DrNLA: Dual Rewriting for Branching-Time Verification of Non-Linear Arithmetic Programs

Y. Cyrus Liu [1] Ton-Chanh Le [1] Timos Antonopoulos [2]
Eric Koskinen [1] ThanhVu Nguyen [3]

[1]Stevens Institute of Technology

[2]Yale University

[3]George Mason University

NJPLS • Oct 21st, 2022

Solving the Problem

Goal

- Want to verify temporal properties of programs with **polynomials**?
- Unfortunately existing CTL tools **are unsound**! (T2, FUNCTION)

Solving the Problem

Goal

- Want to verify temporal properties of programs with **polynomials**?
- Unfortunately existing CTL tools **are unsound**! (T2, FUNCTION)

Our Results

- We can make these tools support nonlinear arithmetic (NLA).
- We will show you a method to synthesize linear integer arithmetic (LIA) replacements for NLA expressions.
- *Dual Rewriting* algorithm, with dynamic and static analysis.
- New tool DRNLA.
- After pre-processing with DRNLA, existing CTL verifiers can be employed.

A program with a polynomial loop guard

```
int y=1, z=6, c=0, p=2;
int k=*;
while (z*z - 12y - 6z + 12 + c <= k) :
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

A program with a polynomial loop guard

```
int y=1, z=6, c=0, p=2;
int k=*;
while (z*z - 12y - 6z + 12 + c <= k) :
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

What properties can we prove of such programs?

- Reachability, e.g. DIG (TOSEM'14).
- Termination, e.g. DynamiTe (OOPSLA'20) and νTerm (FSE'22).
- LTL, some support, e.g. Ultimate.
- CTL, none.

A program with a polynomial loop guard

Our work: Synthesize LIA replacements for NLA expressions!

```
int y=1, z=6, c=0, p=2;
int k=*;
while(z*z - 12y - 6z + 12 + c <= k) :
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

A program with a polynomial loop guard

Our work: Synthesize LIA replacements for NLA expressions!

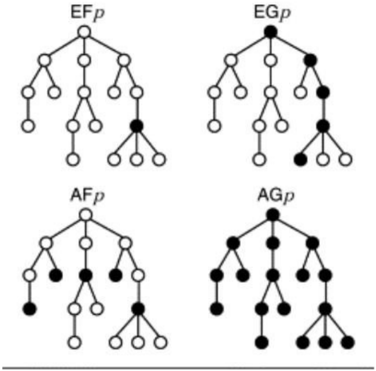
```
int y=1, z=6, c=0, p=2;
int k=*;
while(z*z - 12y - 6z + 12 + c <= k):
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

```
int y=1, z=6, c=0,
  p=2;
int k=*;
while(c <= k):
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

Outcome: Can preprocess NLA programs and then use CTL verifiers.

What is special about CTL? A Reminder.

Safety and liveness properties, with particular focus on **branching**.



(LTL has no branching; implicitly over all paths, one at a time.)

CTL Example

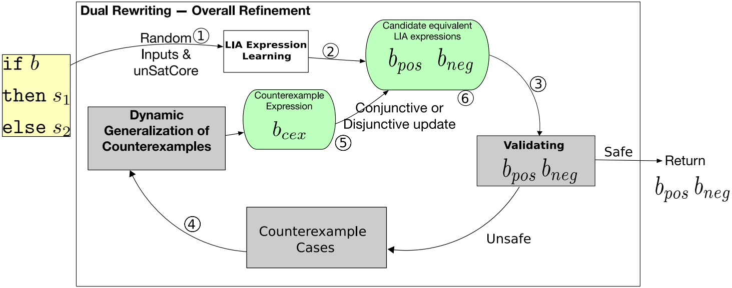
```
int y=1, z=6, c=0, p=2;
int k=*;
while( z*z-12y-6z+12+c<= k ):
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

Valid: $\mathsf{EF}(p = 0) \wedge \mathsf{EF}(p = 1)$

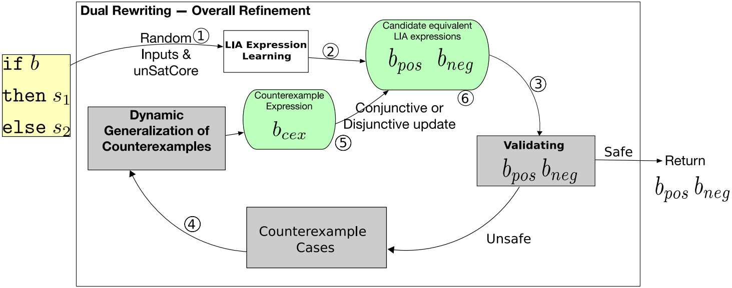
```
int y=1, z=6, c=0, p=2;
int k=*;
while( z*z-12y-6z+12+c<=k ):
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = c-k;
return 0;
```

Invalid: $\mathsf{EF}(p = 0) \wedge \mathsf{EF}(p = 1)$

Our Approach: "Dual Rewriting" with DrNLA



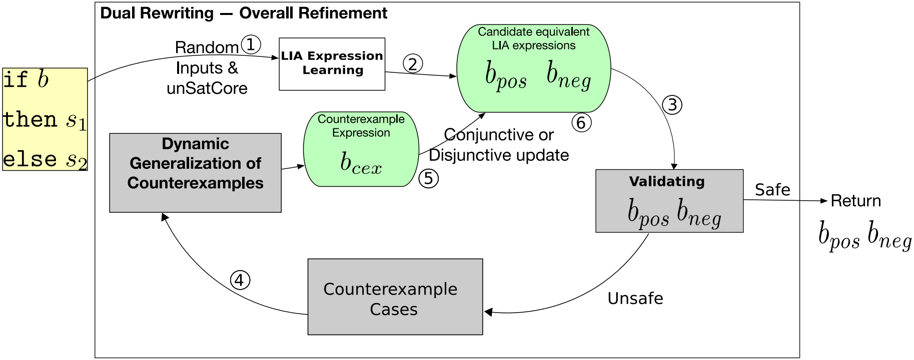
Our Approach: "Dual Rewriting" with DRNLA



Dual Rewriting algorithm

Challenge: Static tools struggle with NLAs.

- **Solution: dual re-writing** algorithm that mixes static and dynamic
- Iteratively synthesize LIA Boolean combinations denoted b_{pos} and b_{neg}.
- **Output:** equivalent LIA expressions of NLA expressions.



Challenges

- Identify all NLA expressions (b) from the program.
- How to know which cases to validate?
- How to do that within program context?

DrNLA Running Example

```
int y=1, z=6, c=0, p=2;
int k=*;
while (true):
  if ((((z * z - 12 * y) - 6 * z) + 12) + c <= k):
      vtrace_if_35(k, n, y, z, c, p) ; //b_pos
  else:
      vtrace_else_35(k, n, y, z, c, p) ; //b_neg
      break;
  y = y + z;
  z = z + 6;
  c = c + 1;
  p = 1;
p = 0;
return 0;
```

```
//Concrete Traces
if_35; k; n; y; z; c; p
294; 78; 18487; 474; 78; 1
271; 57; 9919;  348; 57; 1
26;  8;  217;   54;  8;  1
296; 53; 8587;  324; 53; 1
...
else_35; k; n; y; z; c; p
22; 23; 1657; 144; 23; 1
11; 12; 469;  78;  12; 1
21; 22; 1519; 138; 22; 1
0;  1;  7;    12;  1;  1
...
```

- b: original NLA expression.
- The goal is to an LIA expression b_{pos}, representing b as well as LIA expression b_{neg} representing $\neg b$.

DrNLA Running Example

```
int y=1, z=6, c=0, p=2;
int k=*;
while (true):
  if ((((z * z - 12 * y) - 6 * z) + 12) + c <= k):
        vtrace_if_35(k, n, y, z, c, p) ; //b_pos
  else:
        vtrace_else_35(k, n, y, z, c, p) ; //b_neg
        break;
  y = y + z;
  z = z + 6;
  c = c + 1;
  p = 1;
p = 0;
return 0;
```

```
//Concrete Traces
if_35; k; n; y; z; c; p
294; 78; 18487; 474; 78; 1
271; 57; 9919;  348; 57; 1
26;  8;  217;   54;  8;  1
296; 53; 8587;  324; 53; 1
...
else_35; k; n; y; z; c; p
22; 23; 1657; 144; 23; 1
11; 12; 469;  78;  12; 1
21; 22; 1519; 138; 22; 1
0;  1;  7;    12;  1;  1
...
```

- b: original NLA expression.
- The goal is to an LIA expression b_{pos}, representing b as well as LIA expression b_{neg} representing $\neg b$.

1. Initial guess, learned from random input (100).

$$b_{pos}(b): \quad \{2 \geq p, -p \leq -1, 0 = -6 \times c + z - 6, -p - z \leq -8, 0 \geq -c, 0 \geq c - k\}$$
$$b_{neg}(\neg b): \quad \{0 \geq -c + p, 0 \geq -k, 0 = -6 \times k + z - 12, 0 = p - 1, 0 = c - k - 1\}$$

DrNLA Running Example

```
int y=1, z=6, c=0, p=2;
int k=*;
while (true):
  if ((((z * z - 12 * y) - 6 * z) + 12) + c <= k):
      vtrace_if_35(k, n, y, z, c, p) ; //b_pos
  else:
      vtrace_else_35(k, n, y, z, c, p) ; //b_neg
      break;
  y = y + z;
  z = z + 6;
  c = c + 1;
  p = 1;
p = 0;
return 0;
```

```
//Concrete Traces
if_35; k; n; y; z; c; p
294; 78; 18487; 474; 78; 1
271; 57; 9919;  348; 57; 1
26;  8;  217;   54;  8;  1
296; 53; 8587;  324; 53; 1
...
else_35; k; n; y; z; c; p
22; 23; 1657; 144; 23; 1
11; 12; 469;  78;  12; 1
21; 22; 1519; 138; 22; 1
0;  1;  7;    12;  1;  1
...
```

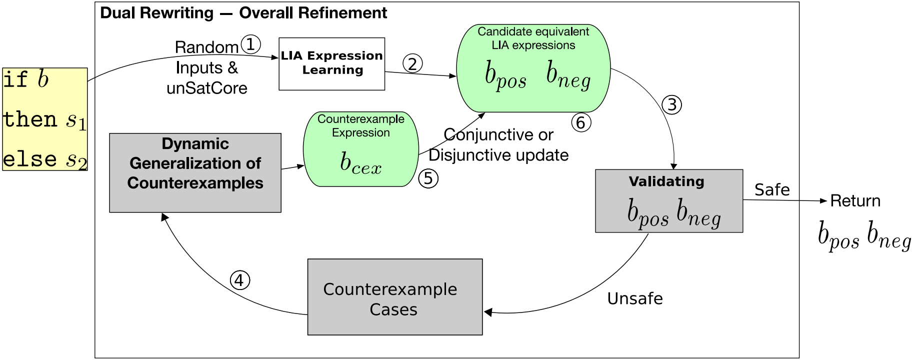
- b: original NLA expression.
- The goal is to an LIA expression b_{pos}, representing b as well as LIA expression b_{neg} representing $\neg b$.

1. Initial guess, learned from random input (100).

$$b_{pos}(b): \quad \{2 \geq p, -p \leq -1, 0 = -6 \times c + z - 6, -p - z \leq -8, 0 \geq -c, 0 \geq c - k\}$$
$$b_{neg}(\neg b): \quad \{0 \geq -c + p, 0 \geq -k, 0 = -6 \times k + z - 12, 0 = p - 1, 0 = c - k - 1\}$$

2. Optimization process, remove identical ones, unsat core.

$$b_{pos} \equiv 0 \geq c - k \qquad b_{neg} \equiv 0 = c - k - 1$$



Dual Rewriting — Overall Refinement
if b
then s_1
else s_2
Random ① Inputs & unSatCore
LIA Expression Learning
②
Candidate equivalent LIA expressions
b_{pos} b_{neg}
③
Validating b_{pos} b_{neg}
Safe
Return b_{pos} b_{neg}
Unsafe
Counterexample Cases
④
Dynamic Generalization of Counterexamples
Counterexample Expression b_{cex}
⑤
Conjunctive or Disjunctive update
⑥

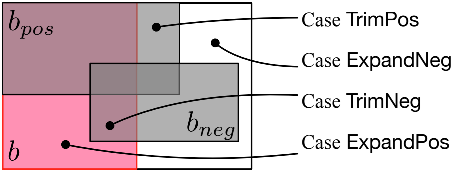
Transformation Example for Static Validation

```
int y=1, z=6, c=0, p=2;
int k=*;
while (true):









  if (z*z-12y-6z+12+c > k):
    break;
  else:
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

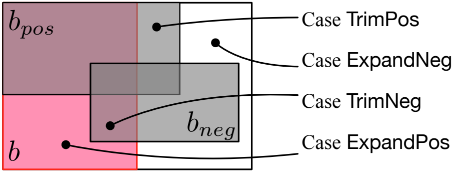
```
int y=1, z=6, c=0, p=2;
int k=*;
while(true):
  if (z*z - 12y - 6z + 12 + c > k && ¬b_pos):
    error_ExpandPos // b_pos too small
  elsif (z*z - 12y - 6z + 12 + c > k && b_neg):
    error_TrimNeg // b_neg too big
  elsif (¬(z*z - 12y - 6z + 12 + c > k) && b_pos):
    error_TrimPos // b_pos too big
  elsif (¬(z*z - 12y - 6z + 12 + c > k) && ¬b_neg):
    error_ExpandNeg // b_neg too small
  if(z*z - 12y - 6z + 12 + c > k):
    break
  else:
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

Figure: Demonstration of instrumentation for static validation.

3. Static Validation for the Pair (b_{pos}, b_{neg}).



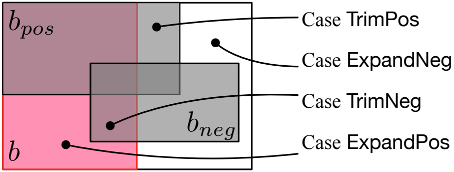
3. Static Validation for the Pair (b_{pos}, b_{neg}).



b_{pos}

TrimPos, it includes executions where $\neg b$ holds and must be trimmed down.
ExpandPos, it does not include all executions where b does hold.

3. Static Validation for the Pair (b_{pos}, b_{neg}).



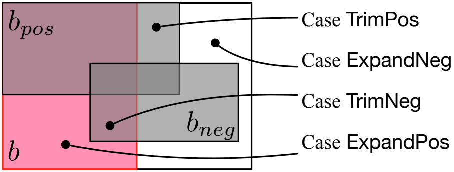
b_{pos}

TrimPos, it includes executions where $\neg b$ holds and must be trimmed down.
ExpandPos, it does not include all executions where b does hold.

b_{neg}

TrimNeg, it includes executions where b holds and must be trimmed down.
ExpandNeg, it does not include all executions where $\neg b$ does hold.

3. Static Validation for the Pair (b_{pos}, b_{neg}).



b_{pos}

TrimPos, it includes executions where $\neg b$ holds and must be trimmed down.
ExpandPos, it does not include all executions where b does hold.

b_{neg}

TrimNeg, it includes executions where b holds and must be trimmed down.
ExpandNeg, it does not include all executions where $\neg b$ does hold.

$$b \equiv c \leq k, \qquad b_{pos} \equiv 0 \geq c - k, \qquad b_{neg} \equiv 0 = c - k - 1$$

For this running example, we need to ExpandNeg.

Counterexamples from Static Validation

```
ℓ1 :  int y=1, z=6, c=0, p=2;
ℓ2 :  int k=*;
ℓ3 :  assume(true);
ℓ12 : assume(!(z*z-12y-6z+12+c>k));
ℓ15 : y = y + z;
ℓ16 : z = z + 6;
ℓ17 : c = c + 1;
ℓ18 : p = 1;
ℓ3 :  assume(true);
```

```
ℓ12 : assume(!(z*z-12y-6z+12+c>k));
ℓ15 : y = y + z;
ℓ16 : z = z + 6;
ℓ17 : c = c + 1;
ℓ18 : p = 1;
ℓ3 :  p = 1;
ℓ8 :  assume(!(z*z-12y-6z+12+c>k));
ℓ8 :  assume(b_pos);
ℓ9 :  error_ExpandPos
```

Category: Reached $\texttt{error}_{\mathsf{ExpandPos}}$. So we need to expand. $(b_{pos} \rightarrow b_{neg})$

Counterexamples from Static Validation

$\ell\,1:$ `int y=1, z=6, c=0, p=2;`	$\ell\,12:$ `assume(!(z*z-12y-6z+12+c>k));`
$\ell\,2:$ `int k=*;`	$\ell\,15:$ `y = y + z;`
$\ell\,3:$ `assume(true);`	$\ell\,16:$ `z = z + 6;`
$\ell\,12:$ `assume(!(z*z-12y-6z+12+c>k));`	$\ell\,17:$ `c = c + 1;`
$\ell\,15:$ `y = y + z;`	$\ell\,18:$ `p = 1;`
$\ell\,16:$ `z = z + 6;`	$\ell\,3:$ `p = 1;`
$\ell\,17:$ `c = c + 1;`	$\ell\,8:$ `assume(!(z*z-12y-6z+12+c>k));`
$\ell\,18:$ `p = 1;`	$\ell\,8:$ `assume` (b_{pos});
$\ell\,3:$ `assume(true);`	$\ell\,9:$ $\texttt{error}_{\text{ExpandPos}}$

Category: Reached $\texttt{error}_{\text{ExpandPos}}$. So we need to expand. $(b_{pos} \to b_{neg})$

Counterexample: $c = 0, k = -2, p = 2, y = 1, z = 6.$
Another counterexample! $c = 0, k = -3, p = 2, y = 1, z = 6.$

Will there be more? Expand with generalization of this counterexample.

Dynamic Counterexample Generalization

Counterexample Models

y	z	c	k	p
1	6	0	-2	2
1	6	0	-3	2
1	6	0	-4	2
1	6	0	-5	2
1	6	0	-6	2
1	6	0	-7	2
1	6	0	-8	2

Dynamic Counterexample Generalization

Counterexample Models

y	z	c	k	p
1	6	0	-2	2
1	6	0	-3	2
1	6	0	-4	2
1	6	0	-5	2
1	6	0	-6	2
1	6	0	-7	2
1	6	0	-8	2

4. Output of DIG:
   - We generate 1000 counterexample models using Z3.
   - Run DIG (dynamic learning) on all these counterexamples

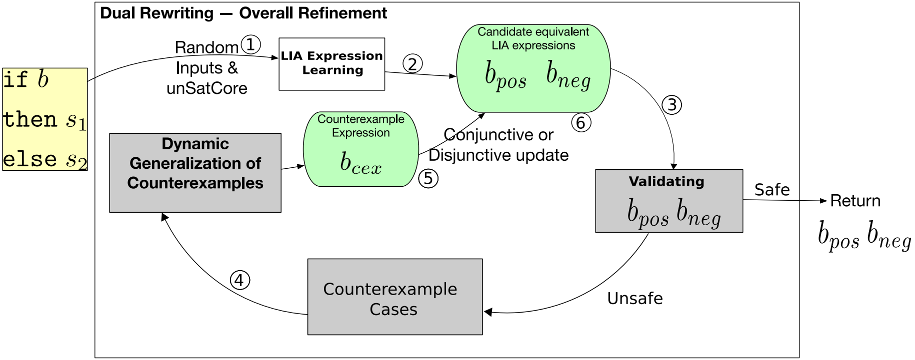
$$b_{cex} \equiv 0 \geq p + k \wedge 0 = p - 2 \wedge 0 = c \ \hat{\vee}\ b_{neg} \equiv 0 = c - k - 1$$

Dynamic Generalization Algorithm

```
procedure dyGeneralize(b_cur, cex, expand):
  S := getModels(formula(cex), iters=1000);
  b_cex := learn(S);
  if (expand):
    match UnsatCorePair(b_cur, b_cex) with
    | Some(b_usc) → return b_usc
    | None → return b_cex
  else:
    return b_cex
```

- Encode counterexample with program context into formula.
- Generate more data with SMT solver Z3.
- Dynamic learn linear invariants from data.
- Return generalized counterexample with optimization.

We are back to the top of loop!



5. ExpandNeg, with the help of convex hull (b_{pos} and b_{neg} are updated).

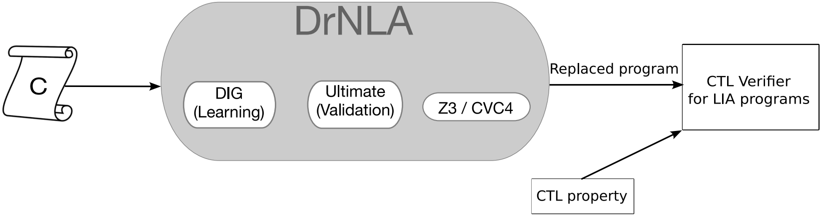
$$b_{pos} \equiv 0 \geq c - k, \qquad b_{neg} \equiv k - c \leq -1$$

6. Static validation returns correct, rewrite the program and prove CTL properties.

DrNLA Implementation

New Tool DrNLA

Built on top of DIG, Ultimate, Z3, CVC4, CIL, etc.



Nonlinear CTL Benchmarks

Note: No CTL Benchmarks for NLA programs, we built new ones!

CTLNLABench-DYNAMITE

NLA benchmarks in the Dynamite termination work [a], insert CTL property $EF(p = 0) \wedge EF(p = 1)$ for all programs (56).

[a] Le *et al.*2020. DynamiTe: Dynamic Termination and Non-termination Proofs

Nonlinear CTL Benchmarks

Note: No CTL Benchmarks for NLA programs, we built new ones!

CTLNLABench-DYNAMITE

NLA benchmarks in the Dynamite termination work [a], insert CTL property $EF(p = 0) \wedge EF(p = 1)$ for all programs (56).

[a] Le *et al.*2020. DynamiTe: Dynamic Termination and Non-termination Proofs

CTLNLABench-PLDI13

26 benchmarks with CTL properties from Terminatior project [a], insert a simple terminating loop that contains NLA expressions.

[a] Cook *et al.*2013. Reasoning about Nondeterminism in Programs.

Nonlinear CTL Benchmarks

Note: No CTL Benchmarks for NLA programs, we built new ones!

CTLNLABench-DYNAMITE

NLA benchmarks in the Dynamite termination work [a], insert CTL property $EF(p = 0) \wedge EF(p = 1)$ for all programs (56).

[a] Le *et al.*2020. DynamiTe: Dynamic Termination and Non-termination Proofs

CTLNLABench-PLDI13

26 benchmarks with CTL properties from Terminatior project [a], insert a simple terminating loop that contains NLA expressions.

[a] Cook *et al.*2013. Reasoning about Nondeterminism in Programs.

Handcrafted Benchmarks

10 benchmarks with Combinations of NLA expressions with higher degrees, its LIA equivalent expressions of these NLA expressions are also more complex and involve disjunctions of linear constraints.

Learning Results

Example output of DrNLA on CTLNLABench-DYNAMITE

Benchmark	Source NLA	Output b_{pos}	Output b_{neg}
bresenham1-T.c	$\ell_{36} : 2Yx - 2X^2y + 2Y - v + c \leq k$	$0 \geq c - k$,	$k - c \leq -1$
cohencu2-T.c	$\ell_{32} : 3n^2 + 3n + 1 \leq k$	$0 \geq y - k$,	$k - y \leq -1$
egcd2-T.c	$\ell_{33} : c \geq xq + ys$	$0 \geq b - c$,	$-b + c \leq -1$

Learning Results

Example output of DrNLA on CTLNLABench-DYNAMITE

Benchmark	Source NLA	Output b_{pos}	Output b_{neg}
`bresenham1-T.c`	$\ell_{36} : 2Yx - 2X^2y + 2Y - v + c \leq k$	$0 \geq c - k$,	$k - c \leq -1$
`cohencu2-T.c`	$\ell_{32} : 3n^2 + 3n + 1 \leq k$	$0 \geq y - k$,	$k - y \leq -1$
`egcd2-T.c`	$\ell_{33} : c \geq xq + ys$	$0 \geq b - c$,	$-b + c \leq -1$

Example output of DrNLA on CTLNLABench-PLDI13

Benchmark	Source NLA	Output b_{pos}	Output b_{neg}
`afefp-T.c`	$\ell_{19} : t^2 - 4s + 2t + 1 + c \leq k$	$0 \geq a - k$	$k - a \leq -1$
`afegp-F.c`	$\ell_{18} : t^2 - 4s + 2t + 1 + c \leq k$	$0 \geq a - k$	$k - a \leq -1$
`afagp-T.c`	$\ell_{21} : \neg(xz - x - y + 1 + c < k)$	$0 \geq -c + k$	$c - k \leq -1$

Rewrite Results

Table: DRNLA's rewrite results for handcrafted benchmarks

Benchmark	Res	T(s)	It.	Benchmark	Res	T(s)	It.
if-cubic-F.c	✔	51.5	3	square-loop-F.c	✔	383.6	12
if-cubic-T.c	✔	51.3	3	square-loop-T.c	✔	233.3	7
if-F.c	✔	78.8	6	while-cubic-F.c	✔	87.0	7
if-T.c	✔	76.5	6	while-cubic-T.c	✔	84.4	7
				while-F.c	✔	190.4	6
				while-T.c	✔	189.6	6

Rewrite Results

Table: DrNLA's rewrite results for handcrafted benchmarks

Benchmark	Res	T(s)	It.	Benchmark	Res	T(s)	It.
if-cubic-F.c	✔	51.5	3	square-loop-F.c	✔	383.6	12
if-cubic-T.c	✔	51.3	3	square-loop-T.c	✔	233.3	7
if-F.c	✔	78.8	6	while-cubic-F.c	✔	87.0	7
if-T.c	✔	76.5	6	while-cubic-T.c	✔	84.4	7
				while-F.c	✔	190.4	6
				while-T.c	✔	189.6	6

An example result for 6 iterations (*if-F.c* | *p=2*).

ℓ_6: $(36 == (x \times x)) \mapsto$

b_{pos}: $((((0 + (x) <= 6) \wedge !(((0 == (p - 2)) \wedge ((-(p) + x) <= -(1)))))||((x <= -(6)) \wedge (-(p) <= -(2)) \wedge (8 >= (p - x)))) \wedge !(((0 == (p - 2)) \wedge (0 >= (p - x)) \wedge (3 >= (-(p) + x))))$

b_{neg}: $(((((0 == (p - 2)) \wedge (0 >= -(x)) \wedge !(((2 >= p) \wedge (-(x) <= -(6)) \wedge (4 >= (-(p) + x)))))|| ((0 == (p - 2)) \wedge ((-(p) + x) <= -(1)))) \wedge !(((x <= -(6)) \wedge (-(p) <= -(2)) \wedge (8 >= (p - x)))))||((0 == (p - 2)) \wedge (0 >= (p - x)) \wedge (3 >= (-(p) + x))))$

Improvements on CTL Tools

Table: DrNLA's improvements for handcrafted benchmarks

Benchmark	Improve T2				Improve FuncTion			
	T2		DrNLA		FT		DrNLA	
	Res	T(s)	Res	T(s)	Res	T(s)	Res	T(s)
`if-cubic-F.c`	↯✓	0.9	**X**	0.7	??	0.1	??	0.1
`if-cubic-T.c`	✓	0.6	✓	0.7	??	0.0	??	0.0
`if-F.c`	↯✓	0.7	**X**	0.7	??	0.0	??	0.1
`if-T.c`	✓	0.7	✓	0.7	??	0.0	??	0.1

Benchmark	Improve T2				Improve FuncTion			
	T2		DrNLA		FT		DrNLA	
	Res	T(s)	Res	T(s)	Res	T(s)	Res	T(s)
`square-loop-F.c`	**X**	0.8	↯✓	0.8	??	0.0	??	0.0
`square-loop-T.c`	✓	0.8	↯**X**	0.9	??	0.1	??	3.1
`while-cubic-F.c`	↯✓	0.7	**X**	0.8	??	0.0	??	0.2
`while-cubic-T.c`	✓	0.7	✓	0.7	??	0.0	??	0.5
`while-F.c`	**X**	0.9	**X**	0.9	??	0.1	??	0.4
`while-T.c`	✓	0.7	✓	0.8	??	0.1	??	3.2

Definition (Imporvements)

Improvements meaning number of benchmarks can be proved now to the total benchmarks ratio.

Experiment Summary

Before DrNLA

- Existing tools simply do not support these NLA programs
- FuncTion returns Unknown
- T2 is Unsound

Experiment Summary

Before DrNLA

- Existing tools simply do not support these NLA programs
- FuncTion returns Unknown
- T2 is Unsound

With DrNLA

- CTLNLABench-DYNAMITE:
  - T2 can now analyze **26 out of the 56** new benchmarks
  - FuncTion correctly analyzes **12 out of the 56** new benchmarks
- CTLNLABench-PLDI13:
  - T2 can now analyze **10 out of the 26** new benchmarks
- Handcrafted Benchmarks:
  - T2 can now analyze **6 out of the 10** new benchmarks

Experiment Summary

Before DrNLA

- Existing tools simply do not support these NLA programs
- FuncTion returns Unknown
- T2 is Unsound

With DrNLA

- CTLNLABench-DYNAMITE:
  - T2 can now analyze **26 out of the 56** new benchmarks
  - FuncTion correctly analyzes **12 out of the 56** new benchmarks
- CTLNLABench-PLDI13:
  - T2 can now analyze **10 out of the 26** new benchmarks
- Handcrafted Benchmarks:
  - T2 can now analyze **6 out of the 10** new benchmarks
- DrNLA's rewrite does not decrease quality of the CTL tools.
- These improvements come with almost no additional runtime cost.

Conclusions

Contributions and Findings

- CTL properties can indeed be verified with NLA programs.
- Dual rewriting technique effectively synthesizes boolean combinations of linear expressions that are equivalent to its NLA counterpart.
- DrNLA can used as a pre-processing step before CTL verification.
- Static verification tools were often useful at validating whether NLA expressions are equivalent to provided LIA alternatives.

Conclusions

Contributions and Findings

- CTL properties can indeed be verified with NLA programs.
- Dual rewriting technique effectively synthesizes boolean combinations of linear expressions that are equivalent to its NLA counterpart.
- DrNLA can used as a pre-processing step before CTL verification.
- Static verification tools were often useful at validating whether NLA expressions are equivalent to provided LIA alternatives.

Future work

- Desired shape of NLA expressions.
- Convergence of DrNLA.
- Other reasons that hurdle CTL verification tools.
- Applications in code efficiency, compiler optimization.

Q & A

Thank You!