10??
8
9
10
11
Constant Range
Integer intervals [a, b)
Signed and Unsigned
[1, 4)
1
2
3



... but synthesis doesn't scale?

... but synthesis doesn't scale?

... but synthesis doesn't scale?

Idea : build up a **meet** of several transformers

$$f1^{\#} \sqcap f2^{\#} \sqcap \ldots \sqcap fn^{\#}$$

- **dynamically adapt** score function to cover input space

Give each transformer **one simple job**

`Add` `AddNsw`

`AddNswNuw` `AddNuw`

Algorithm 2: $\texttt{MCMCSynthesizeTransformer}(\mathcal{F}^s, prob)$

Input: $\mathcal{F}^s$ — *Current set of synthesized transformers;*

Output: *A new sound transformer* $f \in \mathcal{L}$ *that (in the limit) minimizes precision.*

fun $\texttt{Cost}(f^\#)$:

  return $\lambda(1-\mathsf{Soundness}(f^\#)) + \kappa(1-\mathsf{Improvement}(f^\#, \mathcal{F}^s))$ // reward soundness, precision improv.

$f \leftarrow \texttt{initialize}()$ // random initial program

for $i \leftarrow 1$ **to** N_{step} **do**

  $f' \leftarrow \texttt{mutate}(f)$ // mutate current candidate

  $p \sim \mathcal{U}(0,1)$ // sample acceptance threshold

  if $\texttt{Cost}(f) - \texttt{Cost}(f') > T \cdot \log(p)$ **then**

    $f \leftarrow f'$ // accept proposed candidate

if $\mathsf{Soundness}(f) < 1$ **then**

  return $\top$ // return trivial top transformer if no sound one found

return f // return the lowest cost sound transformer found

Example: Interval Max

Ideal

$f^{\#}(a, b)$ = [**max**(a.lo, b.lo), **max**(a.hi, b.hi)]

Example: Interval Max

Ideal

$f^{\#}(a, b)$ = [**max**(a.lo, b.lo), **max**(a.hi, b.hi)]

Synth

$f1^{\#}(a, b)$ = [0, **max**(a.hi, b.hi)]

$f2^{\#}(a, b)$ = [a **&** b, MAX_INT]

$f3^{\#}(a, b)$ = [a.lo, a.hi **|** b.hi]

$f4^{\#}(a, b)$ = [b.lo, MAX_INT]

Example: Interval Max

Input: the Domain and Op

Concretization:	$\gamma([a.l, a.r]) = \{a.l, \cdots, a.r\}$
Meet:	$a \sqcap b = [\max(a.l, b.l), \min(a.r, b.r)]$
Join:	$a \sqcup b = [\min(a.l, b.l), \max(a.r, b.r)]$
Abstraction:	$\beta(x) = [x, x]$
Concrete op:	$f(x, y) = \max(x, y)$
DSL ops:	$\{+, -, \&, \mid, \min, \max, \cdots\}$
Size:	$\lvert a \rvert = \lfloor \log_2(\lvert a.l - a.r \rvert) \rfloor$

What are we looking for?

$$\mathcal{P}(C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$$

The best abstract transformer:

$f^{\#} = \alpha \circ f^{*} \circ \gamma$

Algorithm 2: $\texttt{MCMCSynthesizeTransformer}(\mathcal{F}^s, prob)$

Input: $\mathcal{F}^s$ — *Current set of synthesized transformers;*

Output: *A new sound transformer* $f \in \mathcal{L}$ *that (in the limit) minimizes precision.*

fun $\texttt{Cost}(f^\#)$:

  return $\lambda(1-\texttt{Soundness}(f^\#)) + \kappa(1-\texttt{Improvement}(f^\#, \mathcal{F}^s))$ // reward soundness, precision improv.

$f \leftarrow \texttt{initialize}()$ // random initial program

for $i \leftarrow 1$ **to** N_{step} **do**

  $f' \leftarrow \texttt{mutate}(f)$ // mutate current candidate

  $p \sim \mathcal{U}(0, 1)$ // sample acceptance threshold

  if $\texttt{Cost}(f) - \texttt{Cost}(f') > T \cdot \log(p)$ **then**

    $f \leftarrow f'$ // accept proposed candidate

if $\texttt{Soundness}(f) < 1$ **then**

  return $\top$ // return trivial top transformer if no sound one found

return f // return the lowest cost sound transformer found

Algorithm 2: $\texttt{MCMCSynthesizeTransformer}(\mathcal{F}^s, prob)$

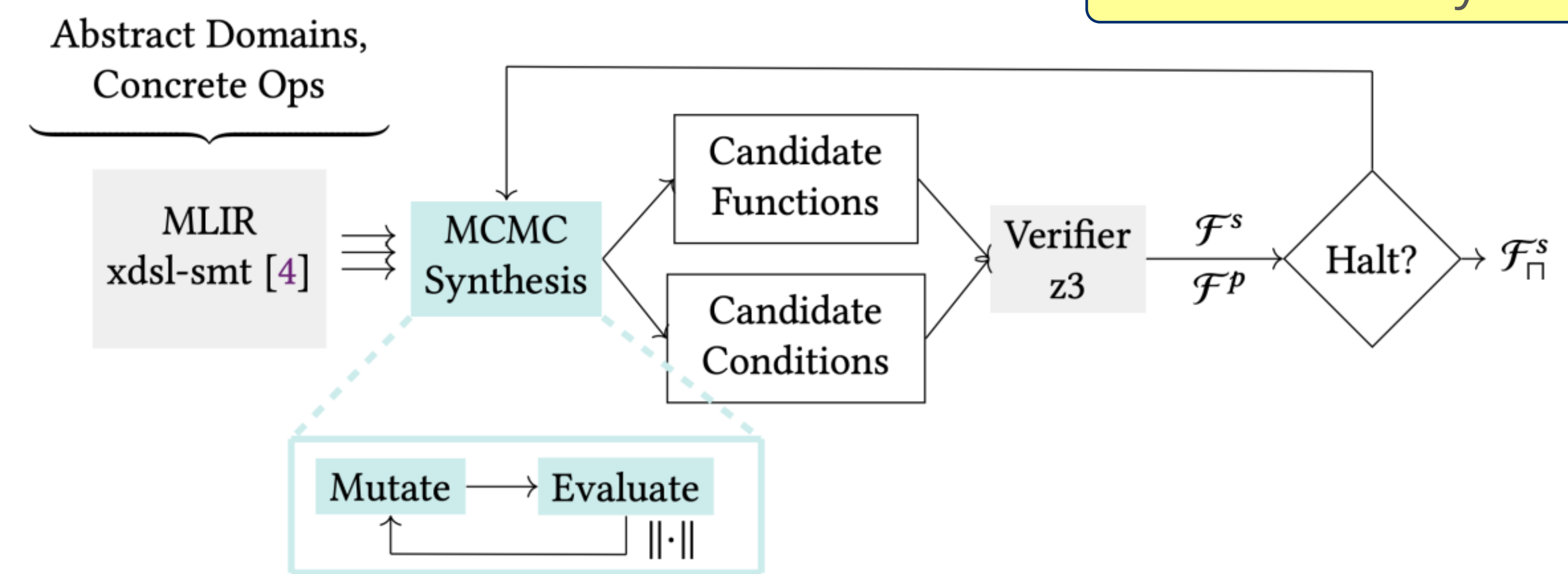
Input: $\mathcal{F}^s$ — *Current set of synthesized transformers;*
Output: *A new sound transformer $f \in \mathcal{L}$ that (in the limit) minimizes precision.*
fun $\texttt{Cost}(f^\#)$:
  return $\lambda(1 - \texttt{Soundness}(f^\#)) + \kappa(1 - \texttt{Improvement}(f^\#, \mathcal{F}^s))$ // reward soundness, precision improv.
$f \leftarrow \texttt{initialize}()$ // random initial program
for $i \leftarrow 1$ **to** N_{step} **do**
  $f' \leftarrow \texttt{mutate}(f)$ // mutate current candidate
  $p \sim \mathcal{U}(0, 1)$ // sample acceptance threshold
  if $\texttt{Cost}(f) - \texttt{Cost}(f') > T \cdot \log(p)$ **then**
    $f \leftarrow f'$ // accept proposed candidate
if $\texttt{Soundness}(f) < 1$ **then**
  return $\top$ // return trivial top transformer if no sound one found
return f // return the lowest cost sound transformer found

Optimal *in the limit*
mutate is invertible

Sound at each step

Nice To Meet You

1. Eval + Verify
2. Condition Synth



https://github.com/dominicmkennedy/synth-xfer

MLIR

https://mlir.llvm.org/

Multi-Level IR Compiler Framework

SSA : Single Static Assignment

One syntax, many dialects

MLIR

https://mlir.llvm.org/

Multi-Level IR Compiler Framework

SSA : Single Static Assignment

One syntax, many dialects

Our dialect: **xDSL-SMT**

U. Cambridge

https://github.com/opencompl/xdsl-smt

CGO'25 **PLDI'25**

Example: MLIR vs. SMT-LIB

```
x * 2 != x + x
```

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

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

Synthesized **urem**

```
func.func @f1(%L : KnownBits, %R : KnownBits) -> KnownBits {
    %1 = countLeadingZero(%L.zero)
    %knownZero = setHighBits(0, %1)
    return makeKnownBits(%knownZero, 0)
}
func.func @f2_cond(%L: KnownBits, %R: KnownBits) -> bool {
    %Lmax = negate(%L.zero)
    %Rmin = %R.one
    %cond = unsignedLessThan(%Lmax, %Rmin)
    return %cond
}
func.func @f2_body(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return %L
}
func.func @f2(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return ite(@f2_cond(%L, %R), @f2_body(%L, %R), %top)
}
...
func.func @solution(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return meet(@f1(%L, %R), ... , @f9(%L, %R))
}
```

Synthesized **urem**

Meet of 9 candidates

Recovers LLVM heuristics

f2 improves precision

```
func.func @f1(%L : KnownBits, %R : KnownBits) -> KnownBits {
    %1 = countLeadingZero(%L.zero)
    %knownZero = setHighBits(0, %1)
    return makeKnownBits(%knownZero, 0)
}
func.func @f2_cond(%L: KnownBits, %R: KnownBits) -> bool {
    %Lmax = negate(%L.zero)
    %Rmin = %R.one
    %cond = unsignedLessThan(%Lmax, %Rmin)
    return %cond
}
func.func @f2_body(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return %L
}
func.func @f2(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return ite(@f2_cond(%L, %R), @f2_body(%L, %R), %top)
}
...
func.func @solution(%L : KnownBits, %R : KnownBits) -> KnownBits {
    return meet(@f1(%L, %R), ... , @f9(%L, %R))
}
```

Evaluation 1: Per-Operator Precision

Complementary on 26 / 40 ops

Known Bits

ConcreteOp	$\#f^\#$	$\#c$	#inst	Tests	$\top$	synth	llvm	meet	Tests	64-bit precision (norm) ↓ $\top$	synth	llvm	meet
Abds	10	3	189	1000	33.90	60.10	100.00	100.00	10000	0.059	0.050	0.000	0.000
Abdu	16	4	259	1000	33.10	59.40	100.00	100.00	10000	0.059	0.050	0.000	0.000
Add	17	2	306	1000	29.60	58.70	100.00	100.00	10000	0.140	0.082	0.000	0.000
AddNsw	13	2	205	1000	24.50	42.00	100.00	100.00	9674	0.147	0.136	0.000	0.000
AddNswNuw	14	3	220	1000	7.40	45.50	100.00	100.00	7479	0.160	0.136	0.000	0.000
AddNuw	17	4	291	1000	15.20	53.90	100.00	100.00	8305	0.152	0.103	0.000	0.000

Constant Range

ConcreteOp	$\#f^\#$	$\#c$	#inst	Tests	$\top$	synth	llvm	meet	Tests	64-bit precision (norm) ↓ $\top$	synth	llvm	meet
Abds*	3	2	70	1000	59.80	59.80	N/A	**59.80**	10000	0.917	0.915	N/A	**0.915**
Abdu	20	6	344	1000	0.00	75.00	N/A	**75.00**	10000	0.990	0.908	N/A	**0.908**
Add	2	2	45	509	36.54	100.00	100.00	100.00	4991	0.949	0.887	0.887	0.887
AddNsw*	8	4	148	1000	7.10	100.00	100.00	100.00	9770	0.982	0.905	0.905	0.905
AddNswNuw	10	2	172	1000	0.00	70.70	84.80	**88.60**	8190	0.994	0.921	0.912	**0.910**
AddNuw	14	5	243	1000	0.00	94.00	100.00	100.00	8267	0.993	0.910	0.906	0.906

Evaluation 2: SPEC CPU 2017 Known Bits

Evaluation 2: SPEC CPU 2017 Known Bits

Meet of synth & LLVM transformers

	# KB	Baseline KB
`openssl`	+2	1.3 M
`ffmpeg`	+14	3.7 M
`cvc5`	+0	16 M

Evaluation 2: SPEC CPU 2017 Known Bits

Synth **vs.** LLVM-- *Synth always loses!*

	Precision	Time
`perlbench`	-3.76%	-1.5s
`gcc`	-1.78%	-6s
`mcf`	0	-0.2s
`omnetpp`	-0.24%	-0.1s
`xalancbmk`	-6.92%	-1s
`x264`	-11.79%	-6s
`deepsjeng`	-5.12%	-0.3s
`leela`	-23.22%	-0.2s
`xz`	-6.26%	-0.5s

Evaluation 3: Product: Known Bits x Constant Range

Concrete Op	Tests	8-bit exact (%) ↑			Tests	64-bit precision (norm) ↓		
		⊤	synth	reduced		⊤	synth	reduced
Abds	1000	7.64	18.06	**28.98**	10000	0.1260	0.1119	**0.1069**
Abdu	1000	7.11	20.76	**71.06**	10000	0.1236	0.1085	**0.0921**
AddNsw	1000	6.68	18.73	**81.65**	10000	0.2820	0.1125	**0.0869**
AddNswNuw	1000	0.22	44.89	**88.20**	10000	0.5592	0.1216	**0.0610**
AddNuw	1000	3.35	31.80	**92.32**	10000	0.4920	0.0930	**0.0557**
AvgCeilS	1000	9.83	18.01	**29.16**	10000	0.1651	0.1573	**0.1503**
AvgFloorS	1000	9.86	17.37	**46.61**	10000	0.1669	0.1598	**0.1412**
AvgFloorU	1000	9.90	19.00	**50.37**	10000	0.1668	0.1481	**0.1336**
Sdiv	1000	17.99	27.85	**45.61**	10000	0.7262	0.2493	**0.2229**
Smax	1000	0.44	59.89	**83.19**	10000	0.4959	0.0926	**0.0822**
Smin	1000	0.43	59.62	**84.23**	10000	0.4954	0.0953	**0.0843**
Srem	1000	13.14	22.41	**26.92**	10000	0.1845	0.1701	**0.1689**
SshlSat	1000	4.08	33.43	**43.35**	10000	0.9542	0.3211	**0.3148**
SubNswNuw	1000	0.33	39.01	**77.20**	10000	0.5617	0.1299	**0.0701**
SubNuw	1000	3.52	36.84	**90.94**	10000	0.4822	0.0763	**0.0466**
UaddSat	1000	4.11	61.03	**83.09**	10000	0.4482	0.0874	**0.0469**
Udiv	1000	0.00	68.66	**75.28**	10000	0.9845	0.0134	**0.0067**
UdivExact	1000	0.02	3.21	**5.78**	10000	1.0000	0.0272	**0.0195**
Umax	1000	0.54	95.28	**99.74**	10000	0.4947	0.0016	**0.0001**
Umin	1000	0.56	92.99	**99.59**	10000	0.4964	0.0023	**0.0003**
Urem	1000	2.12	61.45	**66.53**	10000	0.2677	0.0393	**0.0367**
UsubSat	1000	4.06	56.03	**73.09**	10000	0.4508	0.1106	**0.0700**

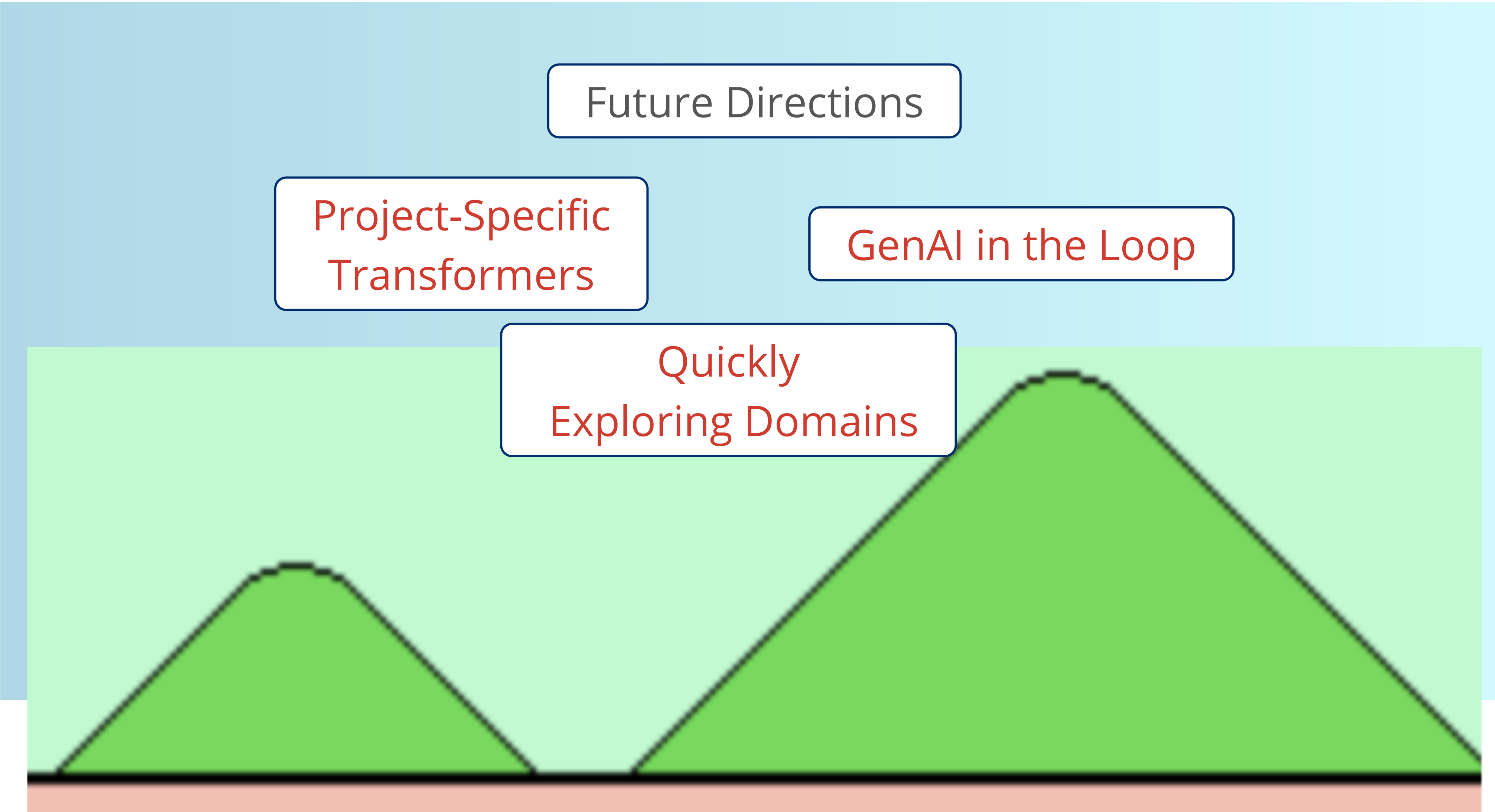
Evaluation 4: Specialization

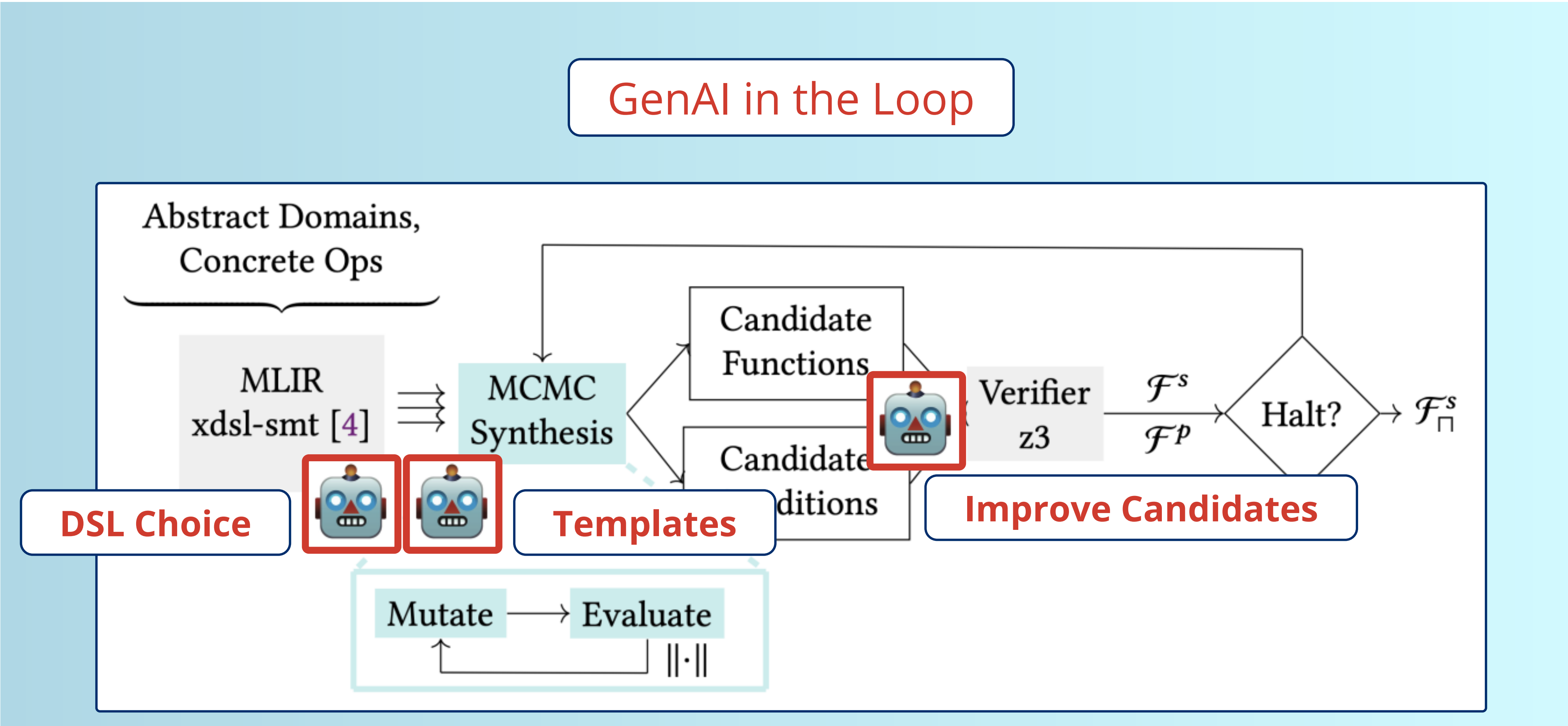
Observation: tranformers compose poorly

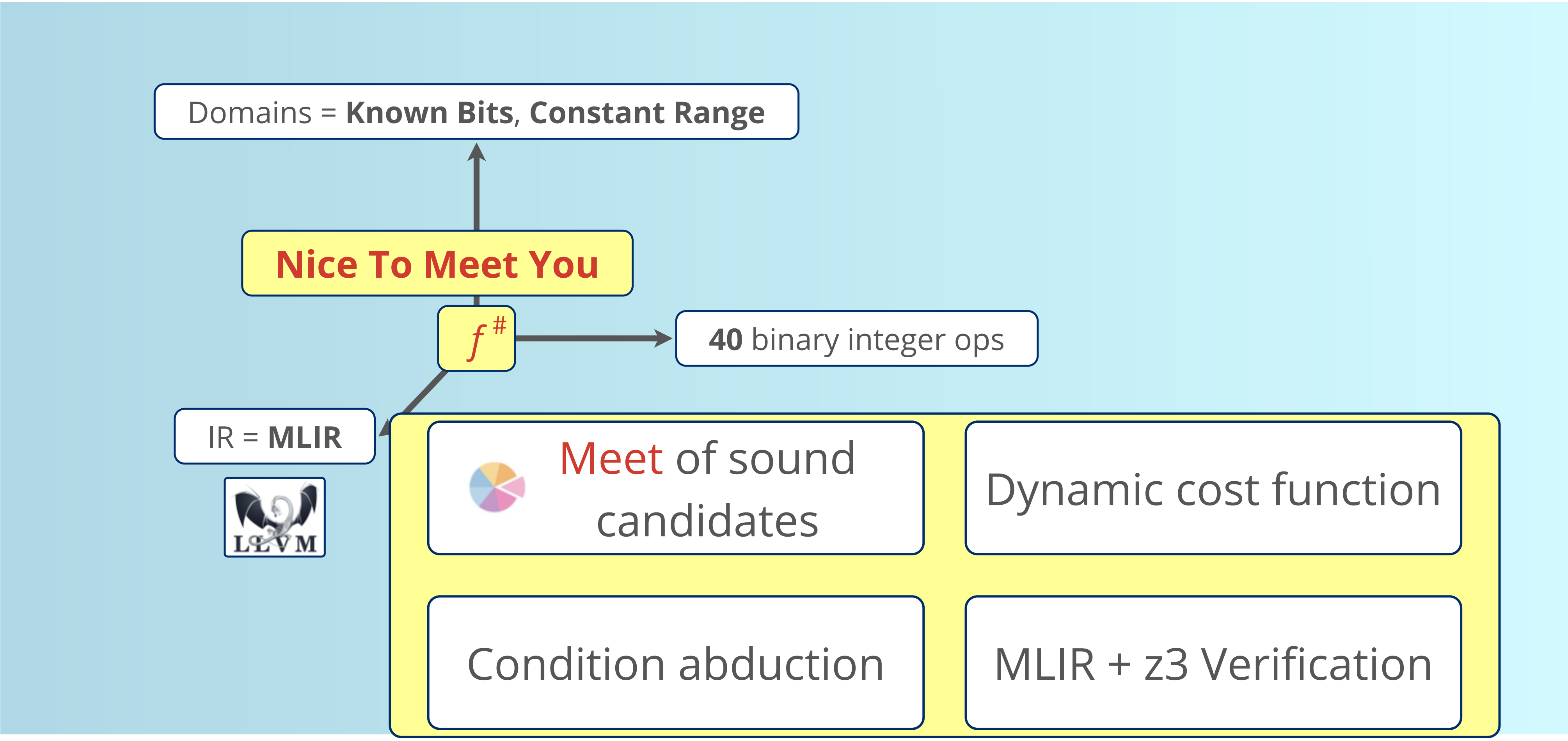
$f2^{\#} \circ f1^{\#} < (f1+f2)^{\#}$

Evaluation 4: Specialization

Category	Concrete Op	Exact (%) ↑			Precision (norm) ↓		
		⊤	composed	synth	⊤	composed	synth
Unary Functions (6,561 test cases)	Abs	1.95	3.92	**100.00**	3372	2552	**0**
	CountRZero	0.00	33.33	**83.63**	5740	3553	**193**
	CountLZero	0.00	0.00	**83.63**	5740	5466	**193**
	PopCount	0.00	0.05	**69.53**	4461	4456	**369**
Binary Functions (43,046,721 test cases)	Smax	4.46	6.33	**56.86**	19,797,600	18,179,000	**5,471,520**
	Smin	4.46	6.04	**70.39**	19,797,600	18,384,100	**4,117,500**
	UaddSat	13.93	22.93	**56.20**	17,358,900	14,111,200	**4,471,530**
	UsubSat	13.93	19.46	**49.11**	17,358,900	15,377,400	**5,369,170**


Future Directions
Project-Specific Transformers
GenAI in the Loop
Quickly Exploring Domains


GenAI in the Loop
Abstract Domains,
Concrete Ops
MLIR
xdsl-smt [4]
MCMC
Synthesis
Candidate
Functions
Candidate
ditions
Verifier
z3
$\mathcal{F}^s$
$\mathcal{F}^p$
Halt?
$\mathcal{F}^s_\sqcap$
Mutate
Evaluate
$\|\cdot\|$
DSL Choice
Templates
Improve Candidates



Domains = **Known Bits, Constant Range**
Nice To Meet You
$f^{\#}$
40 binary integer ops
IR = **MLIR**
LLVM
Meet of sound candidates
Dynamic cost function
Condition abduction
MLIR + z3 Verification

Related Work

Synthesizing Abstract Transformers **OOPSLA'22**
Kalita, Muduli, D'Antoni, Reps, Roy

Automatic Synthesis of Abstract Operators for eBPF **eBFP'25**
Vishwanathan, Shachnai, Narayana, Nagarakatte

Synthesizing Sound and Precise Abstract Transformers for Nonlinear Hyperbolic PDE Solvers **OOPSLA'25**
Laurel, Laguna, Huckelheim

Cost-Driven Synthesis of Sound Abstract Interpreters **arxiv'25**
Gu, Singh, Singh

Definition 2.3 (Transformer Synthesis Problem). *Given a concrete transformer $f : C^k \to C$, an abstract domain $(\mathcal{A}, \top, \gamma, \sqcap, \sqcup, \beta)$, a norm function $\|\cdot\| : (\mathcal{A}^k \to \mathcal{A}) \to \mathbb{N}$, and a DSL $\mathcal{L}$, the transformer synthesis problem is to find a set of transformers $\mathcal{F} = \{f_1^\#, f_2^\#, \cdots, f_n^\#\}$ in $\mathcal{L}$ such that*

- *Their meet $\mathcal{F}_\sqcap$ is sound:* $\mathsf{sound}(\mathcal{F}_\sqcap)$.
- *The norm of $\mathcal{F}_\sqcap$ is minimal, i.e., there is no sound set of transformers $\mathcal{G}$ such that $\|\mathcal{G}_\sqcap\| < \|\mathcal{F}_\sqcap\|$.*
- *No $f_i^\# \in \mathcal{F}$ is redundant:* $\forall f_i^\# \in \mathcal{F}, \exists \vec{a} \in \mathcal{A}^k, \left(\prod_{f^\# \in \mathcal{F} \setminus \{f_i^\#\}} f^\#(\vec{a})\right) \not\sqsubseteq f_i^\#(\vec{a})$.