On Higher-Order Model Checking of Effectful Answer-Type-Polymorphic Programs

Taro Sekiyama

National Institute of Informatics
SOKENDAI

Ugo Dal-Lago

University of Bologna
INRIA

Hiroshi Unno

Tohoku University

Higher-Order Model Checking (HOMC)

[Ong, LICS'06; Kobayashi, JACM'13]

$$M \overset{?}{\models} \phi$$

Higher-Order Model Checking (HOMC)

[Ong, LICS'06; Kobayashi, JACM'13]

System

HO programs yielding trees

- HO recursion schemes
  (tree grammars with HO funcs)
- PCF terms
  (generating Böhm trees)

$$M \overset{?}{\models} \phi$$

Higher-Order Model Checking (HOMC)

[Ong, LICS'06; Kobayashi, JACM'13]

System

HO programs yielding trees

- HO recursion schemes
  (tree grammars with HO funcs)
- PCF terms
  (generating Böhm trees)

$$M \stackrel{?}{\models} \phi$$

Property

Predicates over trees

- MSO logical formulas
- Modal μ-calculus formulas
- Alternating parity tree automata

Higher-Order Model Checking (HOMC)

[Ong, LICS'06; Kobayashi, JACM'13]

System

HO programs yielding trees

- HO recursion schemes (tree grammars with HO funcs)
- PCF terms (generating Böhm trees)

$$M \stackrel{?}{\models} \phi$$

Property

Predicates over trees

- MSO logical formulas
- Modal μ-calculus formulas
- Alternating parity tree automata

HOMC Problem

Whether the trees yielded by M satisfy ϕ?

- Including safety and liveness verification problems (E.g., assertion checking and (non-)termination analysis)

Example

M

```
let rec f () =
  if * then close()
  else (read(); f ())
in
let _ = open("foo.txt") in
f ()
```

$?$

$\models$

ϕ

Is there no infinite path?

Example

M

```
let rec f () =
  if * then close()
  else (read(); f ())
in
let _ = open("foo.txt") in
f ()
```

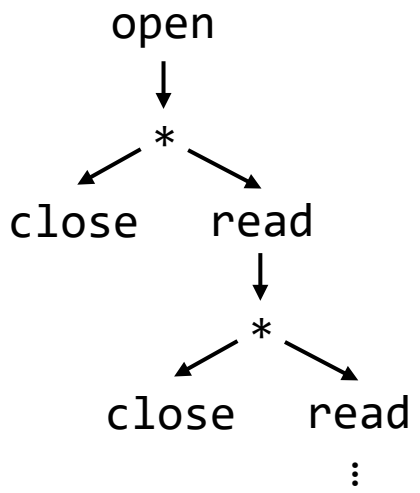
$?$

$\models$

ϕ

Is there no infinite path?

$[\![M]\!] =$



Example

M

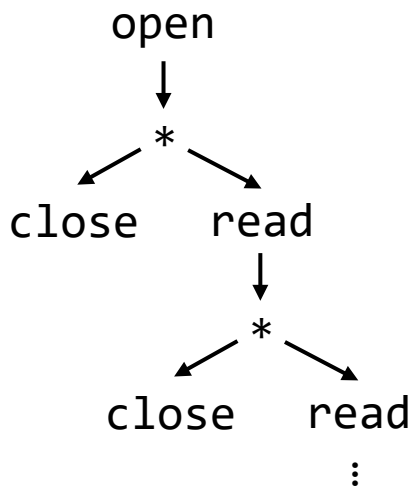
```
let rec f () =
  if * then close()
  else (read(); f ())
in
let _ = open("foo.txt") in
f ()
```

$\not\models$

ϕ

Is there no infinite path?

$[\![M]\!] =$



Example

M

```
let rec f () =
  if * then close()
  else (read(); f ())
in
let _ = open("foo.txt") in
f ()
```

?

$\models$

ϕ

Are the file ops
used in a correct order?

Example

M

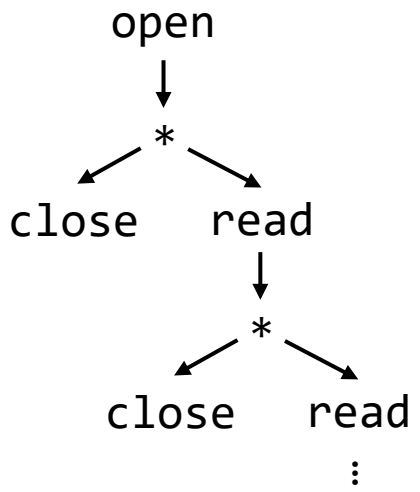
```
let rec f () =
  if * then close()
  else (read(); f ())
in
let _ = open("foo.txt") in
f ()
```

$\models ?$

ϕ

Are the file ops
used in a correct order?

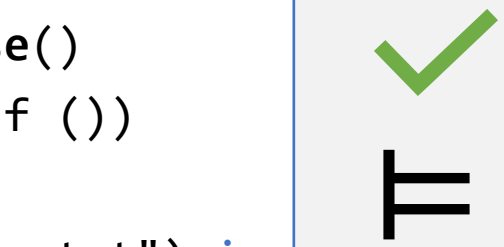
$[\![M]\!] =$



Example

M

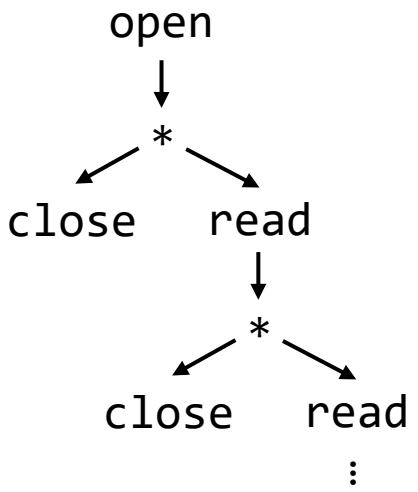
```
let rec f () =
  if * then close()
  else (read(); f ())
in
let _ = open("foo.txt") in
f ()
```



ϕ

Are the file ops
used in a correct order?

$[\![M]\!] =$



Extension with effect handlers [Dal Lago and Ghyselen, POPL'24]

- Effect handlers: features to implement **control effects**
  - Exceptions, coroutines, backtracking, etc.

M

```
with
  return x       -> x
  decide (x, k) ->
   k true;
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

$\models ?$

ϕ

Are the file ops
used in a correct order?

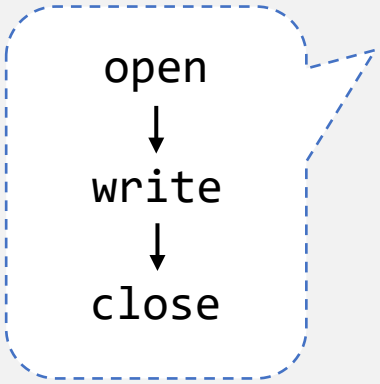
Extension with effect handlers [Dal Lago and Ghyselen, POPL'24]

- Effect handlers: features to implement **control effects**
  - Exceptions, coroutines, backtracking, etc.

M

```
with
  return x       -> x
  decide (x, k) ->
   k true;
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

$[\![M]\!]$



$\models ?$

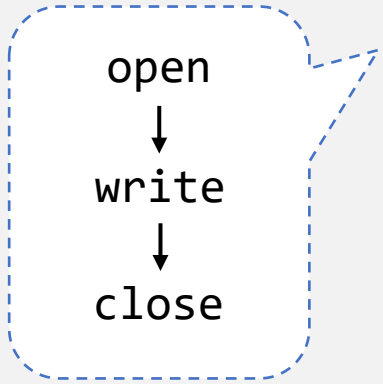
ϕ

Are the file ops
used in a correct order?

Extension with effect handlers [Dal Lago and Ghyselen, POPL'24]

- Effect handlers: features to implement **control effects**
  - Exceptions, coroutines, backtracking, etc.

$[\![M]\!]$



M

```
with
  return x       -> x
  decide (x, k) ->
   k true;
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

$\models$

ϕ

Are the file ops
used in a correct order?

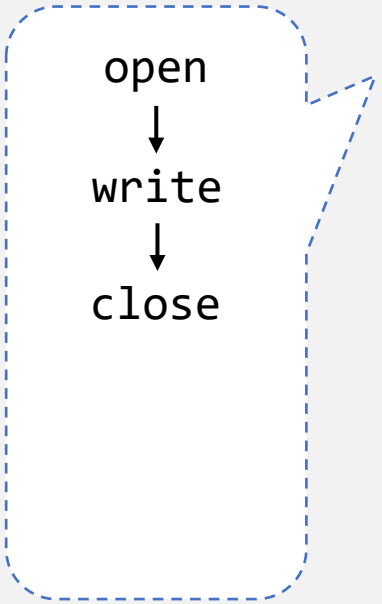
Extension with effect handlers [Dal Lago and Ghyselen, POPL'24]

- Effect handlers: features to implement **control effects**
  - Exceptions, coroutines, backtracking, etc.

M

```
with
  return x       -> x
  decide (x, k) ->
   k true; k false
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

$[\![M]\!]$



$\models ?$

ϕ

Are the file ops
used in a correct order?

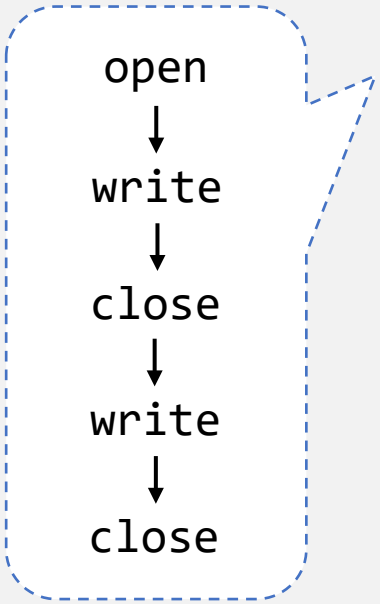
Extension with effect handlers [Dal Lago and Ghyselen, POPL'24]

- Effect handlers: features to implement **control effects**
  - Exceptions, coroutines, backtracking, etc.

M

```
with
  return x       -> x
  decide (x, k) ->
   k true; k false
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

$[\![M]\!]$



$\models ?$

ϕ

Are the file ops
used in a correct order?

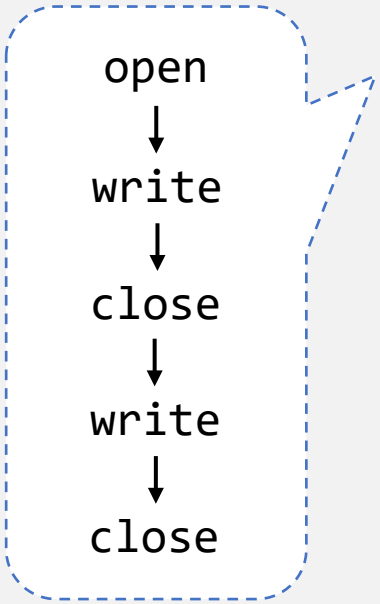
Extension with effect handlers [Dal Lago and Ghyselen, POPL'24]

- Effect handlers: features to implement **control effects**
  - Exceptions, coroutines, backtracking, etc.

M

```
with
  return x       -> x
  decide (x, k) ->
   k true; k false
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

$[\![M]\!]$



$\not\models$

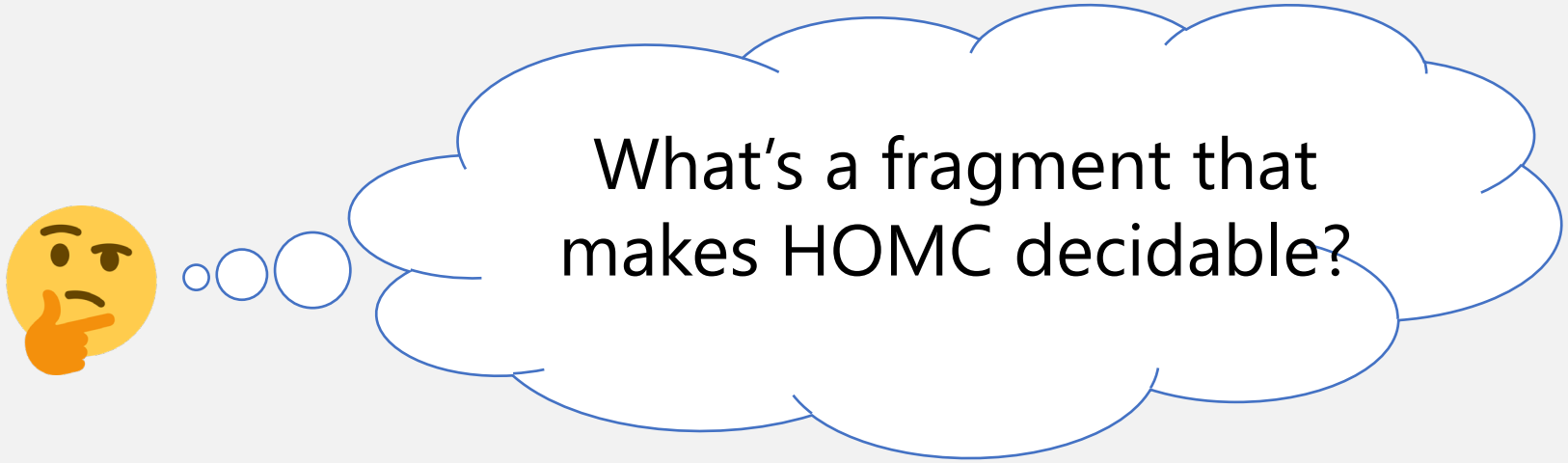
ϕ

Are the file ops
used in a correct order?

HOMC with effect handlers is **undecidable**

Because

- Effect handlers can encode natural numbers, but
- HOMC is undecidable in the presence of an infinite data domain



Contributions

Theory Identifying a class of higher-order programs where

- **HOMC is decidable** ☻
- **No restriction on effect handlers** ☻
- **No restriction on effect invocation** ☻
  if it is only handled in a tail-resumptive manner
- **Otherwise, the interpretation of effect invocation can rely only on a statically bounded number of handlers**

`op (x, k) -> k M` ($k \notin fv(M)$)

Implementation

An HO model checker for a subset of OCaml 5

- It checks an input program belongs to the above class
- If so, it model checks the program

Example

Criteria: a program is in the decidable class if

The interpretation of effect invocation can rely only on a bounded # of handlers if they are not tail-resumptive

Tail-resumptive: `op (x, k) -> k M` ($k \notin fv(M)$)

Decidable

```
with
  return x       -> x
  decide (x, k) ->
   k true
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

```
with
  return x       -> x
  decide (x, k) ->
   k true; k false
handle
  open("foo.txt");
  let s = if decide()
    then "a" else "b" in
  write(s);
  close();
```

Decidable

Because only one handler is used

Example

Criteria: a program is in the decidable class if

The interpretation of effect invocation can rely only on a bounded # of handlers if they are not tail-resumptive

Tail-resumptive: **op** (x, k) -> k M (k ∉ fv(M))

Decidable

```
let rec f () =
 with
  return x      -> x
  raise (_, k) -> ()
  // Forwarding
  *      (_, k) -> k (*)
  write (x, k) -> k (write(x))
 handle
  if * then raise()
  else (write(true); f ())
in
open("foo.txt"); f ()
```

Because

- **raise** is handled only by the nearest handler
- * and **write** are only forwarded to outer effect handlers
- The forwarding is tail-resumptive

Example

Criteria: a program is in the decidable class if

The interpretation of effect invocation can rely only on a bounded # of handlers if they are not tail-resumptive

Tail-resumptive: **op** (x, k) -> k M (k ∉ fv(M))

Decidable

```
let rec f () =
 with
  return x      -> x
  raise (_, k) -> ()
  // Tail-resumptive
  *      (_, k) -> k (not (*))
  write (x, k) -> k (write(not x))
 handle
  if * then raise()
  else (write(true); f ())
in
open("foo.txt"); f ()
```

Example

Criteria: a program is in the decidable class if

The interpretation of effect invocation can rely only on a bounded # of handlers if they are not tail-resumptive

Tail-resumptive: **op** (x, k) -> k M (k ∉ fv(M))

No guarantee

```
let rec f () =
 with
  return x      -> x
  raise (_, k) -> ()
  // Non-tail-resumptive
  *     (_, k) -> not (k (*))
  write (x, k) -> k (write(x)); k (write(x));
 handle
  if * then raise()
  else (write(true); f ())
in
open("foo.txt"); f ()
```

Because, regarding * and **write**,

- The interpretations rest on an unbounded # of handlers, since
  - their handlers calls themselves
  - each recursive call installs one handler
- The handling is ***not*** tail-resumptive

Formalization

HEPCF

HOMC is undecidable

EPCF

HOMC is decidable

Formalization

EPCF + effect handlers

λ + fix + finite data domains + algebraic effects

HEPCF

EPCF

Typeable with
answer-type modification (ATM)
answer-type polymorphism (ATP)
[Kawamata+ POPL'24]

HOMC is undecidable

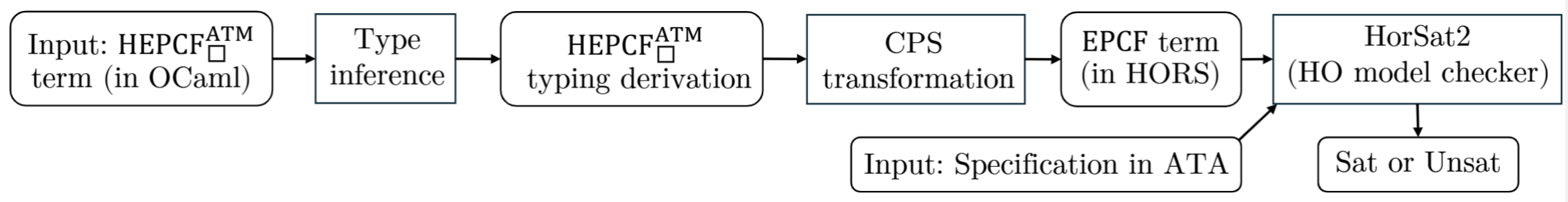
CPS
transformation

HOMC is decidable

- ATM can bound the # of handlers used to interpret effect invocation
- ATP can allow the use of an unbounded # of handlers if they are tail-resumptive

Implementation

- An HO model checker on a subset of OCaml5



- For small benchmarks, the verification completed in less than 0.1s

Contributions

Theory Identifying a class of higher-order programs where

- **HOMC is decidable** ☻
- **No restriction on effect handlers** ☻
- **No restriction on effect invocation** ☻
  if it is only handled in a tail-resumptive manner

`op (x, k) -> k M` ($k \notin fv(M)$)

- **Otherwise, the interpretation of effect invocation can rely only on a statically bounded number of handlers**

Implementation

An HO model checker for a subset of OCaml 5

- It checks an input program belongs to the above class
- If so, it model checks the program

https://github.com/hiroshi-unno/coar/