Automatically Disproving Fair Termination of Higher-Order Functional Programs

Keiichi Watanabe, Ryosuke Sato
Takeshi Tsukada, Naoki Kobayashi

The University of Tokyo

September 20th, 2016

ICFP 2016 at Nara

Our Goal

Automated method for **disproving fair-termination** of higher-order functional programs

cf. Prove Fair-termination [Murase+ POPL16]

includes **LTL properties**

Verification of **ω-regular properties** can be reduced to that of **fair-termination** [Vardi APAL91]

Outline

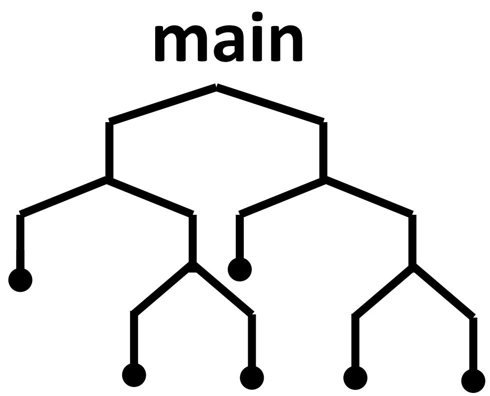
- Termination & Fair-Termination
- Importance of Fair-Termination
- Our Method
- Implementation and Experiments
- Related Work
- Conclusion

Outline

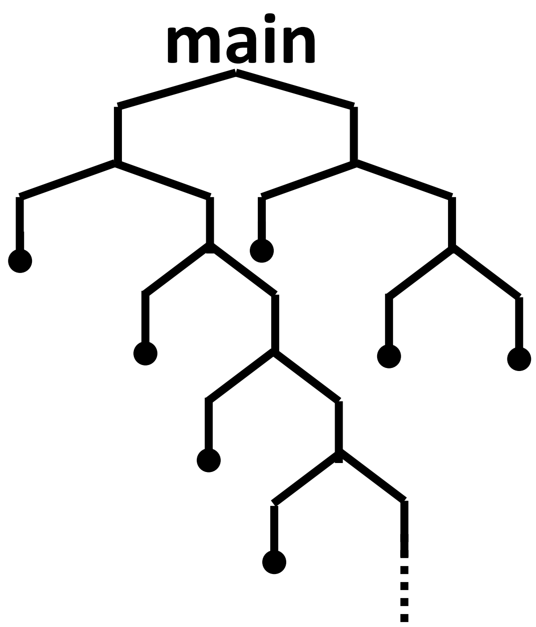
- **Termination & Fair-Termination**
- Importance of Fair-Termination
- Our Method
- Implementation and Experiments
- Related Work
- Conclusion

Plain Termination

Program P is **terminating**
$\Leftrightarrow$ Every execution eventually terminates



Terminating



Not Terminating

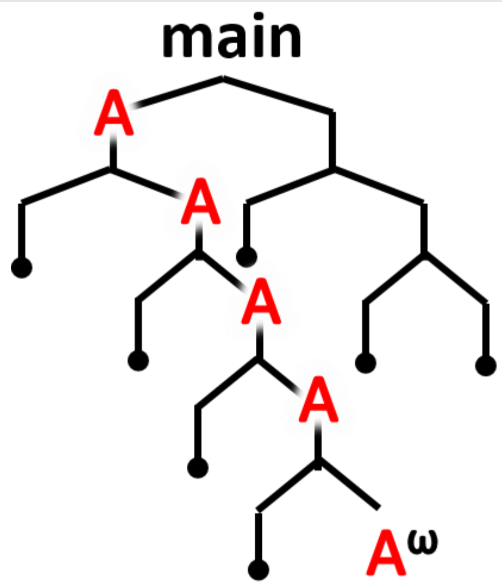
Fair-Termination

Program P is **fair-terminating**

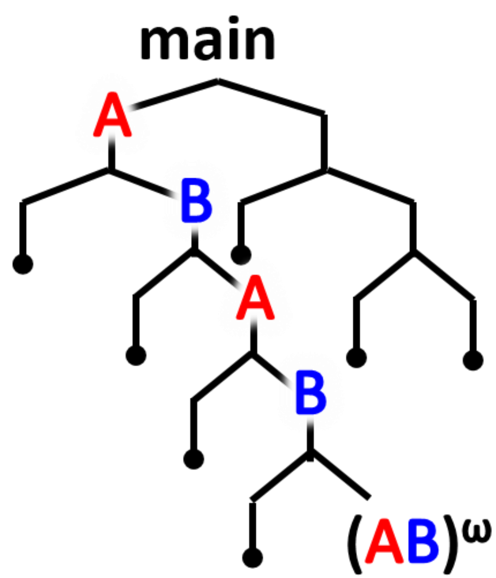
⇔ Every **fair** execution eventually terminates

An example of **fairness** in this talk:

If **A** occurs infinitely often, so does **B**



Fair-Terminating



Not Fair-Terminating

Outline

- Termination & Fair-Termination
- **Importance of Fair-Termination**
- Our Method
- Implementation and Experiments
- Related Work
- Conclusion

Termination assuming Randomness

```
let rand_int () = *int

let rec rand_pos () =
  let x = rand_int () in
  if 0 < x then
    x
  else
    rand_pos ()

let main = rand_pos ()
```

Terminating, assuming **randomness** of $*_{\text{int}}$

Termination assuming Randomness

```
let rand_int () = *int

let rec rand_pos () =
 let x = rand_int () in
 if 0 < x then
   x
 else
   rand_pos ()

let main = rand_pos ()
```

Terminating, assuming **randomness** of $*_{\text{int}}$

Q.

How to incorporate randomness with termination verification?

Termination assuming Randomness

```
let rand_int () =
 let r = *int in
 if 0 < r then
   (event B; r)
 else
   (event A; r)

let rec rand_pos () =
 let x = rand_int () in
 if 0 < x then
   x
 else
   rand_pos ()

let main = rand_pos ()
```

Insert event expressions

Termination assuming Randomness

```
let rand_int (
 let r = *int i
 if 0 < r then
   (event B;
 else
   (event A; r)

let rec rand_pos () =
 let x = rand_int () in
 if 0 < x then
   x
 else
   rand_pos ()

let main = rand_pos ()
```

If *int **never** returns a positive integer, execution is **unfair**

A → A → A → A →…

Termination assuming randomness

→ **Fair-termination**

Our Goal (Again)

Automated method for **disproving fair-termination** of higher-order functional programs

cf. Prove Fair-termination [Murase+ POPL16]

includes **LTL properties**

Verification of ***ω*-regular properties**
can be reduced to
that of **fair-termination** [Vardi APAL91]

Our Goal (Again)

Automated method for **disproving fair-termination** of higher-order functional progra…

Proving the existence of **fair infinite** executions

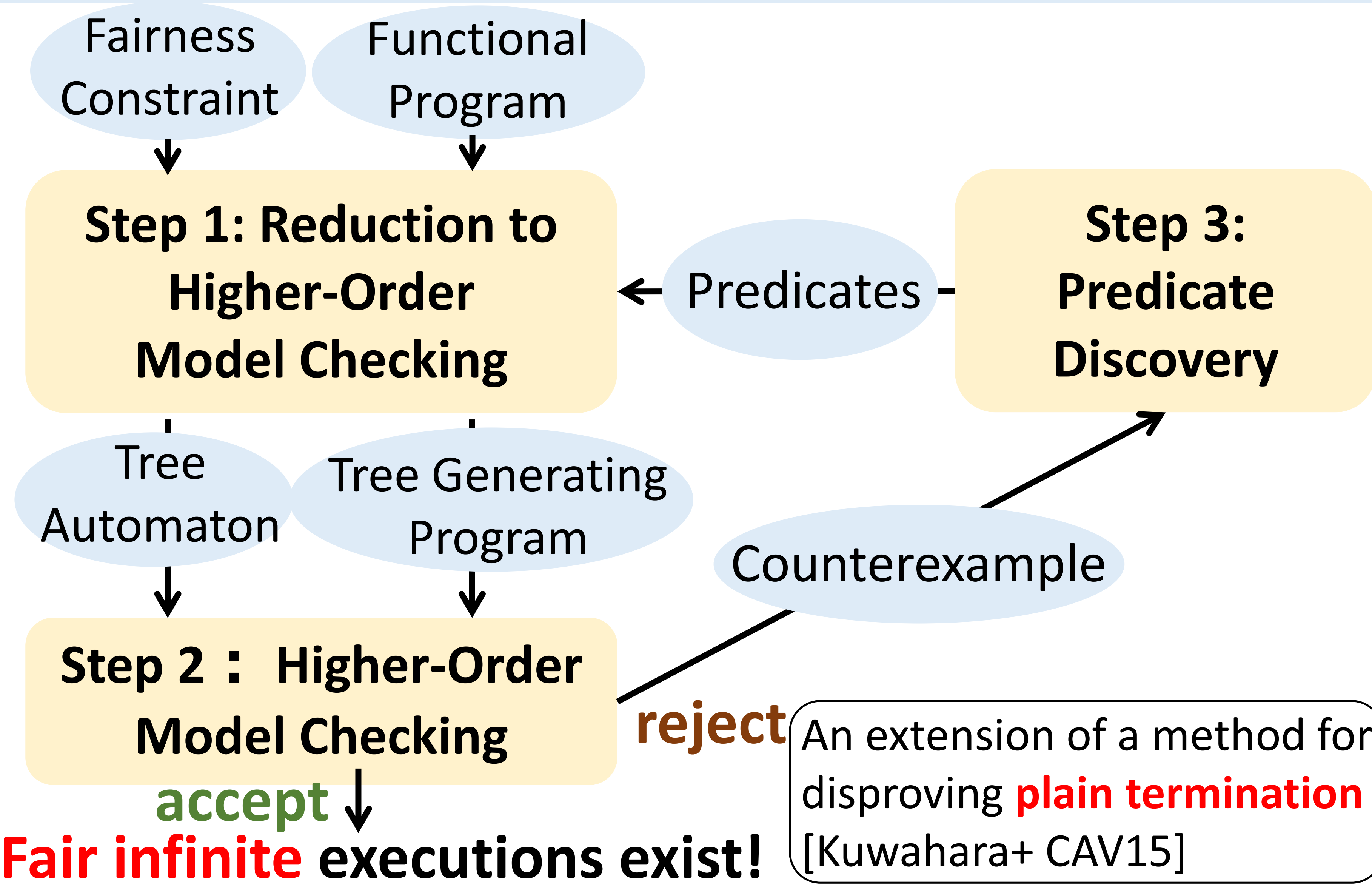
includes **LTL properties**

Verification of **ω-regular properties** can be reduced to that of **fair-termination** [Vardi APAL91]

Outline

- Termination & Fair-Termination
- Importance of Fair-Termination
- **Our Method**
  - Overview of Method
  - Step 1, Step 2, Step 3
  - Properties of Our Method
- Implementation and Experiments
- Related Work
- Conclusion

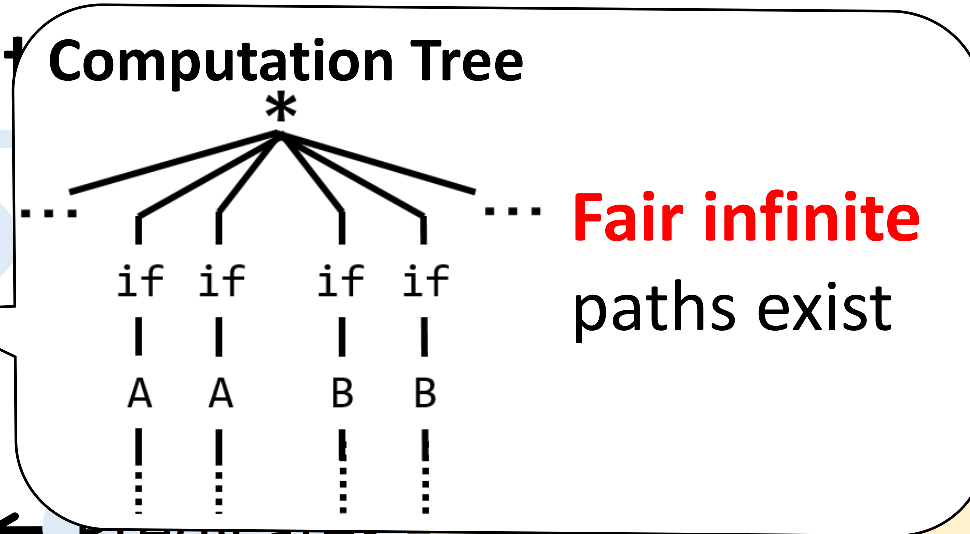
Overview of Method



Overview of Met

Fairness Constraint

Functional Program



Step 1: Reduction to Higher-Order Model Checking

Discovery

Tree Automaton

Tree Generating Program

Counterexample

Step 2 : Higher-Order Model Checking

reject

accept

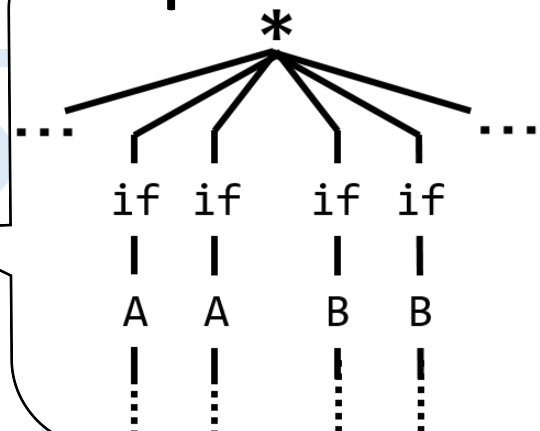
Fair infinite executions exist!

Overview of Met

Fairness Constraint

Functional Program

Computation Tree



Fair infinite paths exist

Step 1: Reduction to Higher-Order Model Checking

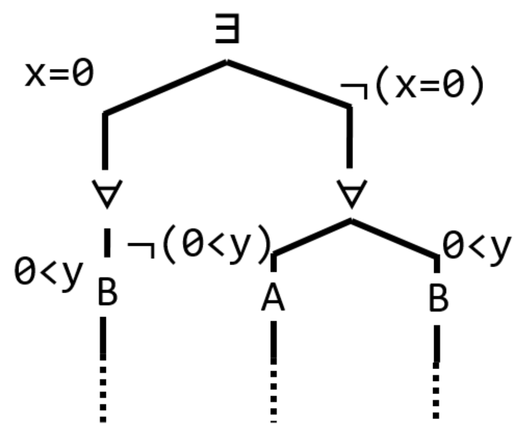
Predicates

Predicate Discovery

Tree Automaton

Tree Generat Program

Abstracted Tree



Accepted by the automaton

Step 2 : Higher-Order Model Checking

accept

Fair infinite executions exist?

Overview of Met

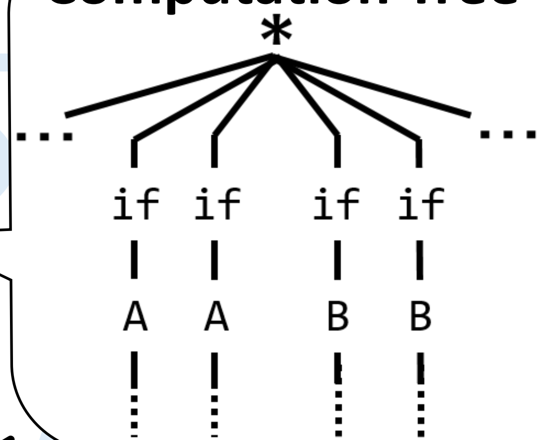
Fairness Constraint

Functional Program

Step 1: Reduction to Higher-Order Model Checking

Predicates

Computation Tree



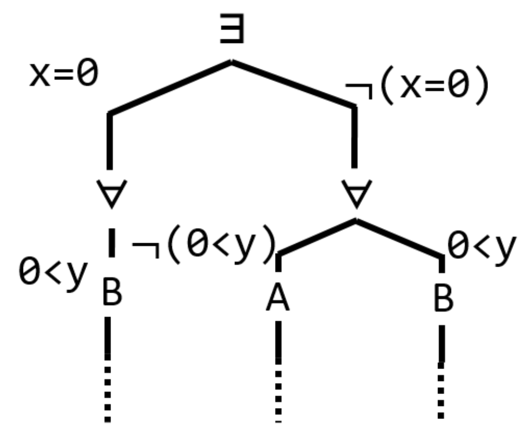
Fair infinite paths exist

Sufficient condition

Tree Automaton

Tree Generat Program

Abstracted Tree



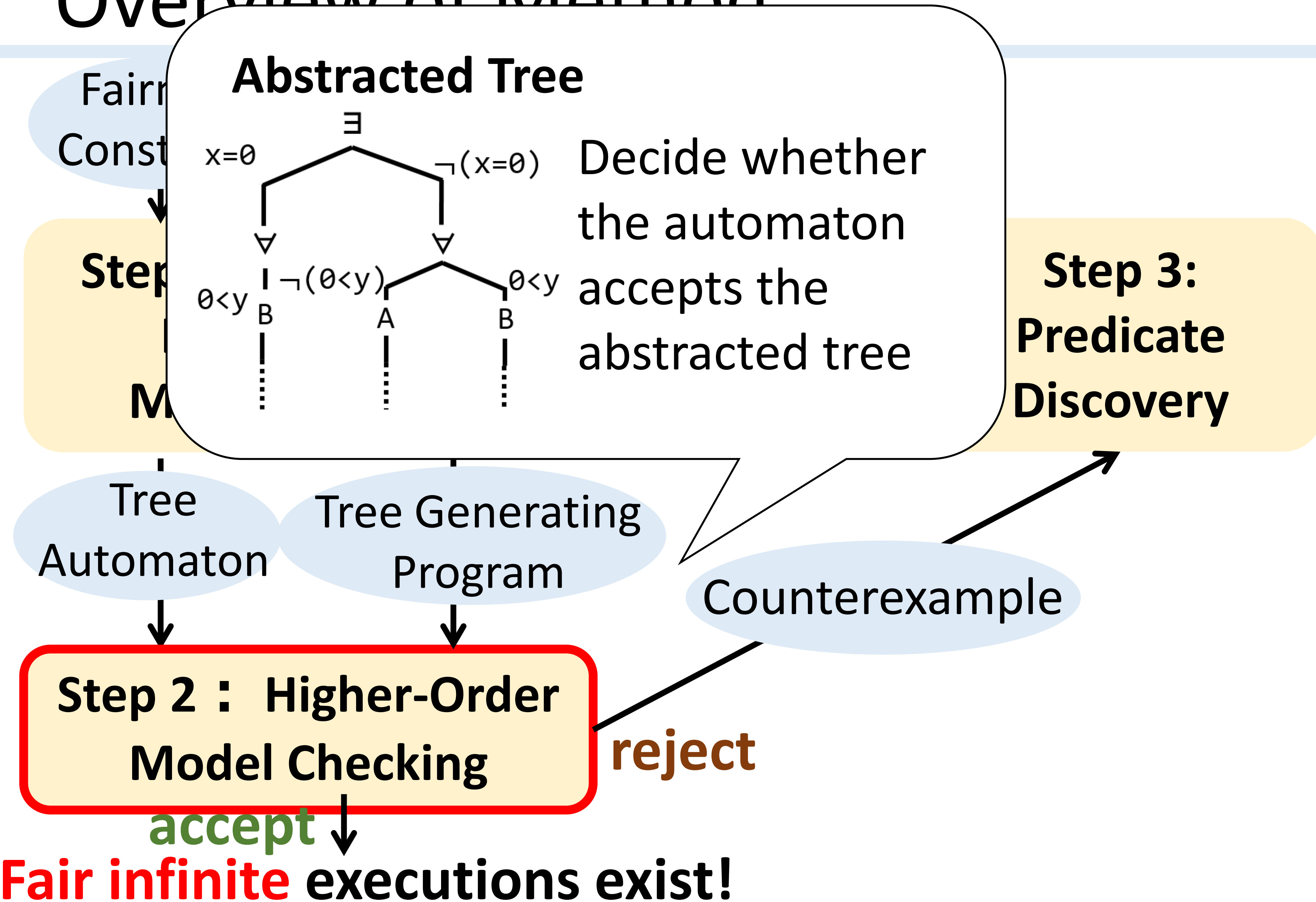
Accepted by the automaton

Step 2： Higher-Order Model Checking

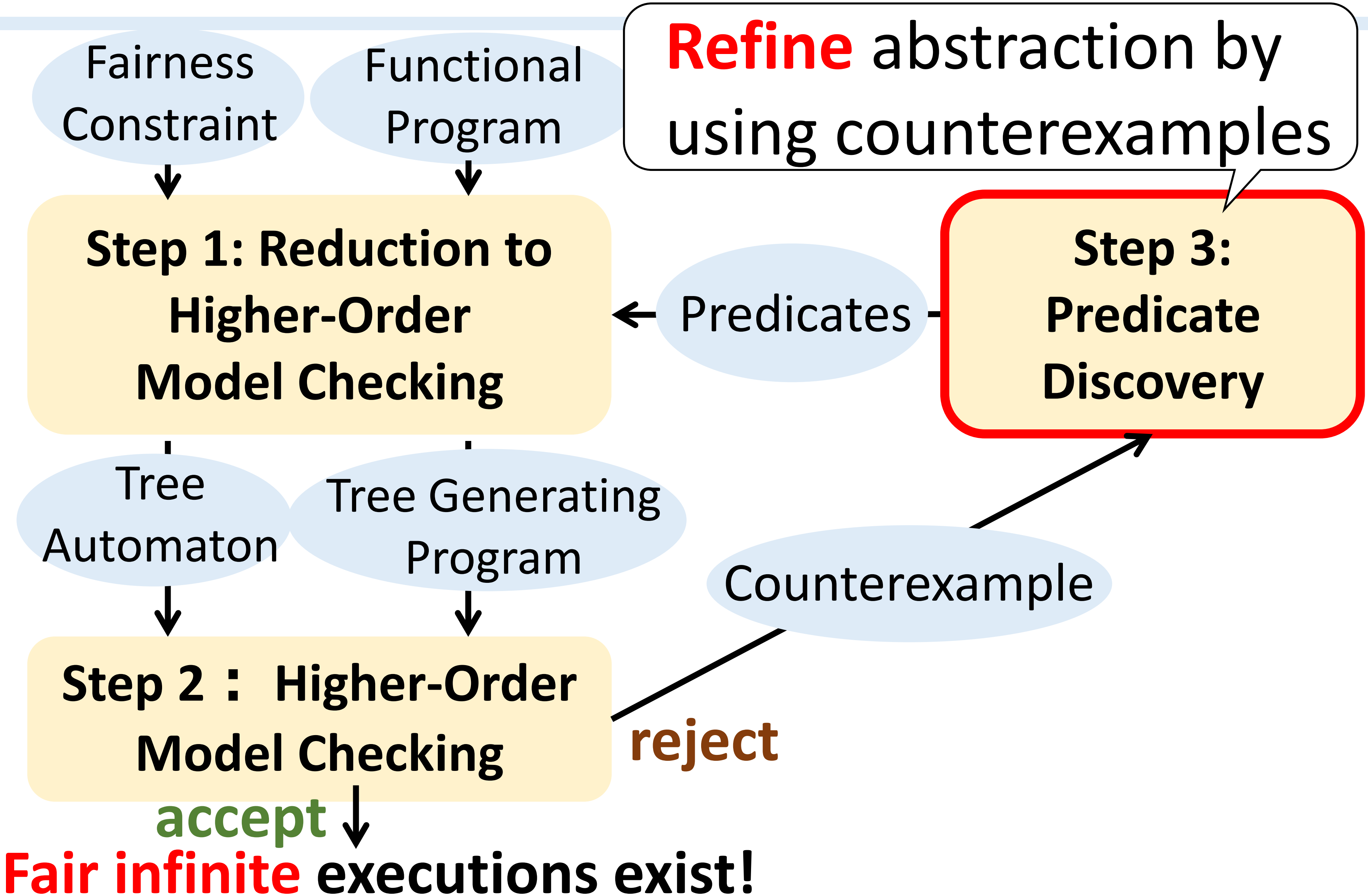
accept

Fair infinite executions exist?

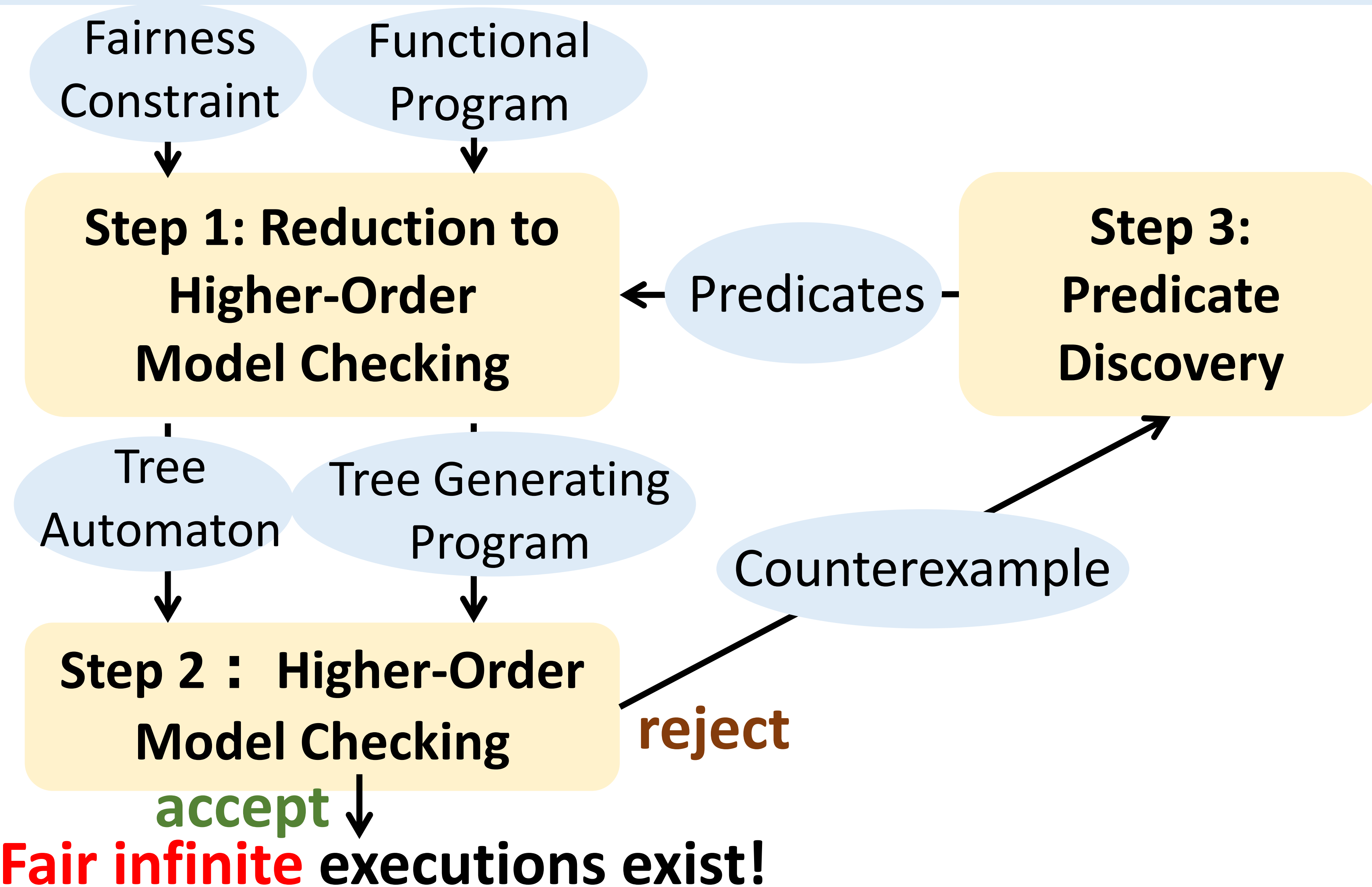
Overview of Method



Overview of Method



Overview of Method

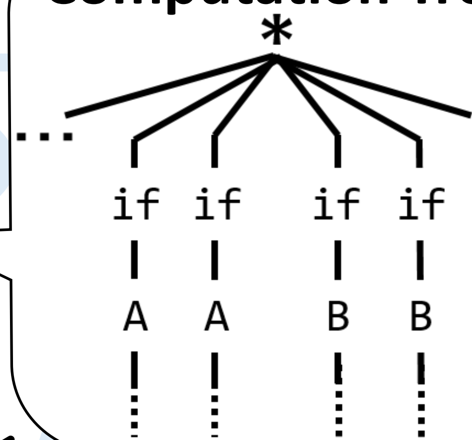


Overview of Met

Fairness Constraint

Functional Program

Computation Tree



Fair infinite paths exist

Step 1: Reduction to Higher-Order Model Checking

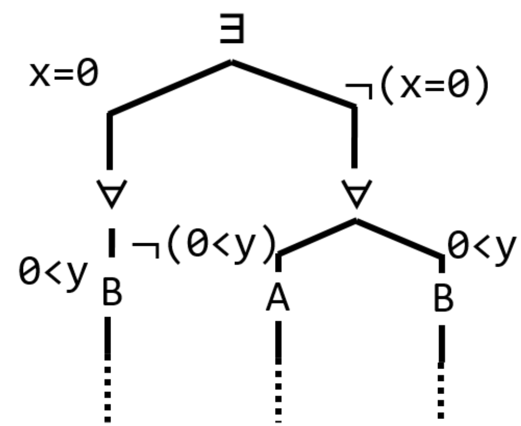
Predicates

Sufficient condition

Tree Automaton

Tree Generat Program

Abstracted Tree



Accepted by the automaton

Step 2： Higher-Order Model Checking

accept

Fair infinite executions exist!

Two Branching Nodes in Abstracted Trees

[Kuwahara+ CAV15]

∃-node

- Represents **inherent non-determinism** in programs
  - e.g. random integer, inputs
- We should check if **there exists** a fair infinite branch

∀-node

- Represents **non-determinism** introduced **by abstraction**
- We should check if **every branch** is fair and infinite

Two Branching Nodes in Abstracted Trees

[Kuwahara+ CAV15]

P

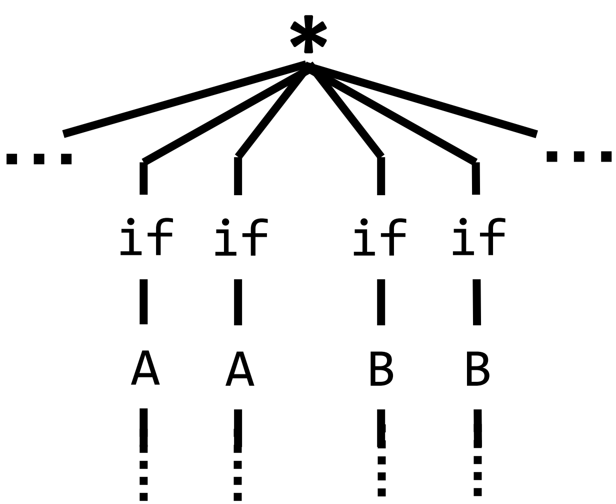
```
let f x =
 let y = x+1 in
 if 0 < y then
  event B; g y
 else
  event A; g y
in f *int
```

Abstract by $x = 0, 0 < y$

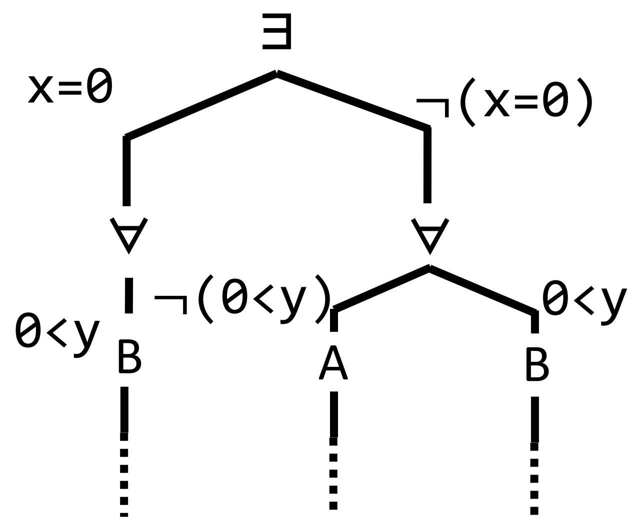
D

```
let f bx=0 =
 if bx=0 then
  ∀(B(g true))
 else
  ∀(B(g true), A(g false))
in ∃(f true, f false)
```

Computation tree of P



Tree(D)



∃-node: Inherent Non-Determinism

P

```
let f x =
  let y = x+1 in
  if 0 < y then
   event B; g y
  else
   event A; g y
in f *int
```

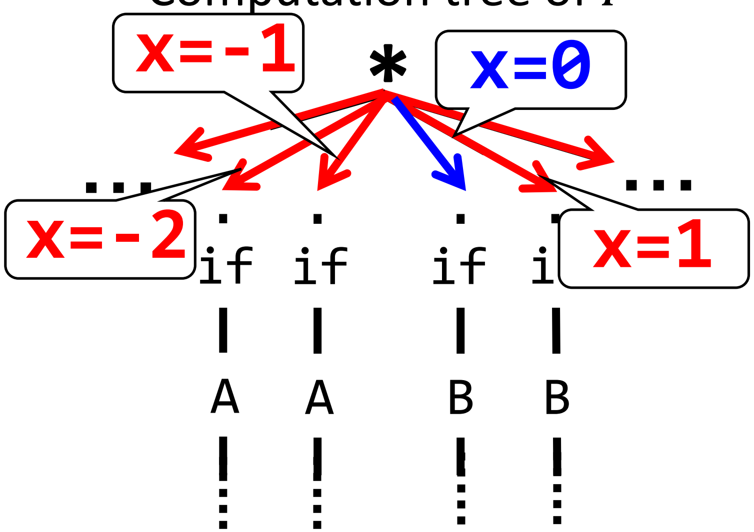
Abstract by $x = 0, 0 < y$

D

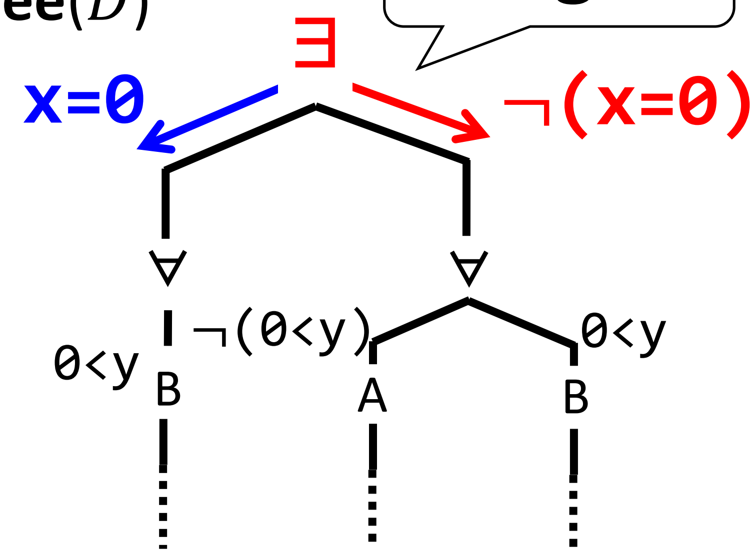
```
let f bx=0 =
  if bx=0 then
   ∀(B(g true))
  else
   ∀(B(g true), A(g false))
in ∃(f true, f false)
```

merged

Computation tree of P



Tree(D)



∃-node: Inherent Non-Determinism

```
let f x =
 let y = x+1 in
 if 0 < y then
  event B; g y
 else
  event A; g y
in f *int
```

P

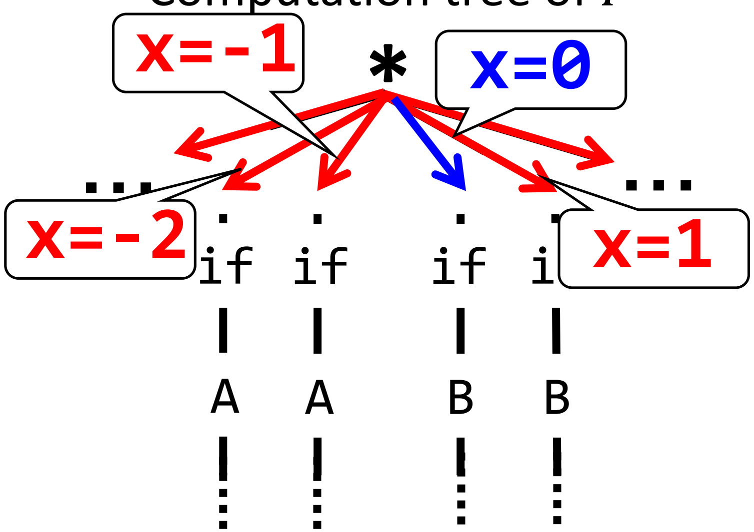
Abstract by $x = 0, 0 < y$

D

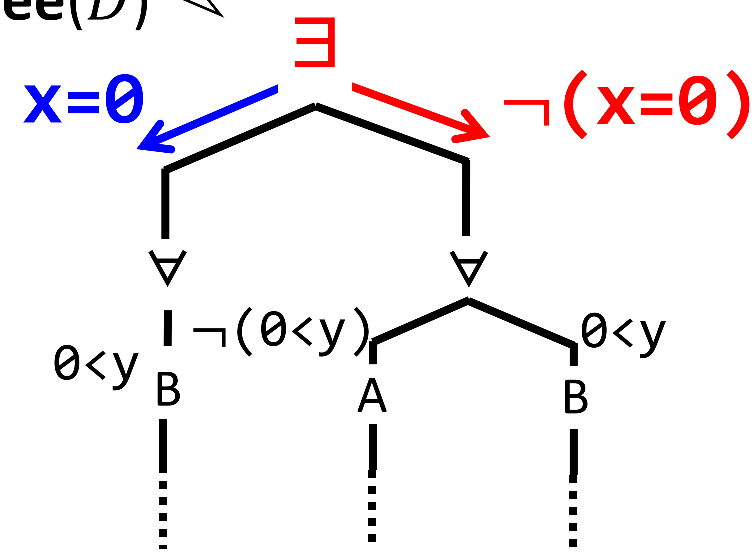
```
let f bx=0 =
 if bx=0 then
  ∀(B(g true))
```

Check if **either branch** is fair and infinite

Computation tree of *P*



Tree(D)



∀-node: Non-Determinism introduced by Abstraction

```
let f x =
  let y = x+1 in
  if 0 < y then
   event B; g y
  else
   event A; g y
in f *int
```

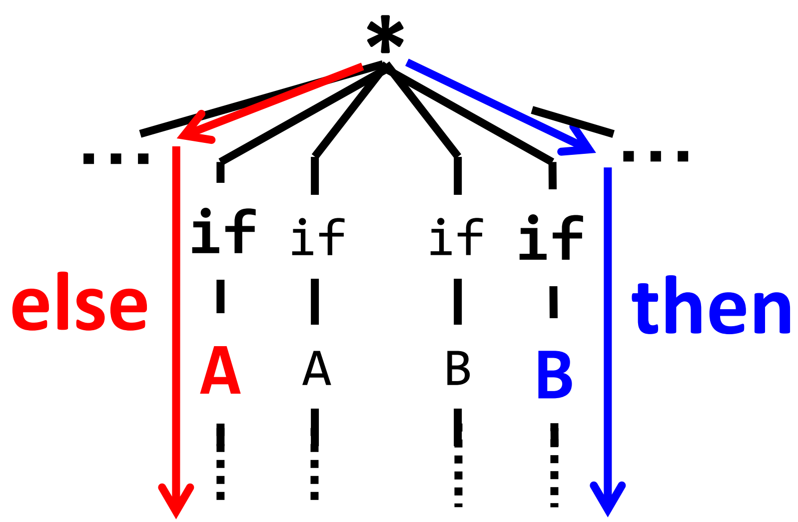
P

Abstract by $x = 0, 0 < y$

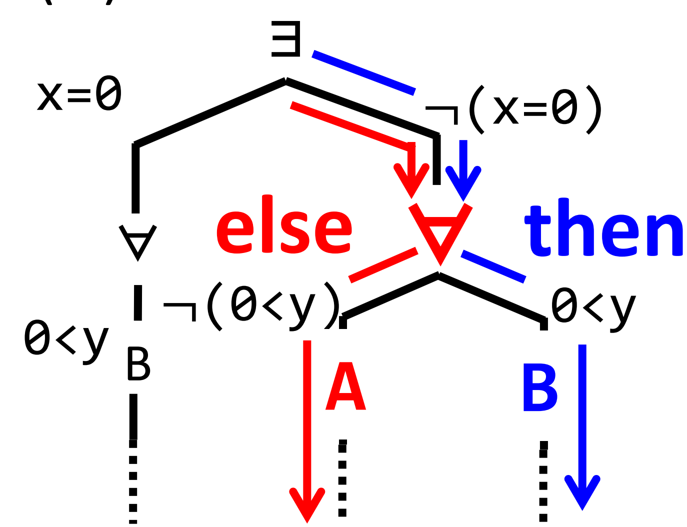
```
let f bx=0 =
  if bx=0 then
   ∀(B(g true))
  else
   ∀(B(g true), A(g false))
in ∃(f true, f false)
```

D

Computation tree of P



Tree(D)



∀-node: Non-Determinism introduced by Abstraction

P

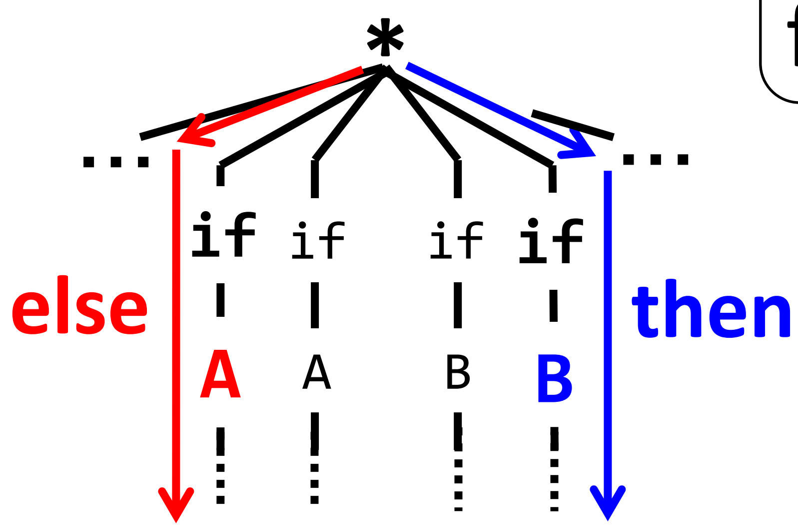
```
let f x =
  let y = x+1 in
  if 0 < y then
    event B; g y
  else
    event A; g y
in f *int
```

Abstract by $x = 0, 0 < y$

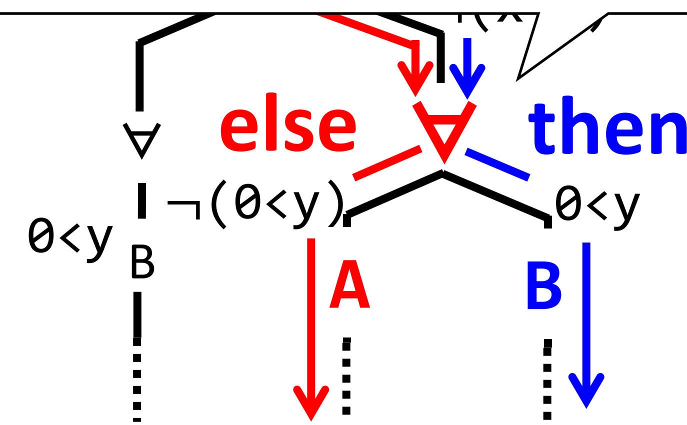
D

```
let f bx=0 =
  if bx=0 then
    ∀(B(g true))
  else
    ∀(B(g true), A(g false))
in ∃(f true, f false)
```

Computation tree of P

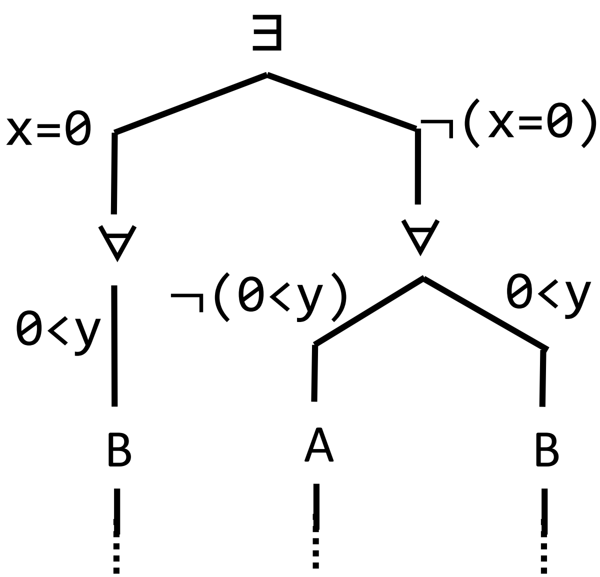


Check if **both branches** are fair and infinite



Parity Tree Automaton A_C

If **Tree**(D) is accepted by A_C,
P is **NOT** fair-terminating



Tree(D) is accepted by A_C if

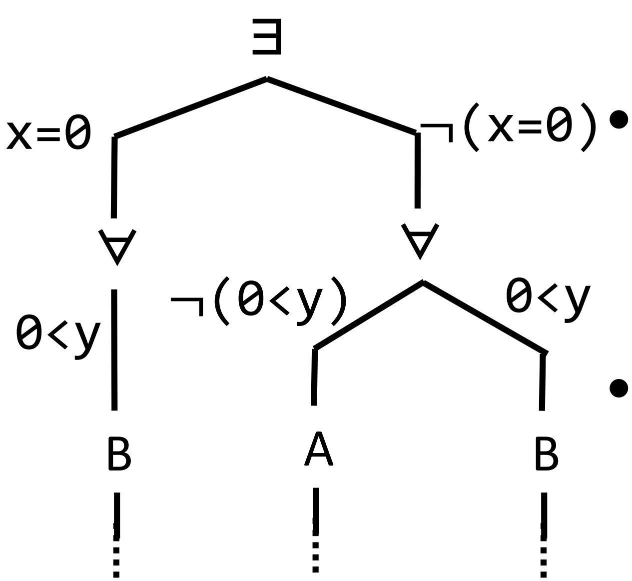
- $\exists$-node
  Some branches have fair infinite paths
- $\forall$-node
  All branches have fair infinite paths

Parity Tree Automaton A_C

Needed to express **fairness**

ted by A_C,
P is **NOT** fair-terminating

Tree(D) is accepted by A_C if



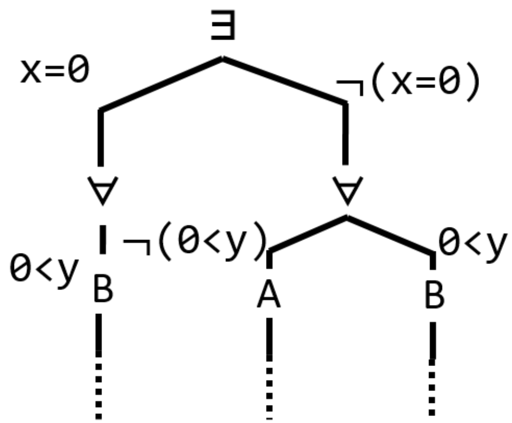
- ∃-node
  Some branches have fair infinite paths
- ∀-node
  All branches have fair infinite paths

Overview of Method

Fairn
Const

Step
M

Abstracted Tree



Decide whether the automaton accepts the abstracted tree

Step 3: Predicate Discovery

Tree Automaton

Tree Generating Program

Counterexample

Step 2： Higher-Order Model Checking

reject

accept

Fair infinite executions exist!

Step 2

Input:

- Tree generating Boolean Program D
- Parity tree automaton A_C

Output of Step 1

Output:

Whether A_C accepts $\mathbf{Tree}(D)$

If A_c rejects the tree,

counterexample will be returned

Step 2

Input:

- Tree generating Boolean Program D
- Parity tree automaton A_C

Output of Step 1

Higher-order model checking

[Ong LICS06]

Output:

Whether A_C accepts $\mathbf{Tree}(D)$

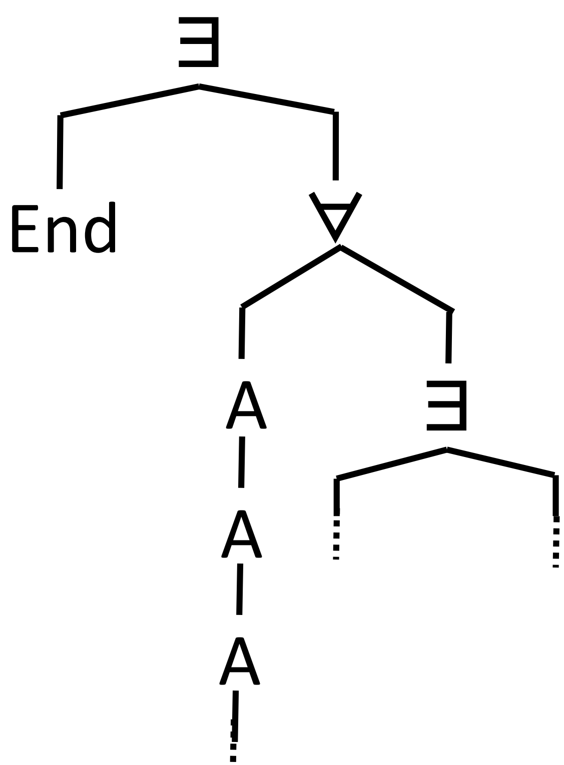
If A_c rejects the tree,

counterexample will be returned

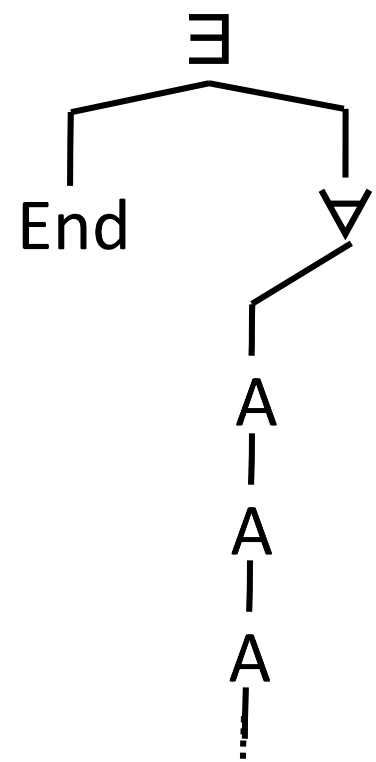
Counterexample Tree

Subtree that is **NOT** accepted by A_C

Abstracted computation tree



Counterexample tree



Counterexample Representation

Challenge:

How to represent an **infinite** counterexample tree?

Counterexample Representation

Challenge:

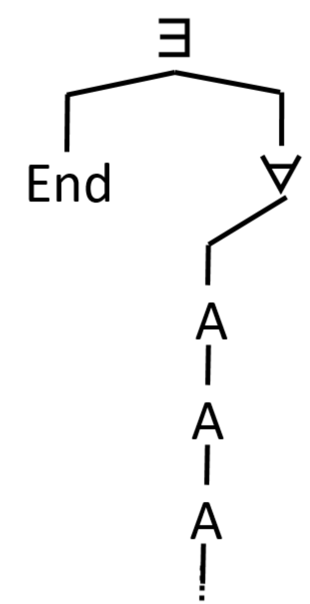
How to represent an **infinite** counterexample tree?

Solution:

Use a **finite program** that generates a counterexample tree

```
main = ∃(End, ∀ f)
f = ∀(A f)
```

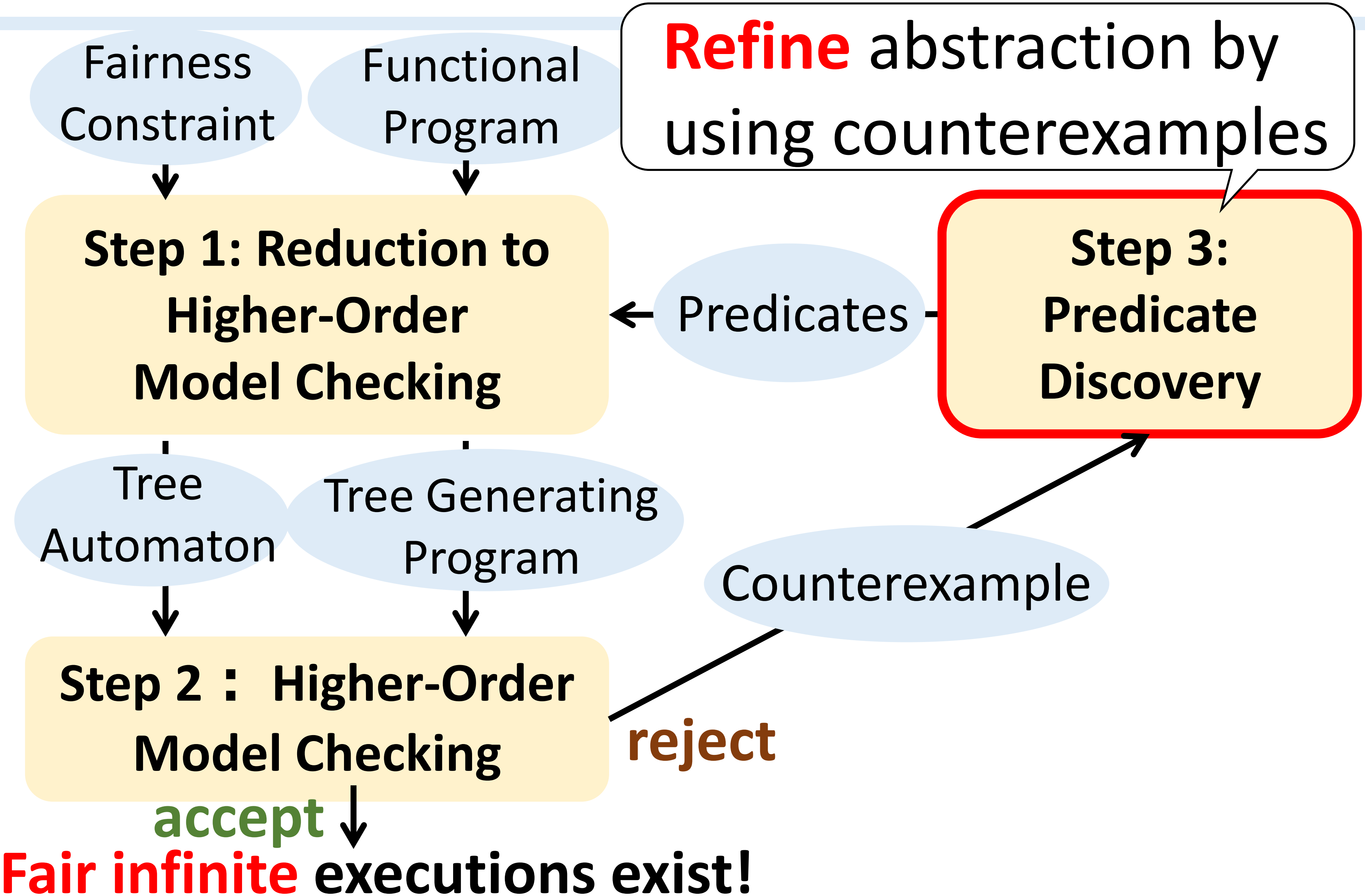
generates



cf. Type based effective selection

[Carayol&Serre LICS12] [Tsukada&Ong LICS14]

Overview of Method



Abstraction Refinement

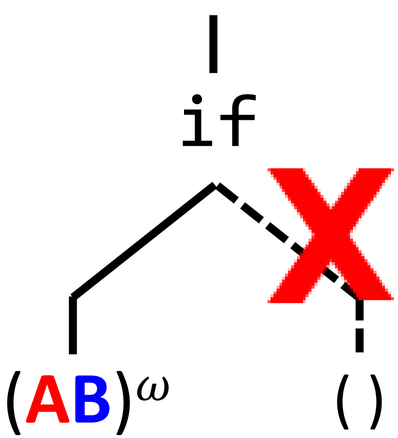
[Kobayashi+ PLDI11]
[Kuwahara+ CAV15]

Discover predicates from counterexample paths

Example: `if flag then fair_loop() else ()`

always **true**

Computation tree



Abstraction Refinement

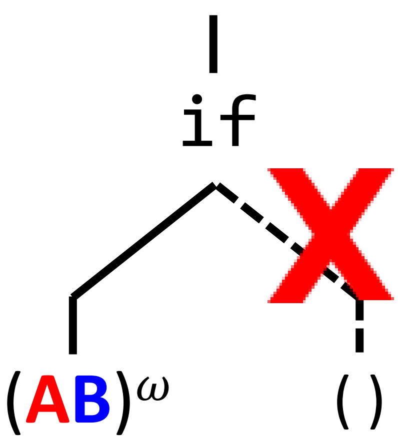
[Kobayashi+ PLDI11]
[Kuwahara+ CAV15]

Discover predicates from counterexample paths

Example:

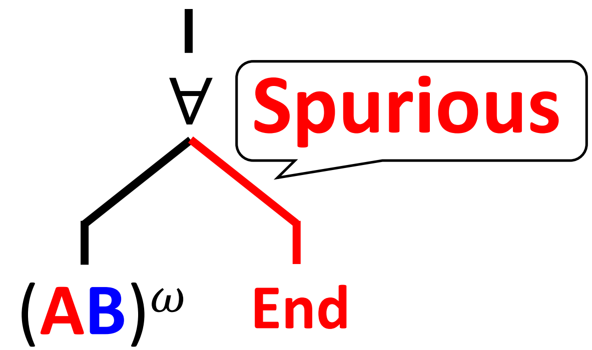
```
if flag then fair_loop() else ()
```

Computation tree



Coarse abstraction

Abstracted tree



Abstraction Refinement

[Kobayashi+ PLDI11]
[Kuwahara+ CAV15]

Discover predicates from counterexample paths

Example: `if flag then fair_loop() else ()`

Computation tree **Abstracted tree**

Discover **new predicates**
by analyzing **counterexample paths**

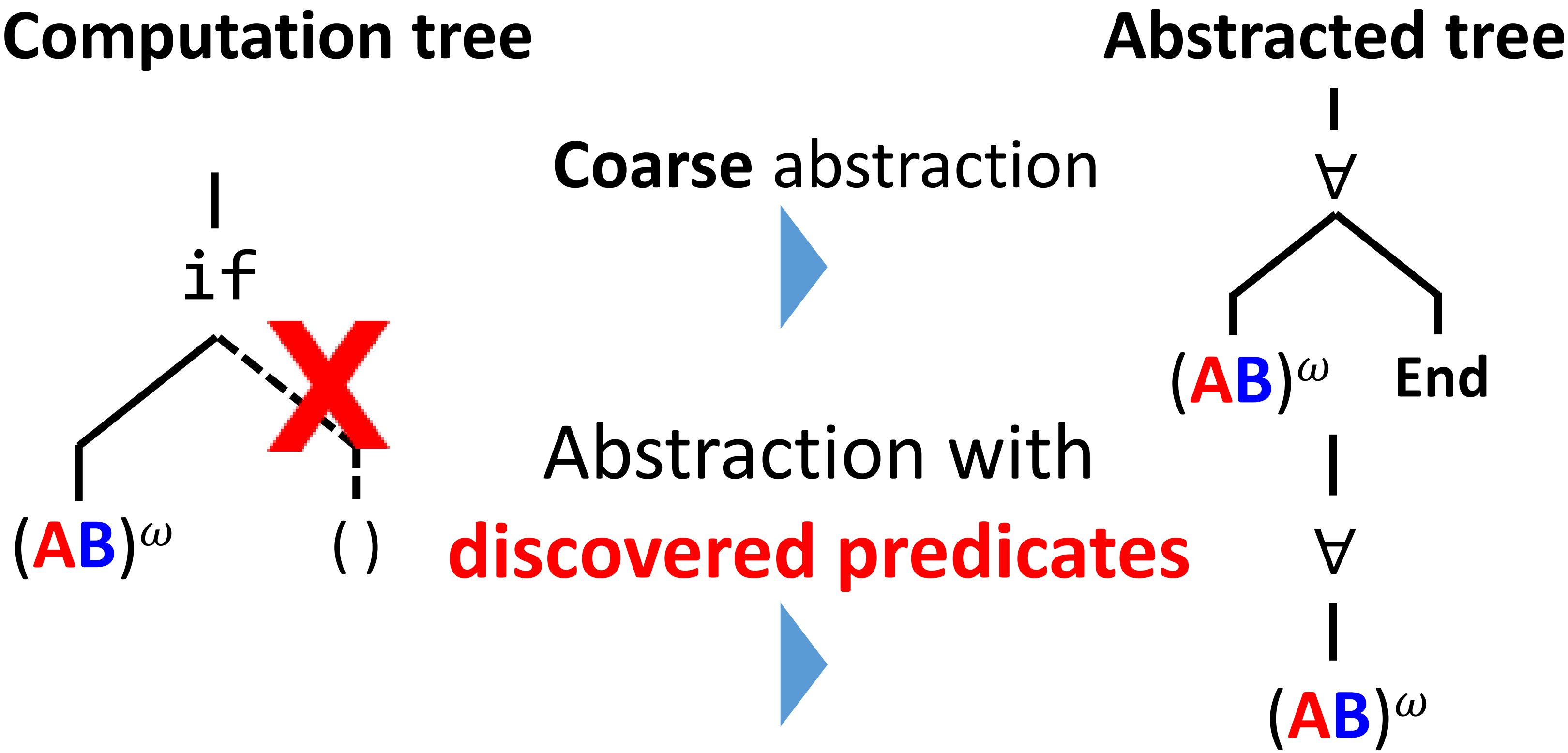
Abstraction Refinement

[Kobayashi+ PLDI11]
[Kuwahara+ CAV15]

Discover predicates from counterexample paths

Example:

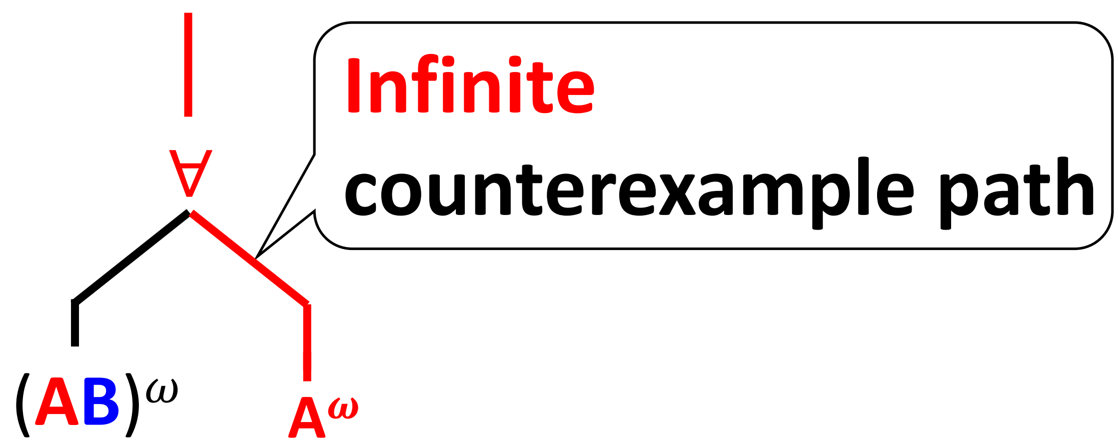
```
if flag then fair_loop() else ()
```



Predicates Discovery from Infinite Paths

Challenge:

Previous techniques are **limited to finite** counterexample paths



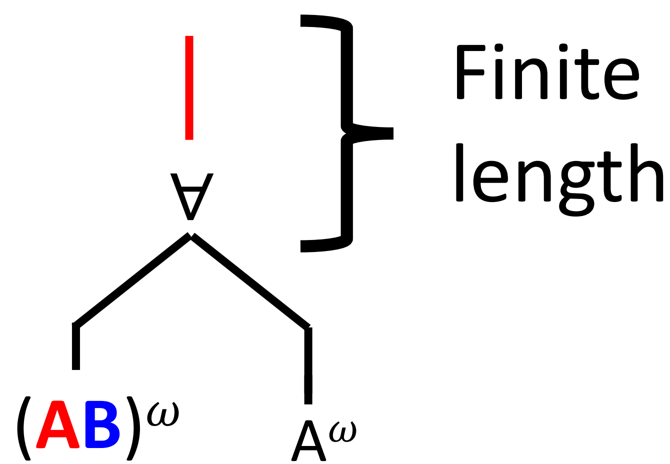
Predicates Discovery from Infinite Paths

Challenge:

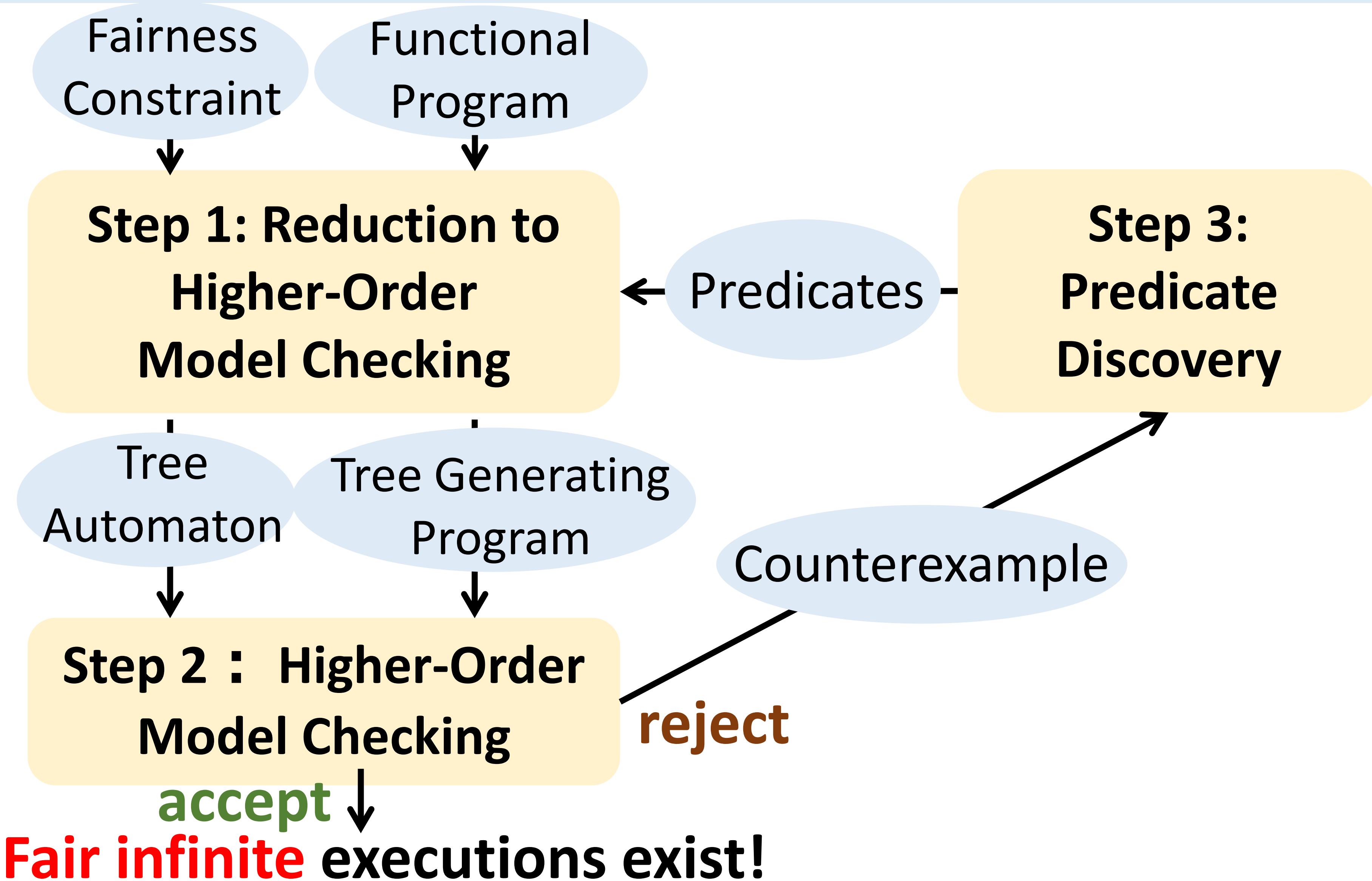
Previous techniques are **limited to finite** counterexample paths

Solution:

Use **finite prefixes** of counterexample paths



Overview of Method



Our Method is …

- Sound

- Incomplete

- Not terminating, when P is fair-terminating
  → Run a fair-termination verifier **at the same time**
  [Murase+ POPL16]

Outline

- Termination & Fair-Termination
- Importance of Fair-Termination
- Our Method
- **Implementation and Experiments**
- Related Work
- Conclusion

Implementation

- An extension of **MoCHi** [Kobayashi+ PLDI11]

- Backend
  - Higher-order model checker:
    HorSatP [Fujima 15]
    +
    Counterexample generation
  - SMT solver:
    Z3 [de Moura & Bjørner TACAS08]

Experiments

Two Benchmarks

1. Small, original benchmark programs
2. Variants of the benchmark programs in [Koskinen&Terauchi LICS14] and [Murase+ POPL16]

All programs are **NOT** fair-terminating

Experiment Results

Program	Order	Cycles	Time[sec]
murase-repeat	2	2	0.98
murase-closure	2	2	0.8
koskinen-1	2	3	2.96
koskinen-2	1	5	9.5
koskinen-3-1	1	4	4.94
koskinen-3-2	1	≧2	timeout
koskinen-3-2 (predicates given by hand)	1	1	0.87
koskinen-3-3	1	4	5.63

(Excerpt)

- Spec: Xeon E5-2680 v3 (2.50GHz, 16GB of memory)
- Time Limit: 300 seconds

Outline

- Termination & Fair-Termination
- Importance of Fair-Termination
- Our Method
- Implementation and Experiments
- **Related Work**
- **Conclusion**

Related Work

Automated verification for **higher-order** programs

- **Proving fair-termination** [Murase+ POPL16]
- **Disproving plain termination** [Kuwahara+ CAV15]

Temporal verification for **first-order** programs

- **Proving fair CTL and CTL*** properties [Cook+ TACAS15] [Cook+ CAV15]
- **Disproving** fair-termination of **multi-threaded** programs [Atig+ CAV12]

Conclusion

Automated method for **disproving fair-termination** of higher-order functional programs

- Reduction to **parity tree automata** HO model checking
- **Finite representations** of infinite counterexample trees
- Predicate discovery from **finite counterexample prefixes**

Future work

- Tighter integration with fair-termination verification
- Scalability
- General temporal property verification

Extra:

Program that Our Method Cannot Verify

```
let rec repeat n =
  if n = 0 then
    ()
  else
    (event A;
     repeat (n-1))

let rec f x =
  repeat x;
  event B;
  f (x+1)

let main = f 0
```

In order to prove the existence of fair infinite path, we must prove that **event** B occurs infinitely often

For this, we **must prove** that **repeat** eventually **terminates** for arbitrary input **x**

Our method **cannot prove** the termination automatically

Extra:

Program that Our Method Cannot Verify

```
let rec repeat n =
  if n = 0 then
    ()
  else
    (event A;
     repeat (n-1))

let rec f x =
  repeat x;
  event B;
  f (x+1)

let main = f 0
```

cf. **Termination verification** for higher-order programs [Giesl+ TOPLAS11] [Kuwahara+ ESOP14]

For this, we **must prove** that **repeat** eventually **terminates** for arbitrary input **x**

Our method **cannot prove** the termination automatically