Automatically Disproving Fair Termination of Higher-Order Functional Programs

Keiichi Watanabe, Ryosuke Sato
Takeshi Tsukada, Naoki Kobayashi
The University of Tokyo

September 20\textsuperscript{th}, 2016
ICFP 2016 at Nara
Our Goal

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

cf. Prove Fair-termination [Murase+ POPL16]

Verification of $\omega$-regular properties includes LTL 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
⇔ Every execution eventually terminates

Terminating

Not Terminating
Fair-Termination

Program $P$ is **fair-terminating**

$\iff$ 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

```ocaml
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 *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 *int

Q.
How to incorporate randomness with termination verification?
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 ()
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 ()

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]

Verification of $\omega$-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 programs.

Proving the existence of fair infinite executions includes LTL properties.

Verification of \(\omega\)-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

Step 1: Reduction to Higher-Order Model Checking
- Tree Automaton
- Tree Generating Program

Step 2: Higher-Order Model Checking
- Fairness Constraint
- Functional Program

Step 3: Predicate Discovery
- Predicates
- Counterexample

An extension of a method for disproving plain termination [Kuwahara+CAV15]
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Step 3: Predicate Discovery

Fair infinite executions exist!

Tree Automaton

Tree Generating Program

Predicate Discovery

Fair infinite paths exist

if if if if
A A B B

Counterexample

reject

accept
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Fair infinite executions exist!
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Tree Automaton

Tree Generating Program

Abstracted Tree

Computation Tree

Fair infinite paths exist

Sufficient condition

Accepted by the automaton

Fair infinite executions exist!
Overview of Method

Step 2: Higher-Order Model Checking

- **Fairness Constraint**: Functional Program
- **Step 1**: Reduction to Higher-Order Model Checking

- **Abstracted Tree**

  - Decide whether the automaton accepts the abstracted tree

- **Step 3: Predicate Discovery**

  - **Tree Automaton**
  - **Tree Generating Program**
  - **Counterexample**

- **Fair infinite executions exist!**

- **Step 2**: Higher-Order Model Checking

- **Tree Generating Program**

- **Counterexample**

- **Step 3: Predicate Discovery**
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Step 3: Predicate Discovery

Refine abstraction by using counterexamples

Counterexample

Predicates

Tree Automaton

Tree Generating Program

Functional Program

Fairness Constraint

Accept

Reject

Fair infinite executions exist!
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Step 3: Predicate Discovery

Fairness Constraint
Functional Program

Tree Automaton
Tree Generating Program

accept
reject

Counterexample
Predicates

Fair infinite executions exist!
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Step 3: Predicate Discovery

Fair infinite executions exist!

Functional Program

Tree Automaton

Tree Generating Program

Computation Tree

Fair infinite paths exist

Sufficient condition

Accepted by the automaton

Abstracted Tree

Fair infinite executions exist!
Two Branching Nodes in Abstracted Trees

\( \exists \)-node
- Represents \textbf{inherent non-determinism} in programs
  - e.g. random integer, inputs
- We should check if \textbf{there exists} a fair infinite branch

\( \forall \)-node
- Represents \textbf{non-determinism} introduced \textbf{by abstraction}
- We should check if \textbf{every branch} is fair and infinite
Two Branching Nodes in Abstracted Trees

[Kuwahara+ CAV15]

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

Computation tree of $P$

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

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

Computation tree of P

x=-1
* x=0

... if if if if...

x=-2

... if if if if...

x=1

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

Tree(D)

x=0

∃ ¬(x=0)

... ¬(0<y)

0<y
\textbf{∃-node: Inherent Non-Determinism}

\begin{align*}
\text{let } f \ x &= \\
\text{let } y &= x+1 \text{ in} \\
\text{if } 0 < y \text{ then} \\
\quad \text{event } B; \ g \ y \\
\text{else} \\
\quad \text{event } A; \ g \ y \\
\text{in } f \ \ast \int
\end{align*}

Computation tree of $P$

\begin{align*}
x = -1 & \quad \ast \quad x = 0 \\
\vdots \quad & \quad \vdots \\
x = -2 & \quad \vdots \quad x = 1 \\
\vdots \quad & \quad \vdots
\end{align*}

Abstract by $x = 0, 0 < y$

\begin{align*}
\text{let } f \ b_{x=0} &= \\
\text{if } b_{x=0} \text{ then} \\
\quad \forall (B(g \ true)) \\
\text{else} \\
\quad \forall (B(g \ true), A(g \ false)) \\
\text{in } \exists (f \ true, f \ false)
\end{align*}

Check if \textbf{either branch is fair and infinite}

Tree($D$)

$\exists \\
\neg (x = 0) \\
\forall \\
\neg (0 < y) \\
\forall \\
0 < y$
\( \forall \)-node:

\[
\begin{align*}
&\text{let } f \ x = \\
&\quad \text{let } y = x + 1 \text{ in } \\
&\quad \text{if } 0 < y \text{ then } \\
&\quad \quad \text{event } B; \ g \ y \\
&\quad \text{else} \\
&\quad \quad \text{event } A; \ g \ y \\
&\text{in } f \ * \text{ int}
\end{align*}
\]

Computation tree of \( P \)

Non-Determinism introduced by Abstraction

Abstract by \( x = 0, 0 < y \)

\[
\begin{align*}
&\text{let } f \ b x = 0 = \\
&\quad \text{if } b x = 0 \text{ then } \\
&\quad \quad \forall(B(g \text{ true})) \\
&\quad \text{else} \\
&\quad \quad \forall(B(g \text{ true}), A(g \text{ false})) \\
&\text{in } \exists(f \text{ true}, f \text{ false})
\end{align*}
\]

Tree\((D)\)
\( \forall \)-node:

Let \( f \) be defined as

\[
\begin{align*}
\text{let } & f \ x = \\
\text{let } & y = x + 1 \text{ in} \\
\text{if } & 0 < y \text{ then} \\
\text{event } & B; \ g \ y \\
\text{else} \\
\text{event } & A; \ g \ y \\
\text{in } & f \ *\ \text{int}
\end{align*}
\]

Computation tree of \( P \)

Non-Determinism introduced by Abstraction

Abstract by \( x = 0, 0 < y \)

Let \( f \) be defined as

\[
\begin{align*}
\text{let } & f \ b x = 0 = \\
\text{if } & b x = 0 \text{ then} \\
\forall ( & B(g \ \text{true})) \\
\text{else} \\
\forall ( & B(g \ \text{true}), A(g \ \text{false})) \\
\text{in } & \exists (f \ \text{true}, f \ \text{false})
\end{align*}
\]

Check if both branches are fair and infinite
Parity Tree Automaton $A_C$

If $\text{Tree}(D)$ is accepted by $A_C$, $P$ is NOT fair-terminating

$\text{Tree}(D)$ 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

$P$ 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

**Step 1:** Reduction to Higher-Order Model Checking

**Step 2:** Higher-Order Model Checking

**Step 3:** Predicate Discovery

Abstracted Tree

- $x = \emptyset$
- $\exists (x = \emptyset)$
- $\forall (0 < y)$
- $\forall (0 < y)$

Decide whether the automaton accepts the abstracted tree.

- $\forall y_B$
- $A$
- $B$

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 $\text{Tree}(D)$
If $A_C$ rejects the tree,
\textcolor{red}{counterexample} will be returned
Step 2

Input:
• Tree generating Boolean Program $D$
• Parity tree automaton $A_C$

Output of Step 1

Output:
Whether $A_C$ accepts $\text{Tree}(D)$
If $A_C$ rejects the tree, $\text{counterexample}$ will be returned

Higher-order model checking
[Ong LICS06]
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

\[
\text{main} = \exists (\text{End}, \forall f) \\
\text{f} = \forall (A \ f)
\]

cf. Type based effective selection
[Carayol&Serre LICS12] [Tsukada&Ong LICS14]
Overview of Method

Step 1: Reduction to Higher-Order Model Checking

Step 2: Higher-Order Model Checking

Step 3: Predicate Discovery

Refine abstraction by using counterexamples

Fairness Constraint → Functional Program

Tree Automaton → Tree Generating Program

Predicates

Counterexample

Fair infinite executions exist!

accept

reject
Abstraction Refinement

Discover predicates from counterexample paths

Example: \( \text{if flag then fair\_loop()} \text{ else } () \)

Computation tree always true

\[(AB)^{\omega} () \]
Abstraction Refinement

Discover predicates from counterexample paths

Example: \( \text{if flag then } \text{fair\_loop}() \text{ else } () \)

Computation tree

Abstracted tree

\[ (A\!\!B)^\omega \]

\[ \forall \]

\[ \text{Spurious} \]

End
Discover predicates from counterexample paths

Example: \[ \text{if flag then fair\_loop()} \text{ else } () \]

Discover new predicates by analyzing counterexample paths

Coarse abstraction

Abstraction Refinement

[Kobayashi+ PLDI11]
[Kuwahara+ CAV15]
Abstraction Refinement

[57x466][Kobayashi+ PLDI11]
[678x466][Kuwahara+ CAV15]

Discover predicates from counterexample paths

Example: \( \text{if flag then fair\_loop()} \text{ else } () \)

Computation tree

Abstracted tree

Coarse abstraction

Abstraction with discovered predicates
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

Fairness Constraint → Functional Program

Step 1: Reduction to Higher-Order Model Checking

Tree Automaton → Tree Generating Program

Step 2: Higher-Order Model Checking

Predicates

Step 3: Predicate Discovery

Counterexample

accept

reject

Fair infinite executions exist!
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

<table>
<thead>
<tr>
<th>Program</th>
<th>Order</th>
<th>Cycles</th>
<th>Time[sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>murase-repeat</td>
<td>2</td>
<td>2</td>
<td>0.98</td>
</tr>
<tr>
<td>murase-closure</td>
<td>2</td>
<td>2</td>
<td>0.8</td>
</tr>
<tr>
<td>koskinen-1</td>
<td>2</td>
<td>3</td>
<td>2.96</td>
</tr>
<tr>
<td>koskinen-2</td>
<td>1</td>
<td>5</td>
<td>9.5</td>
</tr>
<tr>
<td>koskinen-3-1</td>
<td>1</td>
<td>4</td>
<td>4.94</td>
</tr>
<tr>
<td>koskinen-3-2</td>
<td>1</td>
<td>≥2</td>
<td>timeout</td>
</tr>
<tr>
<td>koskinen-3-2 (predicates given by hand)</td>
<td>1</td>
<td>1</td>
<td>0.87</td>
</tr>
<tr>
<td>koskinen-3-3</td>
<td>1</td>
<td>4</td>
<td>5.63</td>
</tr>
</tbody>
</table>

*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
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

Extra:

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.
Program that Our Method Cannot Verify

```ocaml
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
```

Extra:

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.

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