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:

\[ P \]

Specification:

\[ \models \Psi \]

Concurrency
Recursive Procedures
Heap Data Structures

Safety
Termination
Non-termination

SLAM, BLAST, MAGIC, ...

TERMINATOR, ...

TNT, T2, ...

LTL, CTL, fair CTL, CTL*
Challenge: How To Construct Software Model Checker for OCaml?

Prove Properties of Program Executions

Program:

\[ P \]

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

Specification:

\[ \models \psi \]

• Safety
• Termination
• Non-termination
• LTL, CTL, fair CTL, CTL*

\[ P \models \psi \]
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

$P \models \Psi$
This Tutorial: Software Model Checker MoCHi for OCaml based on HOMC

Prove Properties of Program Executions

Program:  

Specification:  

\[ P \models \Psi \]

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

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-SHOREe, 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

Order-0 scheme

S → a c B
B → b S

S → a c B
B → b S
S → a c B
B → b S
...
Higher-Order Recursion Scheme (HORS)

• Grammar for generating a possibly infinite tree

Order-1 scheme

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

\[ S \rightarrow A \ c \rightarrow a \quad \rightarrow a \quad \rightarrow \ldots \rightarrow \]
\[ c \ A(b \ c) \quad c \ a \quad A(b(b \ c)) \]
Higher-Order Model Checking

Given

\[ G: \text{ a recursion scheme} \]
\[ A: \text{ a tree automaton,} \]

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

- 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

≈

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

```
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

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

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)
```

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

```
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?

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

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

CPS Transformation!
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 $\lambda$-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]

Let $f(x) = x + 1$

Let $f(b) = \text{if } b \text{ then } true \text{ else } \text{rnd}_\text{bool}$

Program $P \equiv \lambda x. x \geq 0$

Predicate $Q \equiv \lambda y. y \geq 0$

Predicate $b = \text{true} \iff P(x)$

Program $\neg P(x) \not\iff Q(x + 1)$

Program $\neg P(x) \not\iff \neg Q(x + 1)$

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

Program

Predicate Abstraction

Boolean Program

Model Checking

New Predicates

CEGAR Loop

Predicate Discovery

Error Trace

Feasibility Check

OK

safe

NG

feasible

infeasible

unsafe
CEGAR [Clarke et al. ’00]

Program

Predicate Abstraction

Boolean Program

Model Checking

New Predicates

CEGAR Loop

Error Trace

Feasibility Check

Predicate Discovery

safe

OK

NG

infeasible

feasible

unsafe

2016/9/20

Workshop on HOMC + CDPS

23
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?

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

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

\( \lambda x'.x' \geq n-1 \)

\( \lambda x.x \geq n \)

\( \lambda x.x \geq m \)
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., \( 3 : \text{int}[\lambda x. x > 0, \text{even?}] \sim (\text{true}, \text{false}) \)

• \( (x : \text{int}[P_1, \ldots, P_n]) \rightarrow \text{int}[Q_1, \ldots, Q_m] \)
  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

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))

sum: (n:int[]) → (int[\x.x ≥ n] → *) → *

no predicates for n

predicate for abstracting the 1st argument of k

Unit type
Example: Predicate Abstraction

Let \( \text{sum} \ n \ k = \) if \( n \leq 0 \) then \( k \) 0 \n\else \n\) \else \n\sum \ (n-1) (\lambda x'.k \ (x'+n)) \n\) \n\let \main m = \sum m (\lambda x.\text{assert}(x\geq m)) \n\) \n\sum: (n:int\[]) \rightarrow (\text{int}[\lambda x.x\geq n] \rightarrow \star) \rightarrow \star \n\)

Let \( \text{sum} () \ k = \)

if * then \( k \) true

else \sum () (\lambda b'.k \ (\text{if } b' \text{ then } true \text{ else } \text{rndbool})) \n\) \n\let \main () = \sum () (\lambda b.\text{assert}(b)) \n\) \n\Successfully model checked!
Type-Directed Predicate Abstraction

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

Abstraction Type Environment

HO Int Expression

Abstraction Type

HO Bool Expression

\[
\begin{align*}
\Gamma & \vdash M : \tau' \rightarrow \tau \leadsto s \\
\Gamma & \vdash N : \tau' \leadsto t \\
\hline
\Gamma & \vdash MN : \tau \leadsto s \ t
\end{align*}
\]

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)

Infeasible Error Trace

Straightline Higher-Order Program (SHP)

Refinement Type Inference [U. & Kobayashi ’09]

Abstraction Types

Refinement Types
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 \( x \)

FOL formulas (e.g. QFLIA) for type refinement

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

- \( P \) is safe (i.e., \( P \rightarrow^* \text{assert false} \))
- if \( P \) is well-typed (i.e., \( \exists \Gamma. \Gamma \vdash_{Ref} P \) )
Example: Abstraction Type Finding (1/2)

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

Infeasible error trace:

```
main m → sum m (λx.assert(x≥m))
→ if m ≤ 0 then (λx.assert(x ≥ m)) 0 else ...
→ (λx.assert(x≥m)) 0
→ assert(0≥m)
→ fail
```

\(m≤0\)

\(0<m\)
Example: Abstraction Type Finding (2/2)

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

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

\[
\text{main } m \to^* \begin{cases} 
\text{if } m \leq 0 \ldots \to^* _{m \leq 0} \text{assert}(0 \geq m) \to_{0 < m} \text{fail}
\end{cases}
\]

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

[U. & Kobayashi ’09]

Abstraction Type:
\[
\text{sum: } (n: \text{int}[[]]) \to (\text{int} [\lambda x. x \geq n] \to \star) \to \star
\]
Refinement Type Inference
[U. & Kobayashi ’09]
Example: Constraint Generation

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

Refinement Type Templates:
sum: (n:{n:int|P(n)}) →
({x:int|Q(n,x)} → ⋆) → ⋆

Horn Clause Constraints:
T ⇒ P(m)
P(n) ∧ n ≤ 0 ∧ x = 0 ⇒ Q(n,x)
P(m) ∧ Q(m,x) ∧ x < m ⇒ ⊥
Example: Constraint Solving (1/2)

Horn Clause Constraints:

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

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. $\text{FV}(\phi) \subseteq \text{FV}(\phi_1) \cap \text{FV}(\phi_2)$

• 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:

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

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

Horn Clauses with \( P1 \) substituted:

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

Interpolating Prover

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

Straightline Higher-Order Program (SHP):
\[
\begin{align*}
\text{let sum } n k &= \text{assume}(n \leq 0); k 0 \\
\text{let main } m &= \text{sum } m (\lambda x.\text{assume}(x < m); \text{fail})
\end{align*}
\]

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

Refinement Types of SHP:
\[
\text{sum: } (n:\{n:int|T\}) \rightarrow \\
\quad (\{x:int|x \geq n\} \rightarrow \star) \rightarrow \star
\]
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,...)

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

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

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

Sound and complete!
Function Encoding of Lists

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

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

```ocaml
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

```plaintext
type btreen = Leaf of int | Node of btreen * btreen
```

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 \times fact (n-1)
  with NotPos -> 1

CPS Trans.

let rec fact n k exn =
  if n ≤ 0 then exn NotPos
  else fact (n-1)
   (fun r -> k (n \times 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
  - predicate abstraction & discovery [Kobayashi+ ’11, U.+ ’09, ’15]
  - 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:

\[ \mathcal{P} \models \Psi \]

- 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 *
```
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: }

\[
T^+ \subseteq \{(s, s') \mid s.pc < s'.pc\}
\cup \{(s, s') \mid s.pc > s'.pc\}
\cup \{(s, s') \mid s.x > s'.x \geq 0\}
\]

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 $\omega$)
2nd 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 $\text{Call}_P^+$ is dwf
  
  – The calling relation $\text{Call}_P$ of $P$:
    $$\{(f\tilde{v},g\tilde{w}) \mid g\tilde{w} \text{ is called from } f\tilde{v} \text{ in an execution of } P\}$$
  – $\text{Call}_P^+ = \{(f\tilde{v},g\tilde{w}) \mid \text{main()} \rightarrow^* E[f\tilde{v}], f\tilde{v} \rightarrow^+ E'[g\tilde{w}]\}$
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())

Call={(fib(n),fib(n-1)) | n>1}
∪ {(fib(n),fib(n-2)) | n>1}
⊆{(fib m,fib n) | m>n≥0}
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

\[
\text{fib } n = \\
\text{if } n < 2 \text{ then } n \\
\text{else fib}(n-1) + \text{fib}(n-2) \\
\text{main()} = \text{fib(rand())}
\]

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

\[
\text{fib } m\ n = \\
\text{assert}(m > n \geq 0); \\
\text{let } m' = \text{if } * \text{ then } m \text{ else } n \text{ in} \\
\text{if } n < 2 \text{ then } n \\
\text{else fib } m' (n-1) + \text{fib } m' (n-2) \\
\text{main()} = \text{fib } \bot (\text{rand()})
\]
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
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 \( T^{+\cap C} \) is dwf

- \( T \): transition relation of \( P \)
- fair transitive closure \( R^{+\cap C} \) of \( R \) is defined by:

\[
R^{+\cap C} = \left\{ (s_1, s_n) \mid \forall 1 \leq i < n. (s_i, s_{i+1}) \in R, \right. \\
\left. s_1 \cdots s_n \text{ is fair wrt } C, n \geq 2 \right\}
\]

(Intuitively means the subset of \( R^+ \) that is fair wrt \( C \))

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

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

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

• Check that \( Call_P^{+\uparrow C} \) is dwf

• \textbf{Unsound}: There is a case that \( Call_P^{+\uparrow C} \) is dwf but \( P \) is not fair-terminating wrt \( C \)
  
  – For example,
  \[
  f \ x = \begin{cases} 
  () & \text{if } x \leq 0 \\
  (f 0; f 1) & \text{else}
  \end{cases}
  \]
  \[
  C = \{ (\text{true}, f 0) \}
  \]
  (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 $\triangleright^C_P$:
  $$\{(f\tilde{\nu}, g\tilde{\omega}) \mid \text{main}() \rightarrow^* E[f\tilde{\nu}], f\tilde{\nu} \rightarrow^{+\uparrow C} E'[g\tilde{\omega}]\}$$
  – Note that $\triangleright^C_P$ is $\text{Call}_P^+$ but $\rightarrow^+$ replaced by $\rightarrow^{+\uparrow C}$

• Theorem:
  $P$ is fair-terminating wrt $C$ iff $\triangleright^C_P$ is dwf
How to Check that $\trianglerighteq_{P}^{C}$ 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 $\text{Call}_P$ and $\triangleright_C^P$.
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
\( f(x+y) \)

Only one of the branches needs to be non-terminating

\[
\exists (\ldots
\quad /* case \ \neg x > 0 */
\quad , \ldots
\quad /* case \ x > 0 */
\quad )
\]
Our Approach: Combination of Under-/Over-approximation

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

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

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

\[
\begin{align*}
\text{let } x=&^\ast \text{ in } \text{pred: } x>0 \\
\text{let } y=&^\ast \text{ in } \text{pred: } 0 \leq y \leq x \\
f(x+y)
\end{align*}
\]

\[
\exists \left( \text{ /* case } \neg x>0 \text{ */ } \right.
\]
\[
\exists (\ldots
\end{align*}
\]

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

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

```
∀ ( /* case ¬x>0 */
    ∀ (/* case ¬0≤y≤x */
        ... 
    )
, ...
)
```
Our Approach: Combination of Under-/Over-approximation

\[
\begin{align*}
\text{let } x = * \text{ in } \quad & \text{pred: } x > 0 \\
\text{let } y = * \text{ in } \quad & \text{pred: } 0 \leq y \leq x \\
\text{f}(x+y) \quad & \text{pred: } x + y > 0
\end{align*}
\]

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