Applications of Higher-Order Model Checking to Program Verification

Hiroshi Unno
University of Tsukuba
(Joint work with Naoki Kobayashi, Ryosuke Sato, Tachio Terauchi, and Takuya Kuwahara)
Success Story: Software Model Checkers for C

Prove Properties of Program Executions

Program:

Concurrency
Recursive Procedures
Heap Data Structures

Specification:

Safety
Termination
Non-termination

LTL, CTL, fair CTL, CTL*

\[ P \models \Psi \]

SLAM, BLAST, MAGIC, ...

TERMINATOR, ...
TNT, T2, ...

T2, ...
Challenge: How To Construct Software Model Checker for OCaml?

Prove Properties of Program Executions

Program:

\[ P \]

Specification:

\[ \models \psi \]

• Higher-order Functions
• Exception Handling
• Algebraic Data Structures
• Objects & Dyn. Dispatch
• General References

Safety
Termination
Non-termination
LTL, CTL, fair CTL, CTL*
This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

Program:
- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Specification:
- Safety
- Termination
- Non-termination
- $\omega$-regular properties
This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

Program: $P$

Specification: $\Psi$

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

$P \models \Psi$

- Safety
- Termination
- Non-termination
- $\omega$-regular properties
Tool Demonstration of MoCHi

• Web interface available from: http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/
Overall Flow of Safety Verification

OCaml Program

Function Encoding [Sato+ ’13]

Higher-order Integer Program

Predicate Abstraction + CEGAR [Kobayashi+ ’11]

Higher-order Boolean Program

Higher-Order Model Checking (TRecS, HorSat, C-SHORE, Preface,...)

\(\lambda \rightarrow^* +\) recursion
+ algebraic data types
+ exceptions
+ integers + booleans

\(\lambda \rightarrow^* +\) recursion
+ integers + booleans

\(\lambda \rightarrow^* +\) recursion
+ booleans

Sound and complete!
Overall Flow of Safety Verification

OCaml Program

Function Encoding [Sato+ ’13]

Higher-order Integer Program

Predicate Abstraction + CEGAR [Kobayashi+ ’11]

Higher-order Boolean Program

Higher-Order Model Checking (TRecS, HorSat, C-SHORE, Preface,...)

\[ \lambda \rightarrow + \text{recursion} \]

\[ + \text{algebraic data types} \]

\[ + \text{exceptions} \]

\[ + \text{integers + booleans} \]

\[ \lambda \rightarrow + \text{recursion} \]

\[ + \text{integers + booleans} \]

\[ \lambda \rightarrow + \text{recursion} \]

\[ + \text{booleans} \]

Sound and complete!
Higher-Order Model Checking

- A generalization of ordinary model checking:
  - Model the target system as a recursion scheme and check if it satisfies the given specification

<table>
<thead>
<tr>
<th>Model Checking</th>
<th>Verification Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite state model checking</td>
<td>Simple loops</td>
</tr>
<tr>
<td>Pushdown model checking</td>
<td>First-order recursive functions</td>
</tr>
<tr>
<td>Higher-order model checking</td>
<td>Higher-order recursive functions</td>
</tr>
</tbody>
</table>
Higher-Order Recursion Scheme (HORS)

- Grammar for generating a possibly infinite tree
Higher-Order Recursion Scheme (HORS)

- Grammar for generating a possibly infinite tree

Order-1 scheme

\[ S \rightarrow A \ c \]
\[ A \ x \rightarrow a \ x \ (A \ (b \ x)) \]

\[ S \rightarrow A \ c \rightarrow a \rightarrow a \rightarrow \ldots \rightarrow \]

\[ c \ A(b \ c) \ c \ a \ A(b(b \ c)) \]
Higher-Order Model Checking

Given

\( G: \) a recursion scheme
\( A: \) a tree automaton,

\( Tree(G) \in L(A)? \)

e.g.

- Does every finite path end with “c”?
- Does “a” occur eventually whenever “b” occurs?

- Decidable but n-EXPTIME-complete (for order-n recursion scheme) [Ong ’06]
- Practical higher-order model checkers have been developed [Kobayashi ’09,...]
HORS as a Programming Language

Recursion schemes

\[ \cong \]

Simply-typed $\lambda$-calculus

+ recursion

+ tree constructors (but no destructors)

(+ finite data domains such as booleans)
From Program Verification to Higher-Order Model Checking [Kobayashi ’09]

Higher-order boolean program + specification (on events or output) → Program Transformation

HORS (describing all event sequences or outputs) + Tree automaton recognizing valid event sequences or outputs → Model Checking
Example: From Program Verification to Higher-Order Model Checking

```ocaml
let rec f(x) =  
  if * then close(x)  
  else (read(x); f(x))  
in  
let y = open "foo"  
in  
f(y)
```

continuation parameter, expressing how "foo" is accessed after the call returns

Is the file "foo" accessed according to read* close?

Is each path of the tree labeled by \( r^* c \)?
Example: From Program Verification to Higher-Order Model Checking

```plaintext
let rec f(x) =  
    if * then close(x) 
    else (read(x); f(x)) 

let y = open "foo" 

f (y)
```

- Is the file “foo” accessed according to read* close?

- Is each path of the tree labeled by $r^* c$?

- Continuation parameter, expressing how “foo” is accessed after the call returns.

CPS Transformation!
Example: From Program Verification to Higher-Order Model Checking

```latex
\[
\text{let rec } f(x) = \\
\text{ if } * \text{ then close(x) } \\
\text{ else (read(x); } f(x) \text{) } \\
\text{ in } \\
\text{ let } y = \text{ open "foo" } \\
\text{ in } \\
\text{ f(y) }
\]
```

Continuation parameter, expressing how "foo" is accessed after the call returns.

CPS Transformation!

Is the file "foo" accessed according to read* close?

Is each path of the tree labeled by $r^* c$?
Example: From Program Verification to Higher-Order Model Checking

let rec f(x) =
  if * then close(x)
  else (read(x); f(x))
in
let y = open "foo"
in
f (y)

Is the file "foo" accessed according to read* close?

continuation parameter, expressing how "foo" is accessed after the call returns

Is each path of the tree labeled by $r^*c$?
Program Verification based on Higher-Order Model Checking [Kobayashi ’09]

Higher-order boolean program + Specification → Program Transformation → HORS + Tree automaton → Model Checking

Sound, complete, and automatic for:
- Simply-typed λ-calculus + recursion
  + tree constructors (but no destructors)
  + finite data domains (e.g. booleans)
  (but not for infinite data domains!)
- A large class of verification problems:
  resource usage verification, reachability, flow analysis, ...
Overall Flow of Safety Verification

OCaml Program

Function Encoding [Sato+ ’13]

Higher-order Integer Program

Predicate Abstraction
+ CEGAR [Kobayashi+ ’11]

Higher-order Boolean Program

Higher-Order Model Checking
(TRecS, HorSat, C-SHORE, Preface,...)

\[ \lambda \rightarrow \]
+ recursion
+ integers + booleans

\[ \lambda \rightarrow \]
+ recursion
+ integers + booleans

\[ \lambda \rightarrow \]
+ recursion
+ booleans

Sound and complete!
Predicate Abstraction [Graf & Saidi ’97]

Program

Predicate Abstraction

Boolean Program

let f x = x+1

\[ P \equiv \lambda x. x \geq 0 \]

\[ Q \equiv \lambda y. y \geq 0 \]

\[ \neg P(x) \not\Leftrightarrow Q(x + 1) \]

\[ \neg P(x) \not\Leftrightarrow \neg Q(x + 1) \]

\[ b = \text{true} \Leftrightarrow P(x) \]

let f b = if b then true else rndbool

\[ P(x) \Rightarrow Q(x + 1) \]
CEGAR [Clarke et al. ’00]

- **Program**
- **Predicate Abstraction**
- **Boolean Program**
- **Model Checking**
- **New Predicates**
- **Predicate Discovery**
- **CEGAR Loop**
- **Error Trace**
- **Feasibility Check**

**Flowchart Notes:**
- OK leads to **safe**
- NG leads to **infeasible**
- Feasible leads to **unsafe**

**Legend:**
- OK: Valid state
- NG: Invalid state
- feasible: Model check is feasible
- infeasible: Model check is not feasible
- unsafe: Model check is unsafe
CEGAR [Clarke et al. ’00]

Program → Predicate Abstraction

Predicate Abstraction → Boolean Program

Boolean Program → Model Checking

Model Checking → Error Trace

Error Trace → Feasibility Check

Feasibility Check → Predicate Discovery

Predicate Discovery → New Predicates

New Predicates → CEGAR Loop

CEGAR Loop → CEGAR Loop

CEGAR Loop → Feasibility Check

Feasibility Check → Feasibility Check

Feasibility Check → OK (safe)

OK (safe) → Feasibility Check

Feasibility Check → NG (infeasible)

NG (infeasible) → Error Trace

Error Trace → OK (feasible)

OK (feasible) → Feasibility Check

Feasibility Check → NG (unsafe)

NG (unsafe) → Error Trace
Challenges in Higher-Order Setting

• Model Checking
  – How to precisely analyze higher-order control flows?
  ⇒ Higher-order model checking!

• Predicate Abstraction
  – How to ensure consistency of abstraction?

• Predicate Discovery
  – How to find new predicates that can eliminate an infeasible error trace from the abstraction?
Challenges in Higher-Order Setting

- Predicate Abstraction
  - How to ensure consistency of abstraction?

```lambda
let sum n k = if n \leq 0 then k 0 else sum (n-1) (\lambda x'. k (x' + n))

let main m = sum m (\lambda x. assert (x $\geq$ m))
```

```
\lambda x. x' \geq n-1
(n-1) (\lambda x'. k (x' + n))
\lambda x. x \geq m
\lambda x. x \geq m
\lambda x. x \geq n
```
Our Solution: Abstraction Types

- Specify which predicates should be used for abstraction of each expression
- \text{int}[P_1, \ldots, P_n]
  Int. exps. that should be abstracted by \( P_1, \ldots, P_n \)
  e.g., \( 3 : \text{int}[\lambda x. x > 0, \text{even}?] \sim (\text{true, false}) \)
- \((x : \text{int}[P_1, \ldots, P_n]) \rightarrow \text{int}[Q_1, \ldots, Q_m]\)
  Assuming that argument \( x \) is abstracted by \( P_1, \ldots, P_n \),
  abstract the return value by \( Q_1, \ldots, Q_m \)
Example: Abstraction Types

\[
\text{sum: } (n:\text{int}[]) \rightarrow (\text{int}[\lambda x.x \geq n] \rightarrow \ast) \rightarrow \ast
\]

\[
\text{let sum } n \ k = \text{if } n \leq 0 \text{ then } k \ 0 \ \lambda x'.x' \geq n-1
\]
\[
\text{else sum } (n-1) \ (\lambda x'.k \ (x'+n)) \ \lambda x.x \geq m
\]

\[
\text{let main } m = \text{sum } m \ (\lambda x.\text{assert}(x \geq m)) \ \lambda x.x \geq n
\]

no predicates for \(n\)

predicate for abstracting the 1st argument of \(k\)

Unit type

2016/9/20
Example: Predicate Abstraction

\[
\text{let sum } n \ k = \begin{cases} \text{if } n \leq 0 \text{ then } k \ 0 & \lambda x'. x' \geq n-1 \\ \text{else } \text{sum} \ (n-1) \ (\lambda x'. k \ (x'+n)) & n > 0 \\ \end{cases}
\]

\[
\text{let main } m = \text{sum} \ m \ (\lambda x. \text{assert}(x \geq m))
\]

\[
\text{sum: } (n: \text{int}[\]) \rightarrow (\text{int}[\lambda x. x \geq n] \rightarrow \star) \rightarrow \star
\]

\[
\text{let sum } () \ k = \begin{cases} \text{if } * \text{ then } k \ \text{true} & x' \geq n-1 \land n > 0 \Rightarrow x' + n \geq n \\ \text{else } \text{sum} () \ (\lambda b'. k \ (\text{if } b' \text{ then } \text{true} \text{ else } \text{rndbool})) & \end{cases}
\]

\[
\text{let main } () = \text{sum} () \ (\lambda b. \text{assert}(b))
\]

Successfully model checked!
Type-Directed Predicate Abstraction

\[ \Gamma \vdash M : \tau \sim t \]

\[ \Gamma \vdash M : \tau' \rightarrow \tau \sim s \quad \Gamma \vdash N : \tau' \sim t \]

\[ \Gamma \vdash M N : \tau \sim s t \]

Predicate Abstraction Rule for Function Applications
Challenges in Higher-Order Setting

• Predicate Discovery
  – How to find new predicates that can eliminate an infeasible error trace from the abstraction?
Challenges in Higher-Order Setting

• Predicate Discovery
  – How to find *abstraction types* that can eliminate an infeasible error trace from the abstraction?
Our Solution

- Reduction to refinement type inference of a straightline higher-order program (SHP)
Refinement Types [Xi & Pfenning ’98, ’99]

• \( \{x : \text{int} \mid x \geq 0\} \)
  Non-negative integers

• \((x : \text{int}) \rightarrow \{r : \text{int} \mid r \geq x\}\)
  Functions that take an integer \(x\) and return an integer \(r\) not less than \(x\)

Soundness of refinement type system \( \vdash_{\text{Ref}} \):

\( P \) is safe (i.e., \( P \rightarrow^{*} \text{assert false} \))
if \( P \) is well-typed (i.e., \( \exists \Gamma. \Gamma \vdash_{\text{Ref}} P \))

FOL formulas (e.g. QFLIA) for type refinement
Example: Abstraction Type Finding (1/2)

```
let sum n k = if n ≤ 0 then k 0
  else sum (n-1) (\x'.k (x'+n))
let main m = sum m (\x.assert(x ≥ m))
```

Infeasible error trace:

```
main m → sum m (\x.assert(x ≥ m))
→ if m ≤ 0 then (\x.assert(x ≥ m)) 0 else ...
→ (\x.assert(x ≥ m)) 0
→ assert(0 ≥ m)
→ fail
```
Example: Abstraction Type Finding (2/2)

```
let sum n k = if n <= 0 then k 0
    else sum (n-1) (\x'.k (x'+n))
let main m = sum m (\x.assert(x>=m))
```

```
main m ->* if m<=0... ->* m<=0 assert(0>=m) ->_0<0 fail
```

Straightline Higher-Order Program (SHP):

```
let sum n k = assume(n<=0); k 0
let main m = sum m (\x.assert(x<m); fail)
```

[U. & Kobayashi ’09]

Abstraction Type:

```
sum: (n:int[]) -> (int[\x.x>=n] -> *) -> *
```
Refinement Type Inference
[U. & Kobayashi ’09]

SHP → Constraint Generation
   → Horn Clause Constraints
   → Constraint Solving
   → Refinement Types
Example: Constraint Generation

Straightline Higher-Order Program (SHP):
let sum n k = assume(n \leq 0); k 0
let main m = sum m (\lambda x. assume(x < m); fail)

Refinement Type Templates:
sum: (n:{n:int | P(n)}) \rightarrow
({x:int | Q(n,x)} \rightarrow \star) \rightarrow \star

Horn Clause Constraints:
T \Rightarrow P(m)
P(n) \land n \leq 0 \land x = 0 \Rightarrow Q(n,x)
P(m) \land Q(m,x) \land x < m \Rightarrow \bot
Horn Clause Constraints:

\[ \top \Rightarrow P(m) \]
\[ P(n) \land n \leq 0 \land x = 0 \Rightarrow Q(n,x) \]
\[ P(m) \land Q(m,x) \land x < m \Rightarrow \bot \]

Horn Clause Constraints with P eliminated:
\[ n \leq 0 \land x = 0 \Rightarrow Q(n,x) \]
\[ Q(n,x) \Rightarrow (n = m \Rightarrow x \geq m) \]

Solution: \[ Q(n,x) \equiv x \geq n \]
Interpolating Prover

- **Input:** $\phi_1, \phi_2$ such that $\phi_1 \Rightarrow \phi_2$
- **Output:** an **interpolant** $\phi$ of $\phi_1, \phi_2$ such that:
  1. $\phi_1 \Rightarrow \phi$
  2. $\phi \Rightarrow \phi_2$
  3. $FV(\phi) \subseteq FV(\phi_1) \cap FV(\phi_2)$

- **Example:** $x \geq n$ is an interpolant of:
  
  $n \leq 0 \land x = 0$ and $n = m \Rightarrow x \geq m$
Example: Constraint Solving (2/2)

Horn Clause Constraints:

\[ \top \Rightarrow P(m) \]

\[ P(n) \land n \leq 0 \land x = 0 \Rightarrow Q(n, x) \]

\[ P(m) \land Q(m, x) \land x < m \Rightarrow \bot \]

Substitute \( Q(n, x) \) with \( x \geq n \)

Horn Clauses with P1 substituted:

\[ \top \Rightarrow P(m) \]

\[ P(n) \Rightarrow (n \leq 0 \land x = 0 \Rightarrow x \geq n) \]

Interpolating Prover

Solution: \( P(n) \equiv \top \)
Example: Refinement Type Inference

**Straightline Higher-Order Program (SHP):**

```plaintext
let sum n k = assume(n \leq 0); k 0
let main m = sum m (\lambda x.assume(x < m); fail)
```

**Refinement Type Templates:**

```
sum: (n:{n:int|P(n)}) \rightarrow
({x:int|Q(n,x)} \rightarrow \ast) \rightarrow \ast
```

**Refinement Types of SHP:**

```
sum: (n:{n:int|\top}) \rightarrow
({x:int|x \geq n} \rightarrow \ast) \rightarrow \ast
```
Overall Flow of Safety Verification

OCaml Program

\( \lambda \rightarrow + \text{recursion} \)
+ algebraic data types
+ exceptions
+ integers + booleans

Function Encoding [Sato+ ’13]

Higher-order Integer Program

\( \lambda \rightarrow + \text{recursion} \)
+ integers + booleans

Predicate Abstraction
+ CEGAR [Kobayashi+ ’11]

Higher-order Boolean Program

\( \lambda \rightarrow + \text{recursion} \)
+ booleans

Higher-Order Model Checking
(TRecS, HorSat, C-SHORE, Preface,...)

Sound and complete!
Function Encoding of Lists

• Encode a list as a pair \((\text{len}, f)\) such that:
  – \(\text{len}\) is the length of the list
  – \(f\) is a function from an index \(i\) to the \(i\)-th element

• e.g., \([3;1;4]\) is encoded as \((3, f)\) where:
  \(f(0)=3, f(1)=1, f(2)=4,\) and undefined otherwise

```ml
let nil = (0, fun i -> ⊥)
let cons a (len, l) = (len + 1, fun i -> if i = 0 then a else l (i - 1))
let hd (len, l) = assert (len ≠ 0); l 0
let tl (len, l) = assert (len ≠ 0); (len - 1, fun i -> l (i + 1))
let is_nil (len, l) = len = 0
```
Function Encoding of Algebraic Data Structures

- Encode an algebraic data structure as a function from the path of a node to its label

```ml
type btree = Leaf of int | Node of btree * btree
```

A function $f$ such that:

- $f[[]] = \text{Node}$
- $f[1] = \text{Leaf}$
- $f[2] = \text{Node}$
- $f[1;1] = 3$
- $f[2;1] = \text{Leaf}$
- $f[2;2] = \text{Leaf}$
- $f[2;1;1] = 1$
- $f[2;2;1] = 4$
exception NotPos

let rec fact n =
  if n ≤ 0 then
    raise NotPos
  else
    try
      n × fact (n-1)
    with NotPos -> 1

CPS Trans.

type exc = NotPos

let rec fact n k exn =
  if n ≤ 0 then
    exn NotPos
  else
    fact (n-1)
      (fun r -> k (n × r))
      (fun NotPos -> k 1)
Summary: Safety Verification by MoCHi

• For finite-data HO programs: sound, complete, and fully-automatic verification by reduction to HO model checking [Kobayashi ’09]

• For infinite-data HO programs: sound and automatic (but incomplete) verification by a combination of:
  – HO model checking
  – program transformation [Sato+ ’13]

  Necessarily incomplete but often more precise than other approaches
  Sometimes relatively complete modulo certain assumptions
  – relatively complete refinement type system [U.+ ’13]
  – relatively complete predicate discovery [Terauchi & U. ’15]
This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

OCaml Program:  Specification:

\[ P \models \Psi \]

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety
Termination
Non-termination
\( \omega \)-regular properties
Termination Verification

- Automatically prove that a program terminates for every input (and non-determinism)

```ocaml
let rec fib n = 
  if n<2 then 1
  else
    fib(n-1) + fib(n-2)
let main () = fib *
```

Input → Terminating

Input → Unknown
Tool Demonstration of MoCHi

• Web interface available from:
  http://www.kb.is.s.u-tokyo.ac.jp/~kuwahara/termination/
1st Naïve Approach to Termination Verification of HO Functional Programs

• Abstract to a finite data HO program, and apply HO model checking

• Problem: many terminating programs are turned into non-terminating ones by abstraction

e.g. \( f(x) = \text{if } x<0 \text{ then } 1 \text{ else } 1+f(x-1) \) terminating

\( \rightarrow f(b_{x<0}) = \text{if } b_{x<0} \text{ then } 1 \text{ else } 1+f(*) \) non-terminating
Termination Verification for Imperative Programs

• Binary Reachability Analysis [Cook+ ’06]
  – Theorem [Podelski & Rybalchenko ’04]:
    $P$ is terminating iff
    $T^+$ is disjunctively well-founded (dwf)
    • $T$: the transition relation of $P$
    • dwf: a finite union of well-founded relations
Example: Binary Reachability Analysis

```
1: x = *;
2: while(x>0){
3:   x--;
4: }
```

\[ T^+ \subseteq \{(s, s') | s.p \text{e} < s'.p \text{e}\} \]
\[ \cup \{(s, s') | s.p \text{e} > s'.p \text{e}\} \]
\[ \cup \{(s, s') | s.x > s'.x \geq 0\}\]

Terminating!
2nd Naïve Approach to Termination Verification of HO Functional Programs

• Check that $\rightarrow^+$ is dwf by [Cook+ ’06]
  $\rightarrow$: the one-step reduction relation of the HO program $P$

• Problem: [Cook+ ’06] needs to reason about change in calling context / call stack
  – Theorem [Berardi+’14, Yokoyama’14]: [Cook+ ’06] can only prove termination of primitive recursive functions (when usable wf relations have height at most $\omega$)
2\textsuperscript{nd} Naïve Approach to Termination

let rec ack m n =
    if m = 0 then n + 1
    else if n = 0 then ack (m-1) 1
    else ack (m-1) (ack m (n-1))
let main m n = if m > 0 && n > 0 then ack m n

Terminates but transition relation is quite complex

– Theorem [Berardi+’14, Yokoyama’14]:
  [Cook+ ’06] can only prove termination of primitive recursive functions (when usable wf relations have height at most $\omega$)
Our Solution: Binary Reachability Analysis Generalized to HO [Kuwahara+ ’14]

• Theorem [Kuwahara+ ’14]:
  HO functional program $P$ is terminating iff $Call^+_P$ is dwf
  – The calling relation $Call_P$ of $P$:
    $$\{(f\tilde{v}, g\tilde{w}) \mid g\tilde{w} \text{ is called from } f\tilde{v} \text{ in an execution of } P\}$$
  – $Call^+_P = \{(f\tilde{v}, g\tilde{w}) \mid \text{main()} \rightarrow^* E[f\tilde{v}], f\tilde{v} \rightarrow^+ E'[g\tilde{w}]\}$
Example: Generalized Binary Reachability Analysis

```
let rec fib n =
  if n<2 then 1
  else fib (n-1)
    + fib (n-2)
let main()=fib(rand())
```

\[
\text{Call=}\{(\text{fib}(n),\text{fib}(n-1))|n>1\}
\cup \{(\text{fib}(n),\text{fib}(n-2))|n>1\}
\subseteq\{(\text{fib }m,\text{fib }n) \mid m>n\geq0\}
\]
Reduce Binary Reachability to Plain Reachability

• Goal: check $Call_P \subseteq W$ for some dwf $W$

• Approach: reduction to a safety verification problem by program transformation
  – To each function $f$, add an extra argument to record the argument of an ancestor call to $f$
  – Assert that $W$ holds when $f$ is called

$$fib\ n =$$
if $n<2$ then $n$
else $fib\ (n-1)+fib\ (n-2)$
main() = $fib\ (rand())$

$W = \{(m,n) \mid m>n \geq 0\}$

$$fib\ m\ n =$$
assert($m>n \geq 0$);
let $m' = \text{if } * \text{ then } m \text{ else } n$ in
if $n<2$ then $n$
else $fib\ m'\ (n-1)+fib\ m'\ (n-2)$
main() = $fib\ \bot\ (rand())$
This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

OCaml Program: 

\[ P \models \Psi \]

Specification:

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety
Termination
Non-termination
\( \omega \)-regular properties
Automata-Theoretic Approach [Vardi’91]

- Input:
  - Program \( P \)
  - \( \omega \)-regular temporal property \( \Psi \)

1. Construct \( \omega \)-automaton \( A_{\neg \Psi} \) (with a fairness acceptance condition) that recognizes \( L(\neg \Psi) \)
2. Construct product program \( P \times A_{\neg \Psi} \)
3. Verify that \( P \times A_{\neg \Psi} \) is fair terminating (i.e., no infinite execution trace that is fair)

Theorem: \( P \models \Psi \) iff \( P \times A_{\neg \Psi} \) is fair terminating
Definition: Fair Termination of $P$

- Fairness Constraint: $C = \{(A_1, B_1), \ldots, (A_n, B_n)\}$
- Infinite sequence $\pi$ is **fair** wrt $C$ if $\forall (A, B) \in C$, 
  - $A$ occurs only finitely often in $\pi$ or 
  - $B$ occurs infinitely often in $\pi$

- $P$ is **fair terminating** wrt $C$ if $P$ has no infinite execution trace that is fair wrt $C$
Fair Termination Verification for Imperative Programs [Cook+ ’07]

• Theorem:
  \( P \) is fair terminating wrt \( C \) iff \( T^{+\uparrow C} \) is dwf
  
  – \( T \): transition relation of \( P \)
  
  – fair transitive closure \( R^{+\uparrow C} \) of \( R \) is defined by:
    \[
    R^{+\uparrow C} = \left\{ (s_1, s_n) \mid \forall 1 \leq i < n. (s_i, s_{i+1}) \in R, \ s_1 \cdots s_n \text{ is fair wrt } C, n \geq 2 \right\}
    \]
    (Intuitively means the subset of \( R^+ \) that is fair wrt \( C \))

  • Finite sequence \( s_1 \cdots s_n \) is fair wrt \( C \) if \( \forall (A, B) \in C, \ A \text{ does not occur in } s_1 \cdots s_n \text{ or } B \text{ occurs in } s_1 \cdots s_n \)
1st Naïve Approach to Fair Termination Verification of HO Functional Programs

• Check that $\rightarrow^{+\upharpoonright C}$ is dwf
  $\rightarrow$ : the one-step reduction relation of the HO program $P$

• Suffers from the same problem as the 1st naïve approach to plain termination verification of HO functional programs:
  – [Cook+ ’07] needs to reason about change in calling context / call stack
2nd Naïve Approach to Fair Termination Verification of HO Functional Programs

• Check that $Call^{+\uparrow C}_P$ is dwf

• **Unsound**: There is a case that $Call^{+\uparrow C}_P$ is dwf but $P$ is not fair-terminating wrt $C$
  
  – For example,
    
    $f\ x = \text{if } x \leq 0 \text{ then } () \text{ else } (f\ 0; f\ 1)$

    $C = \{(\text{true}, f\ 0)\}$

    (fair wrt $C$ iff $f\ 0$ is called infinitely often)

    $f\ 2 \rightarrow^* f\ 0; f\ 1 \rightarrow^* f\ 1 \rightarrow^* f\ 0; f\ 1 \rightarrow^* \ldots$
Our Solution: Fair-Termination Analysis
Generalized to HO Programs [Murase+ ’16]

• Check disjunctive well-foundedness of $\triangleright_P^C$:
  $$\{(f\overline{\nu}, g\overline{w}) \mid \text{main}() \rightarrow^* E[f\overline{\nu}], f\overline{\nu} \rightarrow^+ \upharpoonright C E'[g\overline{w}]\}$$
  – Note that $\triangleright_P^C$ is $\text{Call}_P^+$ but $\rightarrow^+$ replaced by $\rightarrow^+ \upharpoonright C$

• Theorem:
  $P$ is fair-terminating wrt $C$ iff $\triangleright_P^C$ is dwf
How to Check that $\triangleright^C_P$ is dwf?

- By reduction to a safety verification problem via program transformation similar to the one for binary reachability analysis (see our POPL’16 paper [Murase+ ’16] for details)
Summary: Plain and Fair Termination Verification by MoCHi

- Naïve combination of HO model checking and predicate abstraction into HO Boolean programs is too imprecise
- Generalize binary reachability analysis to the HO setting by introducing the calling relations $\Call_P$ and $\triangleright_P^C$
This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

OCaml Program: Specification:

\[ P \models \Psi \]

- Higher-order Functions
- Exception Handling
- Algebraic Data Structures

Safety
Termination
Non-termination
\( \omega \)-regular properties
Verifying Non-Termination (or Disproving Termination) of HO programs

• Goal: prove that a program is non-terminating for some input (or for some non-deterministic choice)
  – complementary to termination verification
Our approach [Kuwahara+ ‘15]

• combine **over- and under-approximation**
  – **over-approximate** deterministic branches, and check that all the branches are non-terminating
  – **under-approximate** non-deterministic branches, and check that one of the branches is non-terminating
Our Approach: Combination of Under-/Over-approximation

let x=⋆ in
let y=⋆ in
f(x+y)

pred: x>0

exists (...
  /* case ¬x>0 */
  , ...
  /* case x>0 */
)

Only one of the branches needs to be non-terminating
Our Approach: Combination of Under-/Over-approximation

let x=* in
let y=* in
f(x+y)

∃ /* case ¬x>0 */
∃(...
/* case ¬0≤y≤x */)
...

Under-approximation: case for ¬x>0 ∧ 0≤y≤x is discarded
Our Approach: Combination of Under-/Over-approximation

\[
\begin{align*}
&\text{let } x = \ast \text{ in} \\
&\text{let } y = \ast \text{ in} \\
&f(x+y)
\end{align*}
\]

Predicate: \( x > 0 \)

Predicate: \( 0 \leq y \leq x \)

\[
\exists ( /* \text{ case } \neg x > 0 */ \\
\exists (\ldots \\
\qquad /* \text{ case } \neg 0 \leq y \leq x */ ) \\
\ldots)
\]

Under-approximation: case for \( \neg x > 0 \land 0 \leq y \leq x \) is discarded
Our Approach: Combination of Under-/Over-approximation

\[ \text{let } x = * \text{ in } \]
\[ \text{let } y = * \text{ in } \]
\[ f(x+y) \]

\[ \exists (\begin{array}{l}
\text{/* case } x > 0 */ \\
\exists (\text{/* case } -0 \leq y \leq x */ \\
\quad \ldots \\
\end{array}) \\
\quad \ldots 
\]
Our Approach:
Combination of Under-/Over-approximation

\[
\text{let } x=\ast \text{ in } \\
\text{let } y=\ast \text{ in } \\
f(x+y)
\]

\[
\exists \ (/* \text{ case } \neg x>0 */) \\
\exists (/* \text{ case } 0\leq y\leq x */) \\
\forall ( f \text{ true } /* \text{ case } x+y>0 */ , \\
\text{ f false } /* \text{ case } \neg x+y>0 */ ) \\
\]

Overapproximation:
both branches should have an infinite path
(since we don't know which branch is valid)

\[
\text{pred: } x>0 \\
\text{pred: } 0\leq y\leq x \\
\text{pred: } x+y>0
\]
Summary: Non-Termination Verification by MoCHi

• Underapproximate non-deterministic computation, and check that one of the branches has a non-terminating path

• Overapproximate deterministic computation, and check that all the branches have non-terminating paths

• Check them by using HO model checking
Conclusions

• HO model checking alone is not enough to construct practical software model checkers for OCaml, Java, ...

• It is often the case that software verification techniques developed for imperative programs cannot be reused in the HO setting
  – Types are useful for generalization to HO