Accurate Inference of Termination Conditions

Biting Huang, Zhilei Han, Fei He

April 17, 2026

清華大學
Tsinghua University

Termination Analysis

Termination Analysis

- A program is **terminating** if all its feasible executions are finite.

Termination Analysis

- A program is **terminating** if all its feasible executions are finite.
- Termination is an essential property to establish total correctness of programs.

Termination Analysis

- A program is **terminating** if all its feasible executions are finite.
- Termination is an essential property to establish total correctness of programs.
- Generally, termination verification is an undecidable program.

Termination Analysis

- A program is **terminating** if all its feasible executions are finite.
- Termination is an essential property to establish total correctness of programs.
- Generally, termination verification is an undecidable program.

```
while (n > 1) {
    if (n % 2 == 0)
        n = n / 2;
    else
        n = 3 * n + 1;
}
```

Even for this simple loop, whether it terminates is still undetermined to this day.

Non-termination: Recurrent Set

Non-termination: Recurrent Set

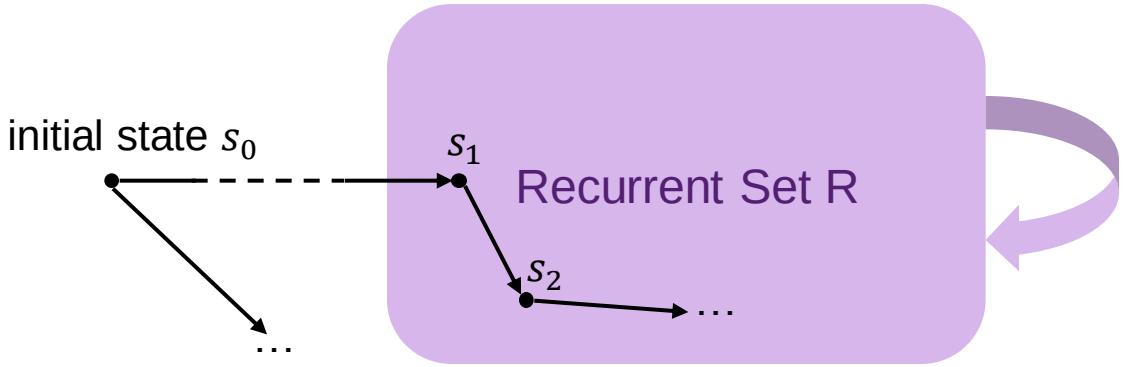
- As the counterpart of termination, non-termination typically admits a **recurrent set** as an argument.

Non-termination: Recurrent Set

- As the counterpart of termination, non-termination typically admits a **recurrent set** as an argument.

A recurrent set is a set of states R satisfying three conditions:

1. R is reachable from an initial state;
2. Any state in R has a successor;
3. For every state s in R, the successor of s remains in R.



Conditional Termination

- However, most programs are neither always terminating nor non-terminating — they may terminate on some inputs but not on others.

Conditional Termination

- However, most programs are neither always terminating nor non-terminating — they may terminate on some inputs but not on others.
- This yields the problem of **conditional termination**, which aims to characterize the condition under which a program terminates.

Conditional Termination

- However, most programs are neither always terminating nor non-terminating — they may terminate on some inputs but not on others.
- This yields the problem of **conditional termination**, which aims to characterize the condition under which a program terminates.

```
while (x > 0) {
    if (x > 5)
        x = x + 1;
    else
        x = x - 1;
}
```

This program terminates when $x \leq 5$

Conditional Termination

- However, most programs are neither always terminating nor non-terminating — they may terminate on some inputs but not on others.
- This yields the problem of **conditional termination**, which aims to characterize the condition under which a program terminates.

```
while (x > 0) {
    if (x > 5)
        x = x + 1;
    else
        x = x - 1;
}
```

This program terminates when $x \leq 5$

A formula φ is a **termination condition** of a program P if P terminates under the condition φ.

Conditional Termination

Conditional Termination

- The accuracy of the termination condition matters.
  - Note that *False* is a valid termination condition for any program.

Conditional Termination

- The accuracy of the termination condition matters.
  - Note that *False* is a valid termination condition for any program.

```
while (x > 0) {
    if (x > 5)
        x = x + 1;
    else
        x = x - 1;
}
```

$False$,

$x \leq 0$,

$x \leq 2$,

…… are all termination conditions

$x \leq 5$ is the accurate one

Conditional Termination

- The accuracy of the termination condition matters.
  - Note that *False* is a valid termination condition for any program.

```
while (x > 0) {
    if (x > 5)
        x = x + 1;
    else
        x = x - 1;
}
```

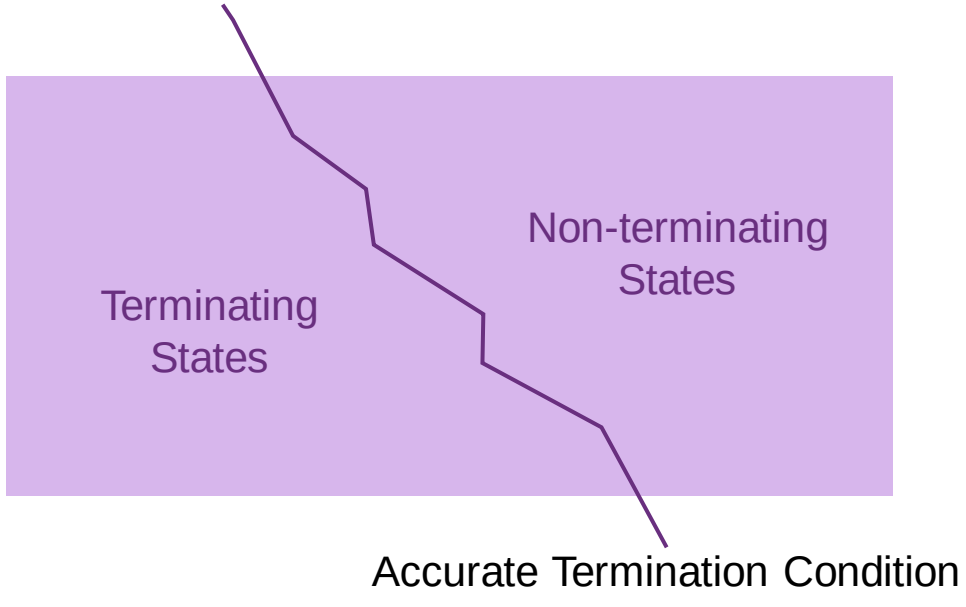
$False$,
$x \leq 0$,
$x \leq 2$,
…… are all termination conditions
$x \leq 5$ is the accurate one

A formula φ is an **accurate termination condition** of a program P if P always terminates under φ and always does not terminate under $\neg\varphi$.

State-of-the-art

- Methods focusing on conditional termination remain limited.
  - Some methods[1] calculate the precondition that making a termination argument valid;
  - Some over-approximate non-terminating states[2] or under-approximate terminating states[3].
- Existing methods do not guarantee the precision of generated conditions.

- [1] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving Conditional Termination. CAV 2008.
- [2] Marius Bozga, Radu Iosif, and Filip Konečný. Deciding Conditional Termination. TACAS 2012.
- [3] Pierre Ganty and Samir Genaim. Proving Termination Starting from the End. CAV 2013.

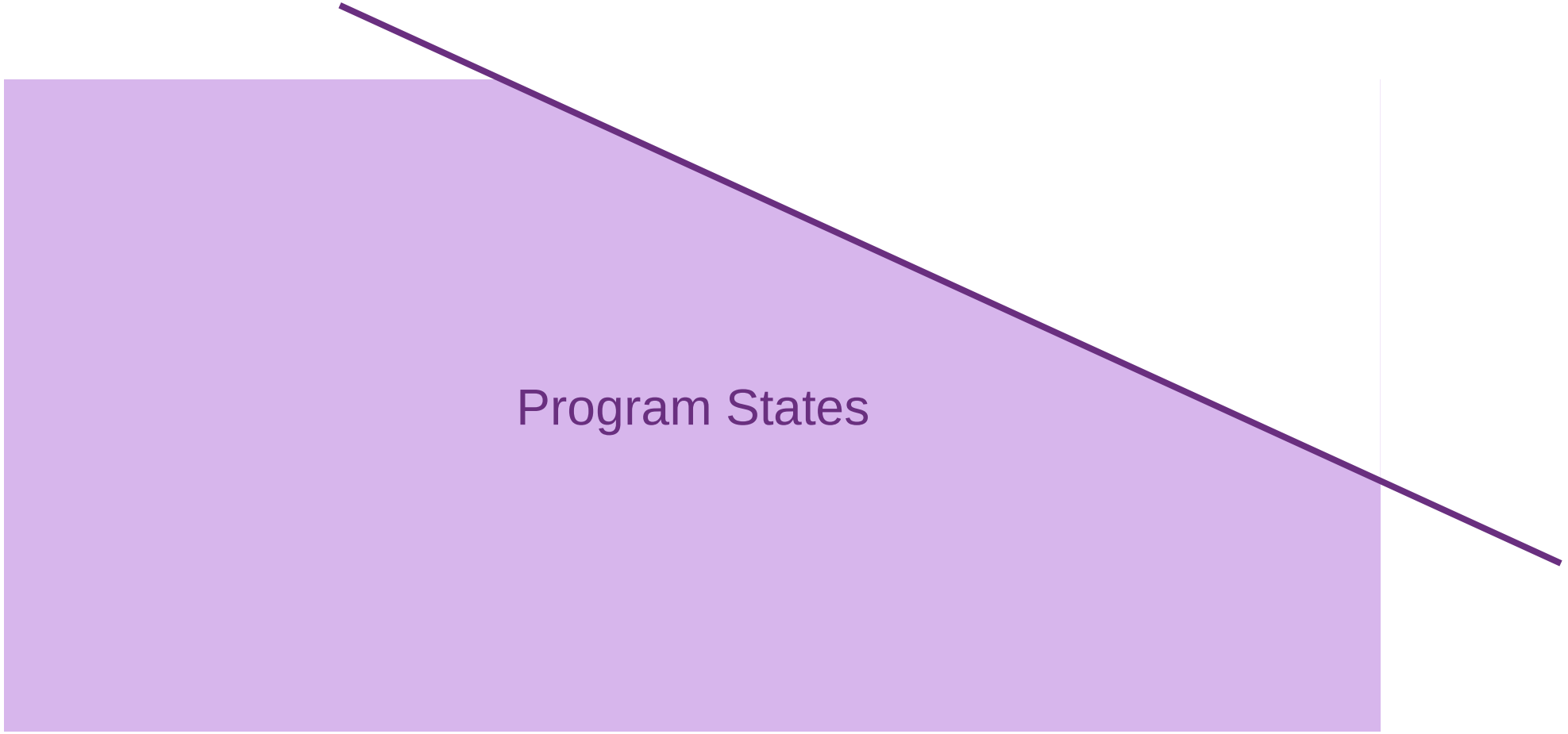


Basic Idea

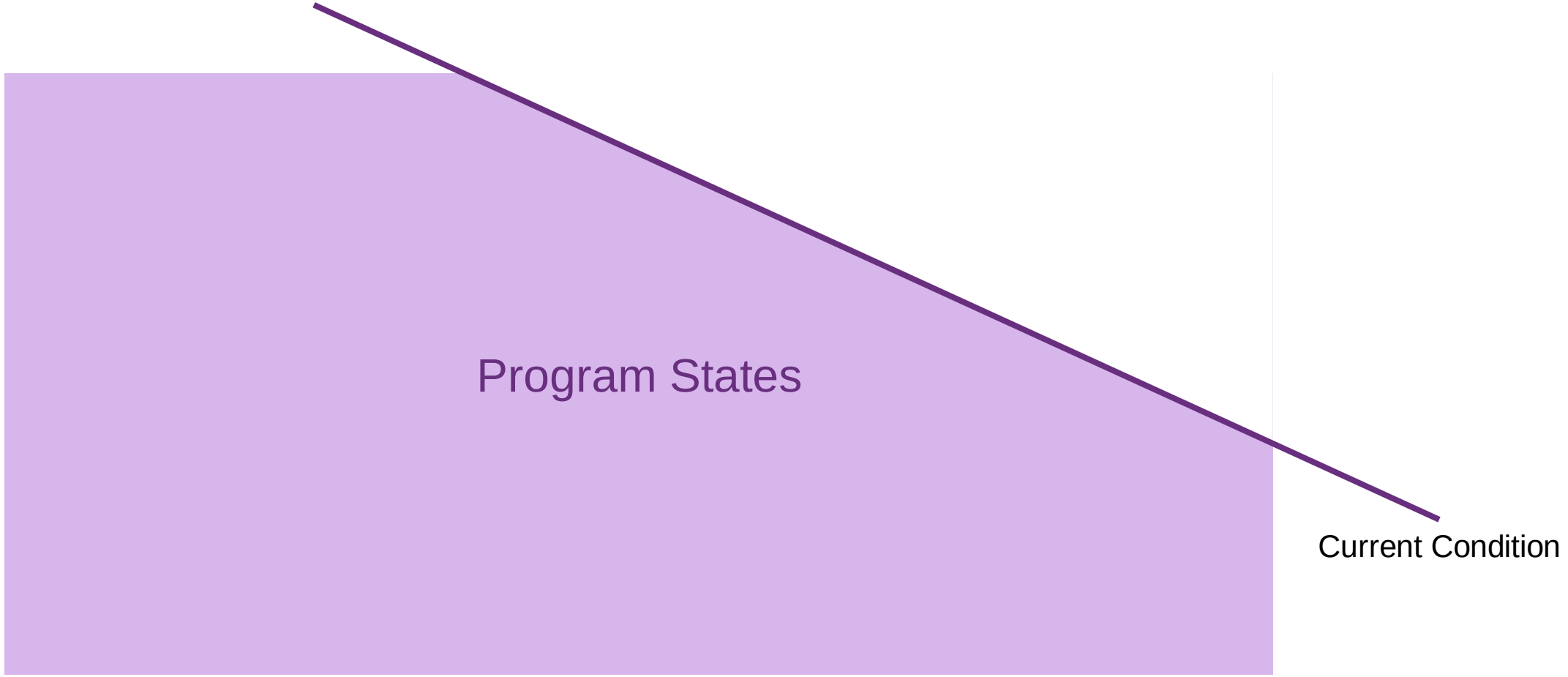
Program States

Basic Idea

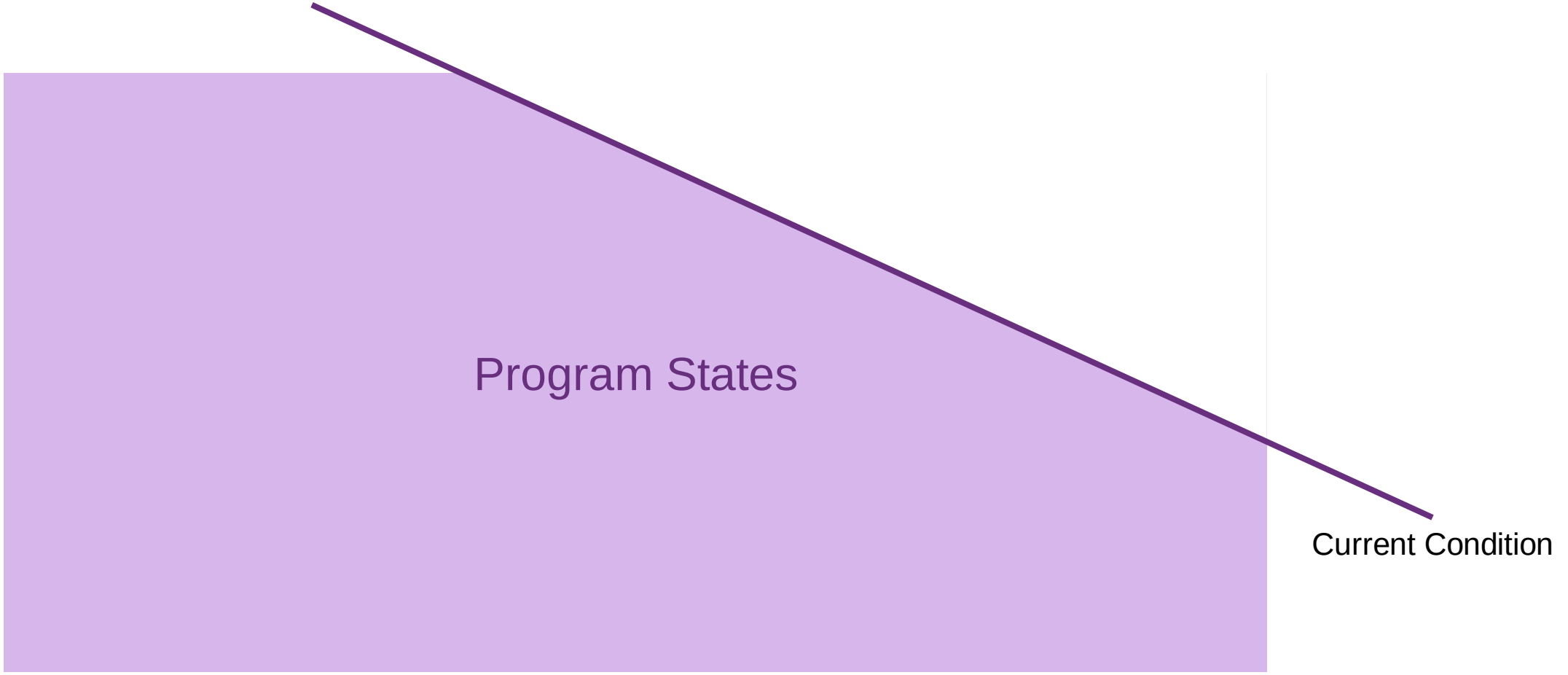
Basic Idea



Basic Idea

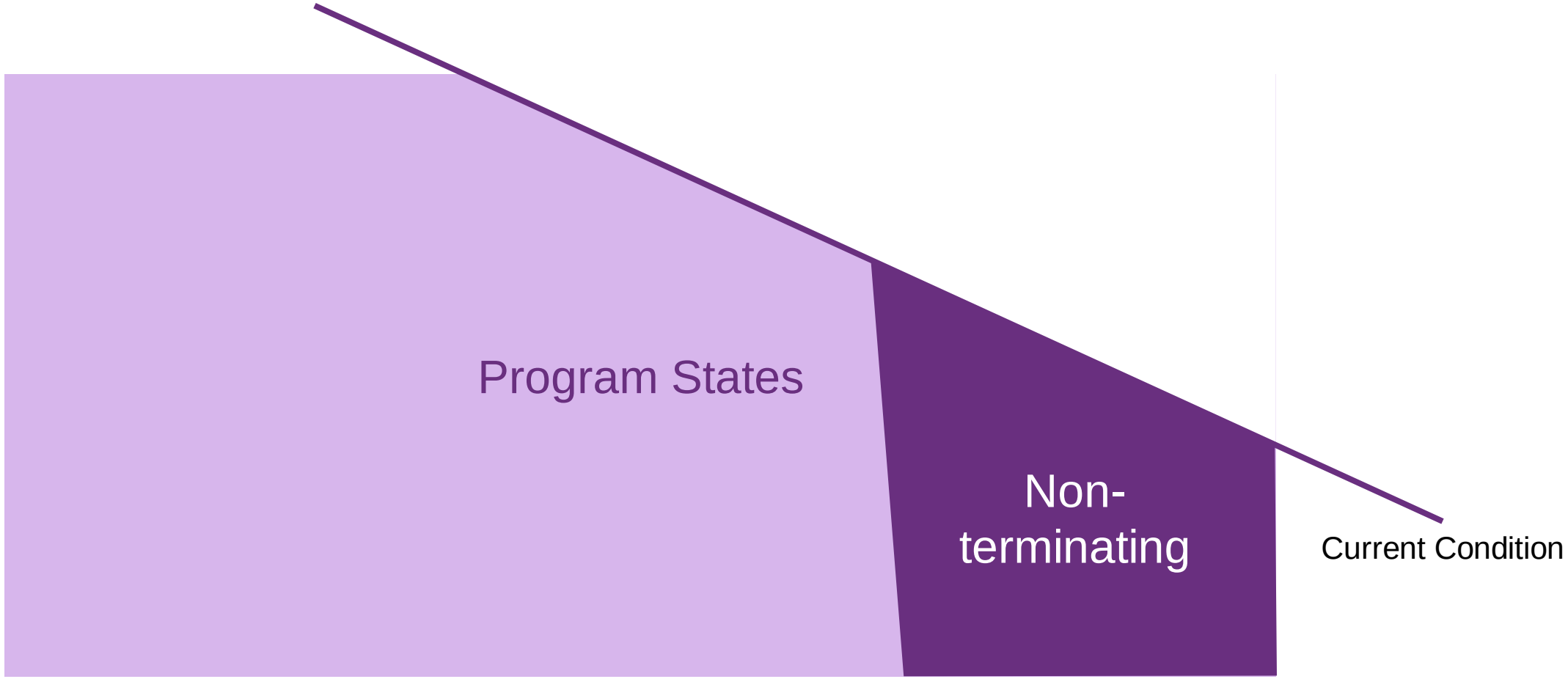


Basic Idea

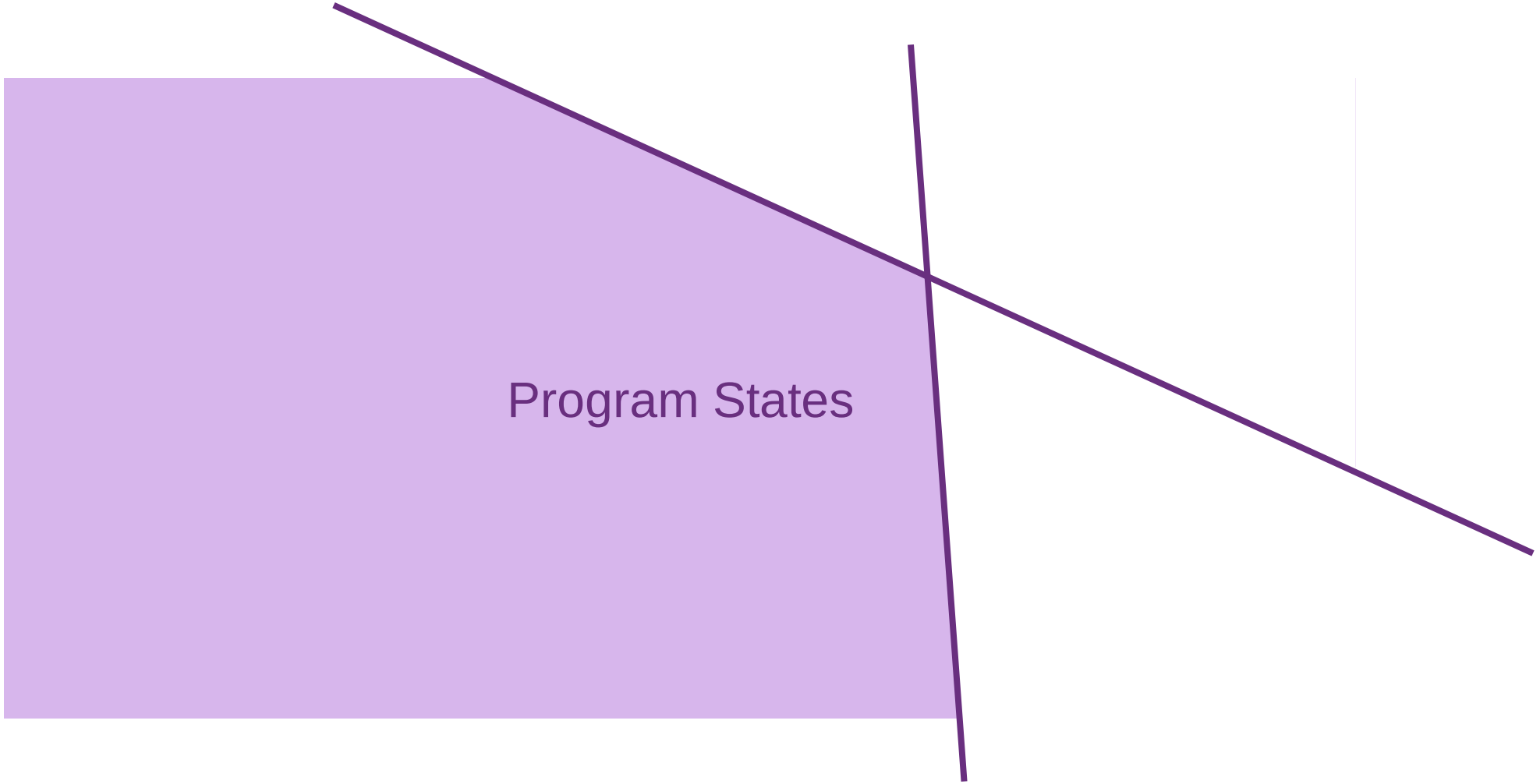


Is this program terminating?

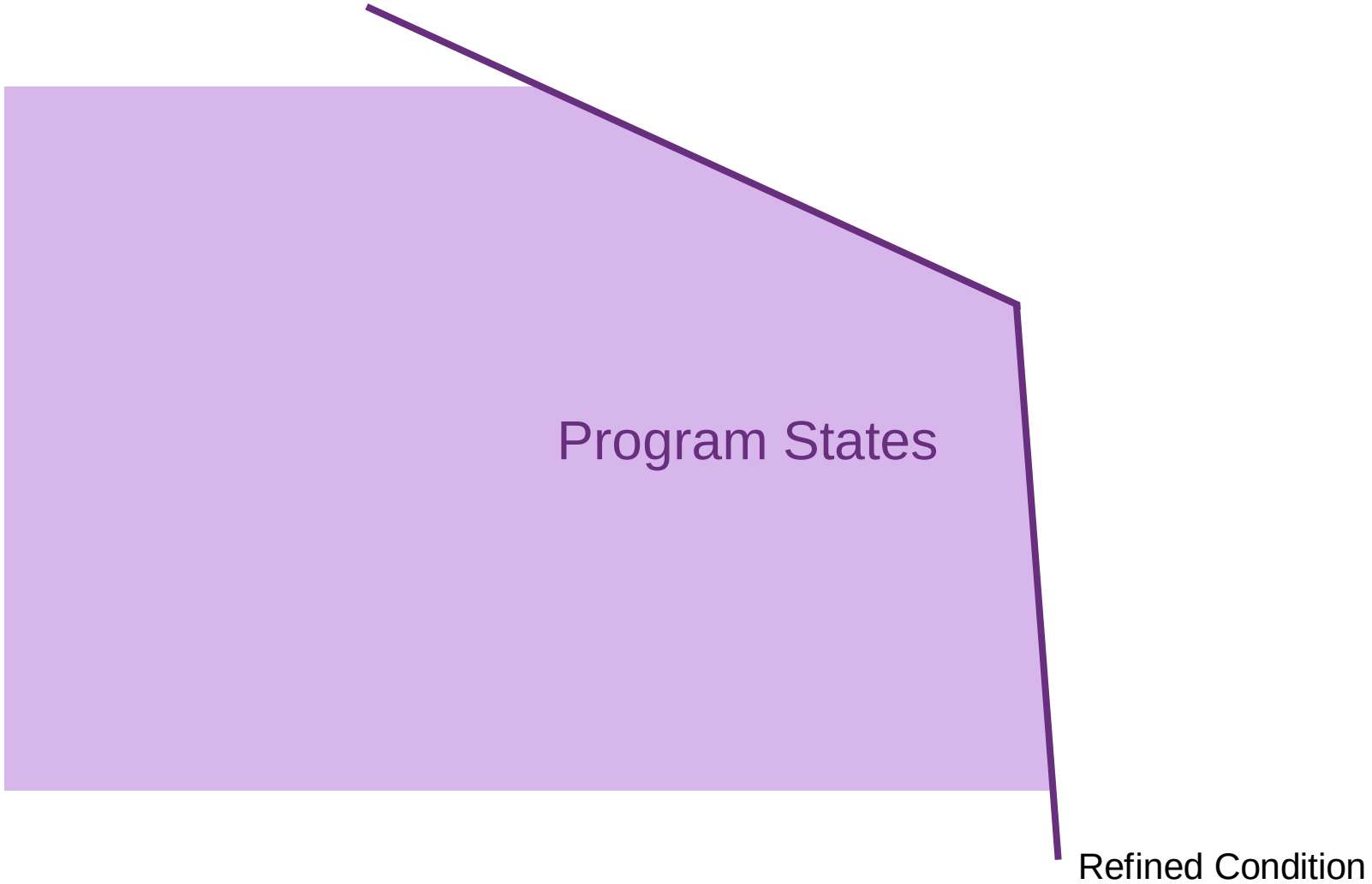
Basic Idea



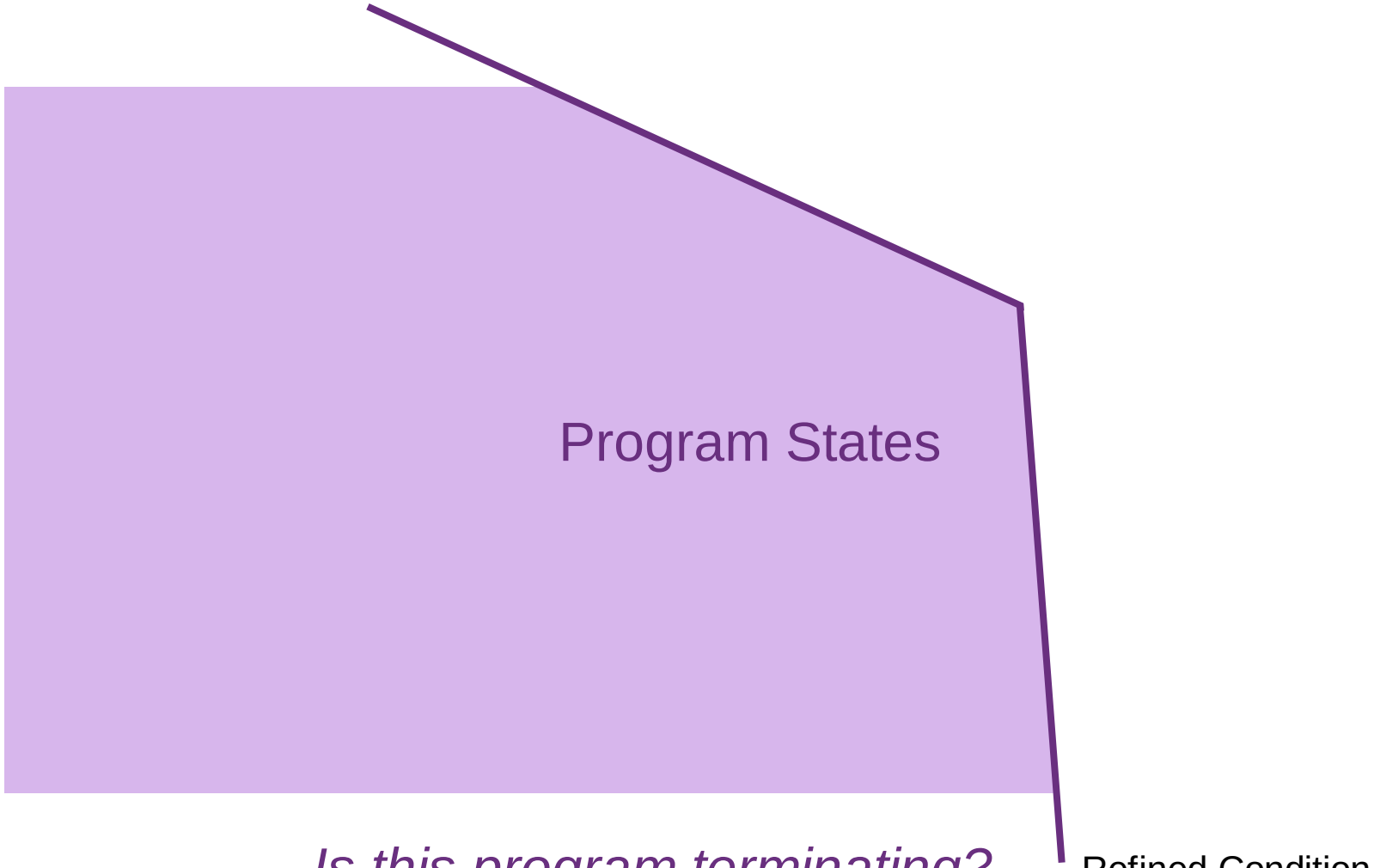
Basic Idea



Basic Idea



Basic Idea



Is this program terminating?

……

Overview

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Overview

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- If $t < -5 \vee t > 5$:
  - the absolute value of t increases $\rightarrow$ diverge

Overview

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- If $t < -5 \vee t > 5$:
  - the absolute value of t increases $\rightarrow$ diverge
- If $-5 \leq t \leq 5$:
  - t is set to $0 \rightarrow$ terminate

Overview

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- If $t < -5 \vee t > 5$:
  - the absolute value of t increases $\rightarrow$ diverge
- If $-5 \leq t \leq 5$:
  - t is set to $0 \rightarrow$ terminate

Accurate termination condition: $-5 \leq t \leq 5$

Overview

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Overview

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

a conjecture of the termination condition φ →

Overview

```
while (t != 0) {
    if (t < -5)
        t = t – 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

a conjecture of the termination condition φ

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t – 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Overview

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Overview

- Start by guessing $\varphi_0 := true$. Termination check fails.

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Overview

- Start by guessing $\varphi_0 := true$. Termination check fails.
- Get non-terminating states as a recurrent set $R_1 : t < -5$.

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Overview

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Start by guessing $\varphi_0 := true$. Termination check fails.
- Get non-terminating states as a recurrent set $R_1 : t < -5$.
- Exclude those non-terminating states by updating φ to $\varphi_1 := \varphi_0 \wedge \neg R_1 \Leftrightarrow t \geq -5$.

Overview

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Start by guessing $\varphi_0 := true$. Termination check fails.
- Get non-terminating states as a recurrent set $R_1 : t < -5$.
- Exclude those non-terminating states by updating φ to $\varphi_1 := \varphi_0 \wedge \neg R_1 \Leftrightarrow t \geq -5$.
- Instrument the loop and try to prove termination.

Overview

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Start by guessing $\varphi_0 := true$. Termination check fails.
- Get non-terminating states as a recurrent set $R_1 : t < -5$.
- Exclude those non-terminating states by updating φ to $\varphi_1 := \varphi_0 \wedge \neg R_1 \Leftrightarrow t \geq -5$.
- Instrument the loop and try to prove termination.
- Get another recurrent set $R_2 : t > 5$.

Overview

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

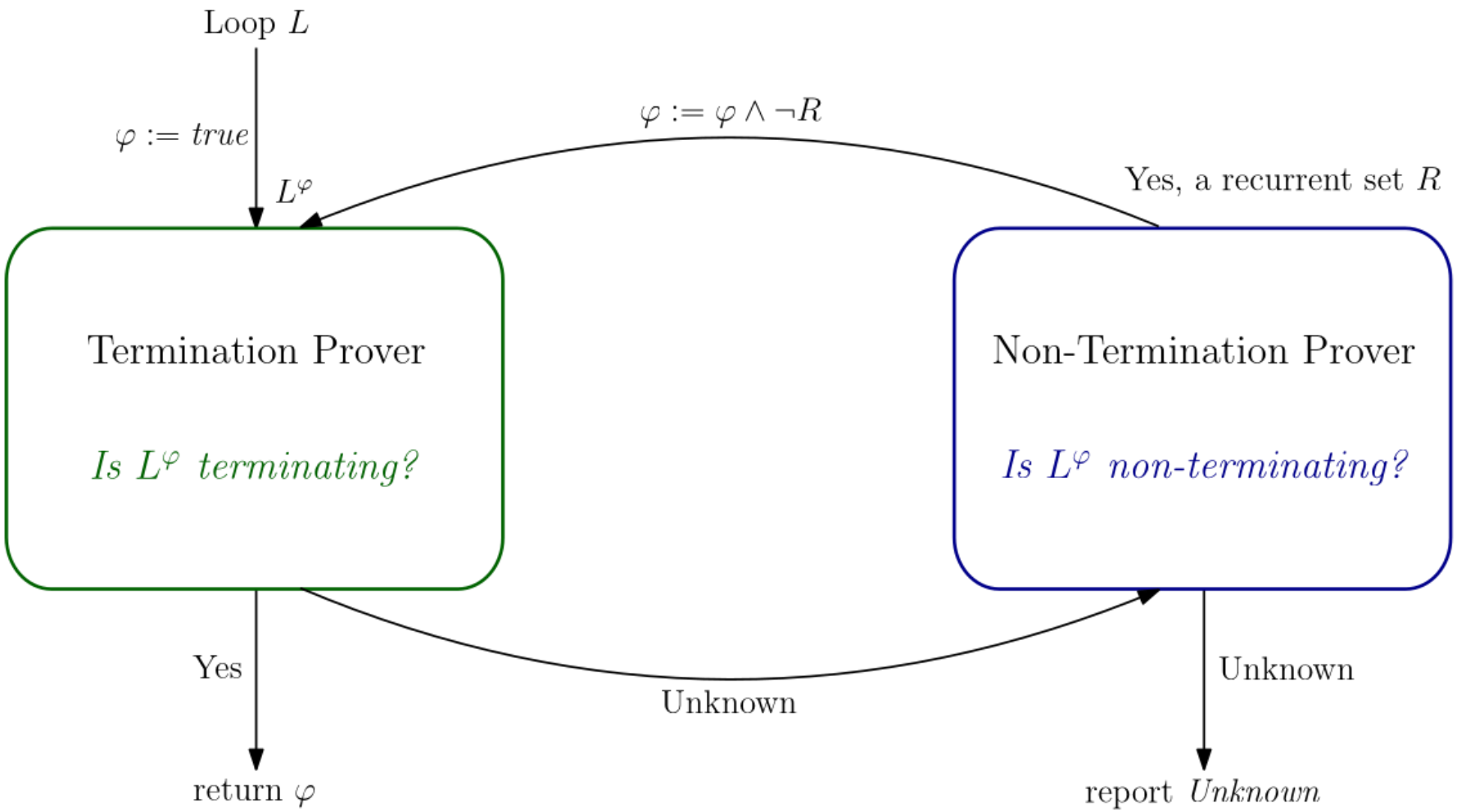
- Start by guessing $\varphi_0 := true$. Termination check fails.
- Get non-terminating states as a recurrent set $R_1 : t < -5$.
- Exclude those non-terminating states by updating φ to $\varphi_1 \coloneqq \varphi_0 \wedge \neg R_1 \Leftrightarrow t \geq -5$.
- Instrument the loop and try to prove termination.
- Get another recurrent set $R_2 : t > 5$.
- Update φ to $\varphi_2 \coloneqq \varphi_1 \wedge \neg R_2 \Leftrightarrow -5 \leq t \leq 5$, which passes the termination proof.

Overview

```
assume φ;
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Start by guessing $\varphi_0 := true$. Termination check fails.
- Get non-terminating states as a recurrent set $R_1 : t < -5$.
- Exclude those non-terminating states by updating φ to $\varphi_1 := \varphi_0 \wedge \neg R_1 \Leftrightarrow t \geq -5$.
- Instrument the loop and try to prove termination.
- Get another recurrent set $R_2 : t > 5$.
- Update φ to $\varphi_2 := \varphi_1 \wedge \neg R_2 \Leftrightarrow -5 \leq t \leq 5$, which passes the termination proof.
- Conclude that φ_2 is the accurate termination condition of the program.

Overview: Framework



Challenges

- Non-terminating states are identified by leveraging existing non-termination provers to synthesize recurrent sets.

Challenges

- Non-terminating states are identified by leveraging existing non-termination provers to synthesize recurrent sets.
- A single witness suffices to establish non-termination, so provers often return a small recurrent set.

Challenges

- Non-terminating states are identified by leveraging existing non-termination provers to synthesize recurrent sets.
- A single witness suffices to establish non-termination, so provers often return a small recurrent set.
- However, to infer the accurate termination condition, all non-terminating states must be excluded.
  - An insufficient recurrent set can lead to an increased number of iterations and thus reduce overall efficiency.

Recurrent Set Generalization

```
while (t != 0) {
    if (t < -5)
        t = t – 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Each loop is formalized using transition formulas:
  - Loop guard $\mathcal{G}(t)$: $t \neq 0$
  - Loop body $\mathcal{L}(t, t')$: $(t < -5 \rightarrow t' = 1 - t) \wedge$
    $(t > 5 \rightarrow t' = -t - 1) \wedge$
    $(-5 \leq t \leq 5 \rightarrow t' = 0)$

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Each loop is formalized using transition formulas:
  - Loop guard $\mathcal{G}(t)$: $t \neq 0$
  - Loop body $\mathcal{L}(t, t')$: $(t < -5 \rightarrow t' = 1 - t) \wedge$
    $(t > 5 \rightarrow t' = -t - 1) \wedge$
    $(-5 \leq t \leq 5 \rightarrow t' = 0)$
- Recall that a recurrent set R should satisfy:
  - $\exists t. R(t)$
  - $\forall t. R(t) \rightarrow \mathcal{G}(t)$
  - $\forall t, t'. R(t) \wedge \mathcal{L}(t, t') \rightarrow R(t')$

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Each loop is formalized using transition formulas:
  - Loop guard $\mathcal{G}(t)$: $t \neq 0$
  - Loop body $\mathcal{L}(t, t')$: $(t < -5 \rightarrow t' = 1 - t) \wedge$
    $(t > 5 \rightarrow t' = -t - 1) \wedge$
    $(-5 \leq t \leq 5 \rightarrow t' = 0)$
- Recall that a recurrent set R should satisfy:
  - $\exists t. R(t)$
  - $\forall t. R(t) \rightarrow \mathcal{G}(t)$
  - $\forall t, t'. R(t) \wedge \mathcal{L}(t, t') \rightarrow R(t')$

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Each loop is formalized using transition formulas:
  - Loop guard $\mathcal{G}(t)$: $t \neq 0$
  - Loop body $\mathcal{L}(t, t')$: $(t < -5 \rightarrow t' = 1 - t) \wedge$
    $(t > 5 \rightarrow t' = -t - 1) \wedge$
    $(-5 \leq t \leq 5 \rightarrow t' = 0)$
- Recall that a recurrent set R should satisfy:
  - $\exists t. R(t)$
  - $\forall t. R(t) \rightarrow \mathcal{G}(t)$
  - $\forall t, t'. R(t) \wedge \mathcal{L}(t, t') \rightarrow R(t')$

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

$\forall t, t'. \Psi(t, t')$[1] is valid

[1] $\Psi(t, t'): (R(t) \rightarrow \mathcal{G}(t)) \wedge (R(t) \wedge \mathcal{L}(t, t') \rightarrow R(t'))$

Recurrent Set Generalization

```
while (t != 0) {
    if (t < -5)
        t = t - 1;
        t = t * (-1);
    else if (t > 5)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Each loop is formalized using transition formulas:
  - Loop guard $\mathcal{G}(t)$: $t \neq 0$
  - Loop body $\mathcal{L}(t, t')$: $(t < -5 \rightarrow t' = 1 - t) \wedge$
    $(t > 5 \rightarrow t' = -t - 1) \wedge$
    $(-5 \leq t \leq 5 \rightarrow t' = 0)$
- Recall that a recurrent set R should satisfy:
  - $\exists t. R(t)$
  - $\forall t. R(t) \rightarrow \mathcal{G}(t)$
  - $\forall t, t'. R(t) \wedge \mathcal{L}(t, t') \rightarrow R(t')$

$\forall t, t'. \Psi(t, t')$[1] is valid

⇕

$\exists t, t'. \neg\Psi(t, t')$ is unsatisfiable

[1] $\Psi(t, t'): (R(t) \rightarrow \mathcal{G}(t)) \wedge (R(t) \wedge \mathcal{L}(t, t') \rightarrow R(t'))$

Recurrent Set Generalization

- Conduct two generalization techniques based on the minimal UNSAT core.

```
while (t != 0) {
    if (t < -w)
        t = t – 1;
        t = t * (-1);
    else if (t > w)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Conduct two generalization techniques based on the minimal UNSAT core.
- Obtain recurrent set R_1: $t < -10 \wedge t < -8$.

```
while (t != 0) {
    if (t < -w)
        t = t - 1;
        t = t * (-1);
    else if (t > w)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Conduct two generalization techniques based on the minimal UNSAT core.
- Obtain recurrent set R_1: $t < -10 \wedge t < -8$.
- **Predicate elimination**: eliminate the redundant atomic predicate $t < -10$.

```
while (t != 0) {
    if (t < -w)
        t = t - 1;
        t = t * (-1);
    else if (t > w)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

- Conduct two generalization techniques based on the minimal UNSAT core.
- Obtain recurrent set R_1: $t < -10 \wedge t < -8$.
- **Predicate elimination**: eliminate the redundant atomic predicate $t < -10$.
- Typically, an atomic predicate may be non-removable, but its range can be generalized.

```
while (t != 0) {
    if (t < -w)
        t = t - 1;
        t = t * (-1);
    else if (t > w)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

Recurrent Set Generalization

```
while (t != 0) {
    if (t < -w)
        t = t - 1;
        t = t * (-1);
    else if (t > w)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Conduct two generalization techniques based on the minimal UNSAT core.
- Obtain recurrent set R_1: $t < -10 \wedge t < -8$.
- **Predicate elimination**: eliminate the redundant atomic predicate $t < -10$.
- Typically, an atomic predicate may be non-removable, but its range can be generalized.
- **Constant generalization**: generalize $t < -8$ to $t < -5$.

Recurrent Set Generalization

```
while (t != 0) {
    if (t < -w)
        t = t - 1;
        t = t * (-1);
    else if (t > w)
        t = t + 1;
        t = t * (-1);
    else
        t = 0;
}
```

- Conduct two generalization techniques based on the minimal UNSAT core.
- Obtain recurrent set $R_1: t < -10 \wedge t < -8$.
- **Predicate elimination**: eliminate the redundant atomic predicate $t < -10$.
- Typically, an atomic predicate may be non-removable, but its range can be generalized.
- **Constant generalization**: generalize $t < -8$ to $t < -5$.
- Generalized recurrent set $R_1': t < -5$.

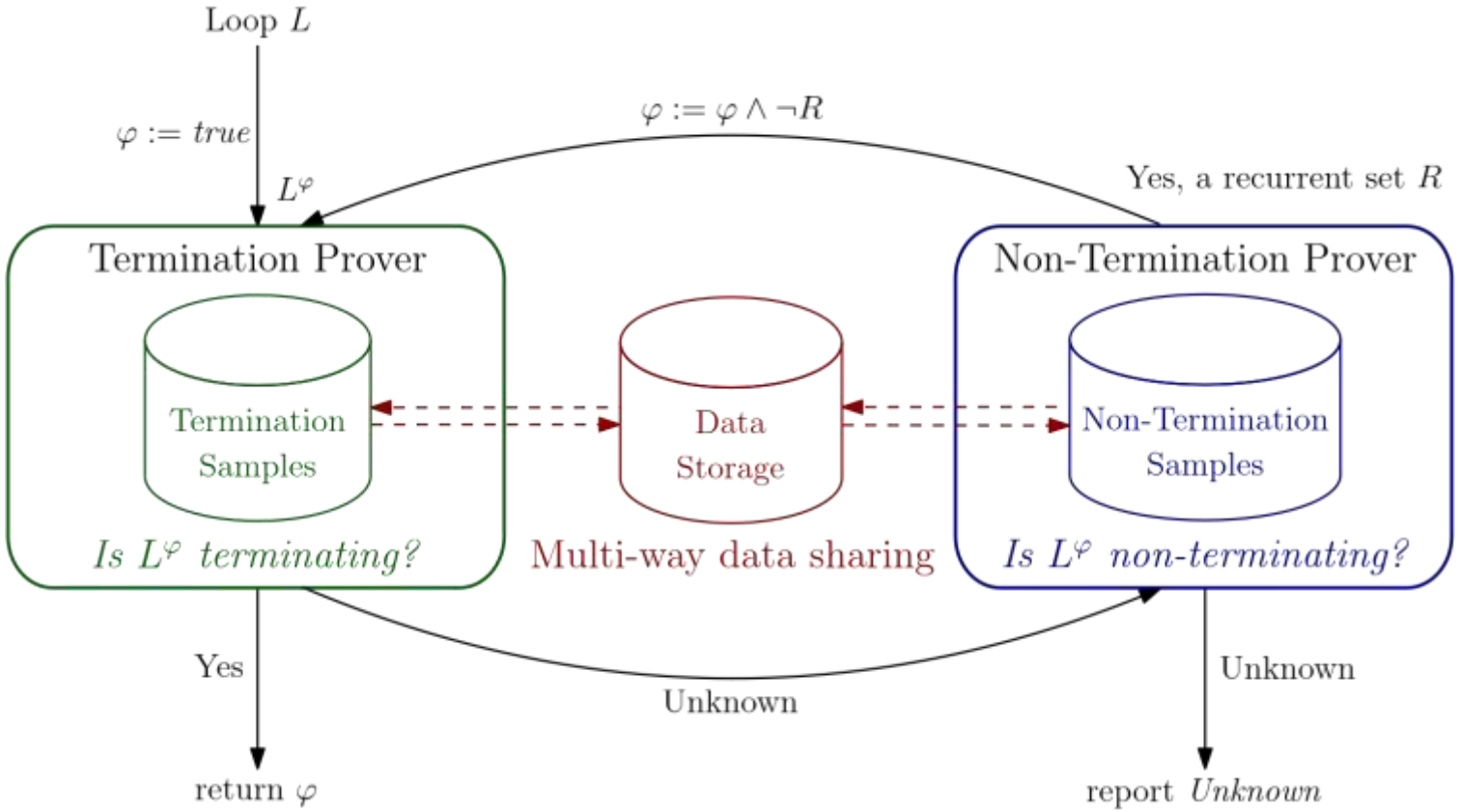
(Non-)Termination Provers

(Non-)Termination Provers

- We adopt **data-driven** provers, which use black-box method to synthesize arguments.

(Non-)Termination Provers

- We adopt **data-driven** provers, which use black-box method to synthesize arguments.
- Since each invocation of a prover targets a similar program, we design a **multi-way data sharing** mechanism, storing samples in each learning process for reuse.



Correctness

- Our method ensures soundness, accurateness, termination, and relative completeness.

Theorem 1 (Soundness)

For any loop L, if our algorithm returns a predicate φ, then φ is guaranteed to be a termination condition of L, provided that the termination prover used in the algorithm is sound.

Theorem 2 (Accurateness)

For any loop L, if our algorithm returns a predicate φ, then φ is guaranteed to be an accurate termination condition of L, provided that the termination and non-termination provers used in the algorithm are sound.

Correctness

Theorem 3 (Termination)

Given a loop L and a set of attributes $A = \{a_1, \ldots, a_n\}$, if the accurate termination condition φ^* can be expressed as a boolean combination of atomic predicates $a \leq c$ where $a \in A$ and c is any integer, then our algorithm is guaranteed to terminate, provided that the termination prover is sound, the non-termination prover is both sound and relatively complete, and the recurrent set identified in each iteration satisfies a specific assumption.

Theorem 4 (Relative Completeness)

Building upon the conditions of Theorem 3, if the termination prover is further complete, then the method is guaranteed to terminate with an accurate termination condition.

Evaluation

- We implement our tool as a prototype called CondTerm on top of data-driven provers ddlTerm[1] and RSLearn[2].
- **Benchmarks**: 170 terminating programs and 96 non-terminating programs.
- **Baseline**: Acabar[3], the latest tool with the same goal of synthesizing termination conditions as comprehensively as possible.

[1] Rongchen Xu, Jianhui Chen, and Fei He. Data-driven loop bound learning for termination analysis. ICSE 2022.
[2] Zhilei Han and Fei He. Data-Driven Recurrent Set Learning For Non-Termination Analysis. ICSE 2023.
[3] Pierre Ganty and Samir Genaim. Proving Termination Starting from the End. CAV 2013.

Evaluation

- Our tool solves more cases and generate more accurate termination conditions for both non-terminating and terminating benchmarks.

Benchmarks	NT (96 total)		T (170 total)	
Tools	CondTerm	Acabar	CondTerm	Acabar
#Solved.	**78**	51	**136**	64
#Accurate Sol.	**78**	41	**136**	56
Avg. T. on Sol.(s)	8.01	**0.81**	14.89	**3.57**

Evaluation

- The recurrent set generalization (R.S.G) and multi-way data sharing (M.D.S) techniques improve both efficiency and the number of solved cases.

Benchmarks	NT	
Enable R.S.G	Y	N
#Solved.	**78**	76(-2)
Overall Avg. T.(s)	**39.11**	54.86(40.3%)
Overall Avg.Iter.	**3.03**	3.53(16.5%)

Benchmarks	NT		T	
Enable M.D.S	Y	N	Y	N
#Solved.	**78**	78(0)	136	134(-2)
Avg. T. on Both Sol.(s)	**8.01**	8.62(7.6%)	14.03	14.44(2.9%)
Avg.Iter. On Both Sol.	**2.29**	2.31(1.3%)	1	1

Thanks!