Propositional Dynamic Logic for Higher-Order Functional Programs

Yuki Satake (University of Tsukuba)
Hiroshi Unno (University of Tsukuba)

Higher-Order Functional Programs

- A higher-order function takes a function as its argument and/or returns a function as its return value

$$\mathbf{twice} : (\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int} \rightarrow \mathbf{int}$$
$$\mathbf{twice}\ f\ x = f\ (f\ x)$$

- Support in Programming Languages:
  - OCaml, Haskell, Rust
  - Jave(+1.8), Scala, Ruby, Python

Contribute significantly to modularity

Higher-Order Functional Programs

- A higher-order function takes a function as its argument and/or returns a function as its return value

takes a function as its argument

$$\mathbf{twice} : (\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int} \rightarrow \mathbf{int}$$
$$\mathbf{twice}\ f\ x = f\ (f\ x)$$

- Support in Programming Languages:
  - OCaml, Haskell, Rust
  - Jave(+1.8), Scala, Ruby, Python

Contribute significantly to modularity

Higher-Order Functional Programs

- A higher-order function takes a function as its argument and/or returns a function as its return value

takes a function as its argument

returns a function as its result

$$\textbf{twice} : (\textbf{int} \rightarrow \textbf{int}) \rightarrow \textbf{int} \rightarrow \textbf{int}$$

$$\textbf{twice}\ f\ x = f\ (f\ x)$$

- Support in Programming Languages:
  - OCaml, Haskell, Rust
  - Jave(+1.8), Scala, Ruby, Python

Contribute significantly to modularity

Higher-Order Functional Programs

- A higher-order function takes a function as its argument and/or returns a function as its return value

takes a function as its argument

returns a function as its result

$$\mathbf{twice} : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$$

$$\mathbf{twice}\ f\ x = f\ (f\ x)$$

- Support in Programming Languages:
  - OCaml, Haskell, Rust
  - Jave(+1.8), Scala, Ruby, Python

Contribute significantly to modularity

Example: Higher-Order Program

Higher Order Program

$\mathbf{let}\ r = * \ \mathbf{in}$
$\mathbf{let\ inc}\ x = x + 1\ \mathbf{in}$
$\mathbf{let\ twice}\ f\ x = f\ (f\ x)\ \mathbf{in}$
$\mathbf{twice\ inc}\ r$

Reduction Sequence for r=0

$\underline{\mathbf{twice\ inc}}\ \mathbf{0}$

Example: Higher-Order Program

Non-deterministic integer

Higher Order Program

$$\mathbf{let}\ r = * \ \mathbf{in}$$
$$\mathbf{let\ inc}\ x = x + 1\ \mathbf{in}$$
$$\mathbf{let\ twice}\ f\ x = f\ (f\ x)\ \mathbf{in}$$
$$\mathbf{twice\ inc}\ r$$

Reduction Sequence for r=0

$\underline{\mathbf{twice\ inc}}\ \mathbf{0}$

Example: Higher-Order Program

Non-deterministic integer

Higher Order Program

$\mathbf{let}\ r = * \ \mathbf{in}$
$\mathbf{let\ inc}\ x = x + 1\ \mathbf{in}$
$\mathbf{let\ twice}\ f\ x = f\ (f\ x)\ \mathbf{in}$
$\mathbf{twice\ inc}\ r$

Higher-Order function **twice**

Reduction Sequence for r=0

twice inc 0

Example: Higher-Order Program

Non-deterministic integer

Higher Order Program

$$\begin{array}{l}\mathbf{let}\ r = *\ \mathbf{in}\\ \mathbf{let\ inc}\ x = x + 1\ \mathbf{in}\\ \mathbf{let\ twice}\ f\ x = f\ (f\ x)\ \mathbf{in}\\ \mathbf{twice\ inc}\ r\end{array}$$

Higher-Order function $\mathbf{twice}$

Reduction Sequence for r=0

$$\underline{\mathbf{twice\ inc}}\ \mathbf{0}$$

$$\rightarrow \underline{(\lambda \mathrm{x}.\ \mathbf{inc}\ (\mathbf{inc}\ \mathrm{x}))\ \mathbf{0}}$$

Example: Higher-Order Program

Non-deterministic integer

Higher Order Program

$\mathbf{let}\ r = *\ \mathbf{in}$
$\mathbf{let\ inc}\ x = x + 1\ \mathbf{in}$
$\mathbf{let\ twice}\ f\ x = f\ (f\ x)\ \mathbf{in}$
$\mathbf{twice\ inc}\ r$

Higher-Order function **twice**

Reduction Sequence for r=0

$\underline{\mathbf{twice\ inc\ 0}}$
$\to\ \underline{(\lambda \mathrm{x}.\ \mathbf{inc}\ (\mathbf{inc}\ \mathrm{x}))\ \mathbf{0}}$
$\to\ \mathbf{inc}\ \underline{(\mathbf{inc}\ \mathbf{0})}$

Example: Higher-Order Program

Non-deterministic integer

Higher Order Program

$\mathbf{let}\ r = *\ \mathbf{in}$
$\mathbf{let\ inc}\ x = x + 1\ \mathbf{in}$
$\mathbf{let\ twice}\ f\ x = f\ (f\ x)\ \mathbf{in}$
$\mathbf{twice\ inc}\ r$

Higher-Order function **twice**

Reduction Sequence for r=0

$\underline{\mathbf{twice\ inc}}\ \mathbf{0}$
$\rightarrow\ \underline{(\lambda \mathrm{x}.\ \mathbf{inc}\ (\mathbf{inc}\ \mathrm{x}))\ \mathbf{0}}$
$\rightarrow\ \mathbf{inc}\ (\underline{\mathbf{inc}\ \mathbf{0}})$
$\rightarrow^*\ \underline{\mathbf{inc}\ \mathbf{1}} \rightarrow^*\ \mathbf{2}$

Temporal Verification of Higher-Order Programs

- Specification languages used in existing verification methods
  - (ω-)regular word languages (that subsume LTL) [Disney+ '11; Hofmann+ '14; Murase+ '16]
  - Modal μ-calculus (that subsumes CTL) [Fujima+ '13; Lester+ '11; Suzuki+ '17]
  - (Extended) dependent refinement types
    - Temporal properties [Koskinen+ '14; Nanjo+ '18]
    - Temporal and branching properties [Unno+ '18]

Temporal Verification of Higher-Order Programs

- Specification languages used in existing verification methods
  - (ω-)regular word languages (that subsume LTL) [Disney+ '11; Hofmann+ '14; Murase+ '16]
  - Modal μ-calculus (that subsumes CTL) [Fujima+ '13; Lester+ '11; Suzuki+ '17]
  - (Extended) dependent refinement types
    - Temporal properties [Koskinen+ '14; Nanjo+ '18]
    - Temporal and branching properties [Unno+ '18]

☹ **they cannot sufficiently express temporal specifications that involve higher-order functions**

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
&\quad \mathbf{twice\ inc\ 0} \\
&\rightarrow\ (\lambda \mathbf{x}.\,\mathbf{inc\ (inc\ x)})\ \ \mathbf{0} \\
&\rightarrow\ \ \mathbf{inc\ (inc\ 0)} \\
&\rightarrow^{*}\ \mathbf{inc\ 1} \rightarrow^{*}\ \mathbf{2}
\end{aligned}
$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
& \mathbf{twice\ inc\ 0} \\
\rightarrow\ & (\lambda \mathrm{x.\ inc\ (inc\ x)})\ \mathbf{0} \\
\rightarrow\ & \mathbf{inc\ (inc\ 0)} \\
\rightarrow^*\ & \mathbf{inc\ 1} \rightarrow^* \mathbf{2}
\end{aligned}
$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$\begin{aligned}
& \textbf{twice inc 0} \\
\to\ & (\lambda x.\,\mathbf{inc}\,(\mathbf{inc}\,x))\ \ \mathbf{0} \\
\to\ & \mathbf{inc}\,(\mathbf{inc}\,\mathbf{0}) \\
\to^*\ & \mathbf{inc}\,\mathbf{1} \to^* \mathbf{2}
\end{aligned}$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
&\mathbf{twice\ inc\ 0} \\
\to\ & (\lambda x.\ \mathbf{inc}\ (\mathbf{inc}\ x))\ \mathbf{0} \\
\to\ & \mathbf{inc}\ (\mathbf{inc}\ 0) \\
\to^*\ & \mathbf{inc}\ 1 \to^* 2
\end{aligned}
$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
& \mathbf{twice\ inc\ 0} \\
\rightarrow\ & (\lambda x.\, \mathbf{inc\ (inc}\ x))\ \mathbf{0} \\
\rightarrow\ & \mathbf{inc\ (inc\ 0)} \\
\rightarrow^*\ & \mathbf{inc\ 1} \rightarrow^* \mathbf{2}
\end{aligned}
$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
& \mathbf{twice}\ \mathbf{inc}\ \mathbf{0} \\
\rightarrow\ & (\lambda \mathbf{x}.\ \mathbf{inc}\ (\mathbf{inc}\ \mathbf{x}))\ \ \mathbf{0} \\
\rightarrow\ & \mathbf{inc}\ (\mathbf{inc}\ \mathbf{0}) \\
\rightarrow^*\ & \mathbf{inc}\ \mathbf{1} \rightarrow^* \mathbf{2}
\end{aligned}
$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
&\quad \mathbf{twice\ inc\ 0} \\
\to &\quad (\lambda x.\, \mathbf{inc}\ (\mathbf{inc}\ x))\ \ \mathbf{0} \\
\to &\quad \mathbf{inc}\ (\mathbf{inc\ 0}) \\
\to^* &\quad \mathbf{inc\ 1} \to^* \mathbf{2}
\end{aligned}
$$

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Example: Property that cannot be expressed by the previous specification languages

$$
\begin{aligned}
& \mathrm{twice\ inc\ 0} \\
\to\ & (\lambda x.\, \mathrm{inc}\ (\mathrm{inc}\ x))\ \ 0 \\
\to\ & \mathrm{inc}\ (\mathrm{inc}\ 0) \\
\to^{*}\ & \mathrm{inc}\ 1 \to^{*} 2
\end{aligned}
$$

Previous languages cannot refer to the control flow of higher-order programs involving a function that is passed to or returned by a higher-order function

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

Our Contributions

- **Higher-Order Trace (HOT)** that captures the control flow of higher-order programs
- **Higher-Order Trace Propositional Dynamic Logic (HOT-PDL)**:
  - Propositional Dynamic Logic (PDL) over HOTs for specifying temporal properties of higher-order programs
- Decidability of HOT-PDL model checking of higher-order programs via a reduction to higher-order model checking

  (See the paper for details)
- Applications (See the paper for details)
  - Modeling and extending dependent refinement types
  - Modeling and extending stack-based access control properties

Our Contributions

- Higher-Order Trace (HOT) that captures the control flow of higher-order programs
- Higher-Order Trace Propositional Dynamic Logic (HOT-PDL):
  - Propositional Dynamic Logic (PDL) over HOTs for specifying temporal properties of higher-order programs
- Decidability of HOT-PDL model checking of higher-order programs via a reduction to higher-order model checking

  (See the paper for details)
- Applications (See the paper for details)
  - Modeling and extending dependent refinement types
  - Modeling and extending stack-based access control properties

Higher-Order Trace (HOT)

- A sequence of **call and return events** equipped with **two kinds of pointers**:
  - **Call and return events**
    - call(f,x): a call event of the function f with the argument x.
    - ret(f, x): a return event of the function f with the return value x.

Inspired by the notion of justification pointers from the game semantics of PCF

  - **Two kinds of pointers labeled with:**
    - CR : capture the correspondence between call and return events
    - CC, RC: capture higher-order control flow involving a function that is passed to or returned by a higher-order function

Higher-Order Trace (HOT)

- A sequence of **call and return events** equipped with **two kinds of pointers**:
  - **Call and return events**
    - call(f,x): a call event of the function f with the argument x.
    - ret(f, x): a return event of the function f with the return value x.

Inspired by the notion of justification pointers from the game semantics of PCF

  - **Two kinds of pointers labeled with:**
    - CR : capture the correspondence between call and return events
    - CC, RC: capture higher-order control flow involving a function that is passed to or returned by a higher-order function

Higher-Order Trace (HOT)

- A sequence of **call and return events** equipped with **two kinds of pointers**:
  - **Call and return events**
    - call(f,x): a call event of the function f with the argument x.
    - ret(f, x): a return event of the function f with the return value x.

Inspired by the notion of justification pointers from the game semantics of PCF

  - **Two kinds of pointers labeled with:**
    - CR : capture the correspondence between call and return events
    - CC, RC: capture higher-order control flow involving a function that is passed to or returned by a higher-order function

Higher-Order Trace (HOT)

- A sequence of **call and return events** equipped with **two kinds of pointers**:
  - **Call and return events**
    - call(f,x): a call event of the function f with the argument x.
    - ret(f, x): a return event of the function f with the return value x.
  - **Two kinds of pointers labeled with:**
    - CR : capture the correspondence between call and return events
    - CC, RC: capture higher-order control flow involving a function that is passed to or returned by a higher-order function

Inspired by the notion of justification pointers from the game semantics of PCF

Higher-Order Trace (HOT)

- A sequence of **call and return events** equipped with **two kinds of pointers**:
  - **Call and return events**
    - call(f,x): a call event of the function f with the argument x.
    - ret(f, x): a return event of the function f with the return value x.

Inspired by the notion of justification pointers from the game semantics of PCF

  - **Two kinds of pointers labeled with:**
    - CR : capture the correspondence between call and return events
    - CC, RC: capture higher-order control flow involving a function that is passed to or returned by a higher-order function

Higher-Order Trace (HOT)

- A sequence of **call and return events** equipped with **two kinds of pointers**:
  - **Call and return events**
    - call(f,x): a call event of the function f with the argument x.
    - ret(f, x): a return event of the function f with the return value x.
  - **Two kinds of pointers labeled with:**
    - CR : capture the correspondence between call and return events
    - CC, RC: capture higher-order control flow involving a function that is passed to or returned by a higher-order function

Inspired by the notion of justification pointers from the game semantics of PCF

Example: Event sequence of $\mathbf{twice\ inc\ 0}$

$$
\begin{aligned}
& \mathbf{twice\ inc\ 0} \\
\rightarrow\ & (\boldsymbol{\lambda}\mathbf{x.\,inc\ (inc\ x)})\ \ \mathbf{0} \\
\rightarrow\ & \mathbf{inc\ (inc\ 0)} \\
\rightarrow^*\ & \mathbf{inc\ 1} \rightarrow^* \mathbf{2}
\end{aligned}
$$

Example: Event sequence of twice inc 0

$$\underline{\text{twice inc }}\ 0$$
$$\to\ (\lambda x.\ \text{inc}\ (\text{inc}\ x))\ \ 0$$
$$\to\ \ \text{inc}\ (\text{inc}\ 0)$$
$$\to^{*}\ \text{inc}\ 1 \to^{*}\ 2$$

call(twice, ●)

Example: Event sequence of $\mathbf{twice\ inc\ 0}$

$$\mathbf{twice\ inc\ 0}$$
$$\rightarrow (\lambda x.\ \mathbf{inc}\ (\mathbf{inc}\ x))\ \mathbf{0}$$
$$\rightarrow \mathbf{inc}\ (\mathbf{inc}\ \mathbf{0})$$
$$\rightarrow^{*} \mathbf{inc}\ \mathbf{1} \rightarrow^{*} \mathbf{2}$$

The function passed to a higher-order function

call(twice, ●)

Example: Event sequence of twice inc 0

$$\underline{\text{twice inc }}0$$
$$\to\ (\lambda x.\,\text{inc}\ (\text{inc}\ x))\ \ 0$$
$$\to\ \text{inc}\ (\text{inc}\ 0)$$
$$\to^{*}\ \text{inc}\ 1 \to^{*}\ 2$$

call(twice, ●) ret(twice, ●)

Example: Event sequence of $\mathbf{twice\ inc\ 0}$

$$\underline{\mathbf{twice\ inc}}\ \mathbf{0}$$
$$\rightarrow\ (\lambda x.\ \mathrm{inc}\ (\mathrm{inc}\ x))\ \ \mathbf{0}$$
$$\rightarrow\ \ \mathbf{inc}\ (\mathbf{inc}\ \mathbf{0})$$
$$\rightarrow^*\ \mathbf{inc}\ \mathbf{1} \rightarrow^*\ \mathbf{2}$$

call(twice, ●) **ret(twice, ●)**

The function returned by a higher-order function

Example: Event sequence of twice inc 0

$$
\begin{array}{ll}
 & \mathbf{twice\ inc\ 0} \\
\to & \underline{(\lambda x.\, \mathbf{inc\ (inc\ x)})\ \ \mathbf{0}} \\
\to & \mathbf{inc\ (inc\ 0)} \\
\to^{*} & \mathbf{inc\ 1} \to^{*} \mathbf{2}
\end{array}
$$

call(twice, ●) ret(twice, ●) **call(●, 0)**

Example: Event sequence of $\mathbf{twice\ inc\ 0}$

$$
\begin{aligned}
& \mathbf{twice\ inc\ 0} \\
\to\ & (\boldsymbol{\lambda}\mathbf{x.\,inc\ (inc\ x)})\ \ \mathbf{0} \\
\to\ & \mathbf{inc\ (inc\ 0)} \\
\to^*\ & \mathbf{inc\ 1} \to^* \mathbf{2}
\end{aligned}
$$

$$
\begin{gathered}
\mathrm{call}(\mathrm{twice}, \bullet)\ \mathrm{ret}(\mathrm{twice}, \bullet)\ \mathrm{call}(\bullet, 0)\ \mathrm{call}(\bullet, 0) \\
\mathrm{call}(\mathrm{inc}, 0)\ \mathrm{ret}(\mathrm{inc}, 1)\ \mathrm{ret}(\bullet, 1)\ \mathrm{call}(\bullet, 1) \\
\mathrm{call}(\mathrm{inc}, 1)\ \mathrm{ret}(\mathrm{inc}, 2)\ \mathrm{ret}(\bullet, 2)\ \mathrm{ret}(\bullet, 2)
\end{gathered}
$$

Example: HOT that models the reduction sequence of $\mathbf{twice\ inc\ 0}$

$\mathrm{call}(\mathrm{twice}, \bullet)$

$\mathrm{ret}(\mathrm{twice}, \bullet)$

$\mathrm{call}(\bullet, 0)$

$\mathrm{call}(\bullet, 0)$

$\mathrm{call}(\mathrm{inc}, 0)$

$\mathrm{ret}(\mathrm{inc}, 1)$

$\mathrm{ret}(\bullet, 1)$

$\mathrm{call}(\bullet, 1)$

$\mathrm{call}(\mathrm{inc}, 1)$

$\mathrm{ret}(\mathrm{inc}, 2)$

$\mathrm{ret}(\bullet, 2)$

$\mathrm{ret}(\bullet, 2)$

Example: HOT that models the reduction sequence of twice inc 0

CR

call(twice, ●)
ret(twice, ●)
call(●, 0)
call(●, 0)
call(inc, 0)
ret(inc, 1)
ret(●, 1)
call(●, 1)
call(inc, 1)
ret(inc, 2)
ret(●, 2)
ret(●, 2)

CR

Correspondence between call and return events

Example: HOT that models the reduction sequence of twice inc 0

CC

CR

call(twice, ●)

ret(twice, ●)

RC

call(●, 0)

call(●, 0)

Correspondence between call and return events

call(inc, 0)

ret(inc, 1)

CC

ret(●, 1)

call(●, 1)

call(inc, 1)

CR

ret(inc, 2)

ret(●, 2)

ret(●, 2)

Higher-order control flow

CC : call of the function passed to a higher-order function

RC : call of the function returned by a higher-order function

Our Contributions

- Higher-Order Trace (HOT) that captures the control flow of higher-order programs
- Higher-Order Trace Propositional Dynamic Logic (HOT-PDL):
  - Propositional Dynamic Logic (PDL) over HOTs for specifying temporal properties of higher-order programs
- Decidability of HOT-PDL model checking of higher-order programs via a reduction to higher-order model checking

  (See the paper for details)
- Applications (See the paper for details)
  - Modeling and extending dependent refinement types
  - Modeling and extending stack-based access control properties

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}, \rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \mathbf{ret}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \ldots | [\boldsymbol{\pi}]\boldsymbol{\phi} | \langle\boldsymbol{\pi}\rangle\boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow | \rightarrow_{\boldsymbol{ret}} | \rightarrow_{\boldsymbol{call}} | \boldsymbol{\pi_1} \cdot \boldsymbol{\pi_2} | \boldsymbol{\pi_1} + \boldsymbol{\pi_2} | \boldsymbol{\pi}^* | \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

Any of top-level functions f, bounded integers $n \in \mathbb{Z}_b$, or anonymous functions ●

$$\phi ::= \mathbf{call}(t_1, t_2) \mid \mathbf{ret}(t_1, t_2) \mid \ldots \mid [\pi]\phi \mid \langle \pi \rangle \phi$$

  - (Path expression)

$$\pi ::= \rightarrow \mid \rightarrow_{ret} \mid \rightarrow_{call} \mid \pi_1 \cdot \pi_2 \mid \pi_1 + \pi_2 \mid \pi^* \mid \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

A call event of a function t_1 with an argument t_2

$$\phi ::= \mathbf{call}(t_1, t_2) \mid \mathbf{ret}(t_1, t_2) \mid \ldots \mid [\pi]\phi \mid \langle\pi\rangle\phi$$

  - (Path expression)

$$\pi ::= \rightarrow \mid \rightarrow_{ret} \mid \rightarrow_{call} \mid \pi_1 \cdot \pi_2 \mid \pi_1 + \pi_2 \mid \pi^* \mid \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

A return event of a function t_1 with a return value t_2

$$\phi ::= \mathbf{call}(t_1, t_2) \mid \mathbf{ret}(t_1, t_2) \mid \ldots \mid [\pi]\phi \mid \langle\pi\rangle\phi$$

  - (Path expression)

$$\pi ::= \rightarrow \mid \rightarrow_{ret} \mid \rightarrow_{call} \mid \pi_1 \cdot \pi_2 \mid \pi_1 + \pi_2 \mid \pi^* \mid \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

Box: ϕ holds at every node reached by the path represented by π

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \mathbf{ret}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \ldots | [\boldsymbol{\pi}]\boldsymbol{\phi} | \langle\boldsymbol{\pi}\rangle\boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow | \rightarrow_{\boldsymbol{ret}} | \rightarrow_{\boldsymbol{call}} | \boldsymbol{\pi_1} \cdot \boldsymbol{\pi_2} | \boldsymbol{\pi_1} + \boldsymbol{\pi_2} | \boldsymbol{\pi}^* | \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

Diamond: there exists a node that can be reached by π and ϕ holds

$$\phi ::= \mathbf{call}(t_1, t_2) | \mathbf{ret}(t_1, t_2) | \ldots | [\pi]\phi | \langle\pi\rangle\phi$$

  - (Path expression)

$$\pi ::= \rightarrow | \rightarrow_{ret} | \rightarrow_{call} | \pi_1 \cdot \pi_2 | \pi_1 + \pi_2 | \pi^* | \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t_1}, \boldsymbol{t_2}) \mid \mathbf{ret}(\boldsymbol{t_1}, \boldsymbol{t_2}) \mid \ldots \mid [\boldsymbol{\pi}]\boldsymbol{\phi} \mid \langle\boldsymbol{\pi}\rangle\boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow \mid \rightarrow_{\boldsymbol{ret}} \mid \rightarrow_{\boldsymbol{call}} \mid \boldsymbol{\pi_1} \cdot \boldsymbol{\pi_2} \mid \boldsymbol{\pi_1} + \boldsymbol{\pi_2} \mid \boldsymbol{\pi}^* \mid \{\phi\}?$$

Concatenation

alternation

Kleene star

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}, \rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \mathbf{ret}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \ldots | [\boldsymbol{\pi}]\boldsymbol{\phi} | \langle\boldsymbol{\pi}\rangle\boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow | \rightarrow_{\boldsymbol{ret}} | \rightarrow_{\boldsymbol{call}} | \boldsymbol{\pi_1} \cdot \boldsymbol{\pi_2} | \boldsymbol{\pi_1} + \boldsymbol{\pi_2} | \boldsymbol{\pi}^* | \{\phi\}?$$

tests if ϕ holds at the current node

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \mathbf{ret}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \ldots | [\boldsymbol{\pi}]\boldsymbol{\phi} | \langle\boldsymbol{\pi}\rangle\boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow | \rightarrow_{\boldsymbol{ret}} | \rightarrow_{\boldsymbol{call}} | \boldsymbol{\pi_1} \cdot \boldsymbol{\pi_2} | \boldsymbol{\pi_1} + \boldsymbol{\pi_2} | \boldsymbol{\pi}^* | \{\phi\}?$$

move to the next event in the sequence

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}, \rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t}_1, \boldsymbol{t}_2) | \mathbf{ret}(\boldsymbol{t}_1, \boldsymbol{t}_2) | \ldots | [\boldsymbol{\pi}]\boldsymbol{\phi} | \langle \boldsymbol{\pi} \rangle \boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow | \rightarrow_{\boldsymbol{ret}} | \rightarrow_{\boldsymbol{call}} | \boldsymbol{\pi}_1 \cdot \boldsymbol{\pi}_2 | \boldsymbol{\pi}_1 + \boldsymbol{\pi}_2 | \boldsymbol{\pi}^* | \{\phi\}?$$

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}, \rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(t_1, t_2) | \mathbf{ret}(t_1, t_2) | \ldots | [\pi]\phi | \langle\pi\rangle\phi$$

  - (Path expression)

$$\pi ::= \rightarrow | \rightarrow_{ret} | \rightarrow_{call} | \pi_1 \cdot \pi_2 | \pi_1 + \pi_2 | \pi^* | \{\phi\}?$$

traverse an edge labeled with CR

Higher-Order Propositional Dynamic Logic (HOT-PDL)

- To traverse the pointers, HOT-PDL extends PDL with new path expressions ($\rightarrow_{ret}$, $\rightarrow_{call}$)
- Each node of a HOT is assigned a truth value indicating whether the given formula is satisfied
- Syntax:
  - (Formulas)

$$\boldsymbol{\phi} ::= \mathbf{call}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \mathbf{ret}(\boldsymbol{t_1}, \boldsymbol{t_2}) | \ldots | [\boldsymbol{\pi}]\boldsymbol{\phi} | \langle\boldsymbol{\pi}\rangle\boldsymbol{\phi}$$

  - (Path expression)

$$\boldsymbol{\pi} ::= \rightarrow | \rightarrow_{\boldsymbol{ret}} | \rightarrow_{\boldsymbol{call}} | \boldsymbol{\pi_1} \cdot \boldsymbol{\pi_2} | \boldsymbol{\pi_1} + \boldsymbol{\pi_2} | \boldsymbol{\pi}^* | \{\phi\}?$$

traverse an edge labeled with CC or RC

Example : The property that cannot be expressed by the previous specification languages

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$[\rightarrow^*] \bigwedge x \in \mathbb{Z}_b \left(\left(\mathbf{call}(\mathbf{twice},\cdot) \wedge \langle \rightarrow_{\mathbf{ret}} \cdot \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right) \Rightarrow \langle \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right)$$

Example : The property that cannot be expressed by the previous specification languages

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$[\to^*] \bigwedge x \in \mathbb{Z}_b \left(\begin{array}{r} \left(\mathbf{call}(\mathbf{twice},\cdot) \wedge \langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x)\right) \\ \Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \end{array} \right)$$

$\mathbb{Z}_b$: bounded integers

Example : The property that cannot be expressed by the previous specification languages

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$[\to^*] \bigwedge x \in \mathbb{Z}_b \left(\begin{array}{r} \big(\mathbf{call}(\mathbf{twice}, \cdot) \wedge \langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x)\big) \\ \Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \end{array} \right)$$

$[\to *]\phi = \mathrm{G}\phi$
(in LTL)

$\mathbb{Z}_b$: bounded integers

Example : The property that cannot be expressed by the previous specification languages

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$\boxed{[\rightarrow^*] \bigwedge x \in \mathbb{Z}_b} \left(\left(\mathbf{call}(\mathbf{twice}, \cdot) \wedge \langle \rightarrow_{\mathbf{ret}} \cdot \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right) \Rightarrow \langle \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right)$$

Globally, for all bounded integer x,

Example : The property that cannot be expressed by the previous specification languages

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$[\to^*] \bigwedge x \in \mathbb{Z}_b \left(\boxed{(\mathbf{call}(\mathbf{twice},\cdot)} \land \langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, \boldsymbol{x})) \Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, \boldsymbol{x}) \right)$$

If the function twice is called

Example : The property that cannot be expressed by the previous specification languages

If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$[\rightarrow^*] \bigwedge_{x \in \mathbb{Z}_b} \left(\left(\mathbf{call}(\mathbf{twice}, \cdot) \boxed{\wedge \langle \rightarrow_{\mathbf{ret}} \cdot \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x))} \Rightarrow \langle \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right) \right)$$

and the function returned by twice is called with an argument x,

Example : The property that cannot be expressed by the previous specification languages

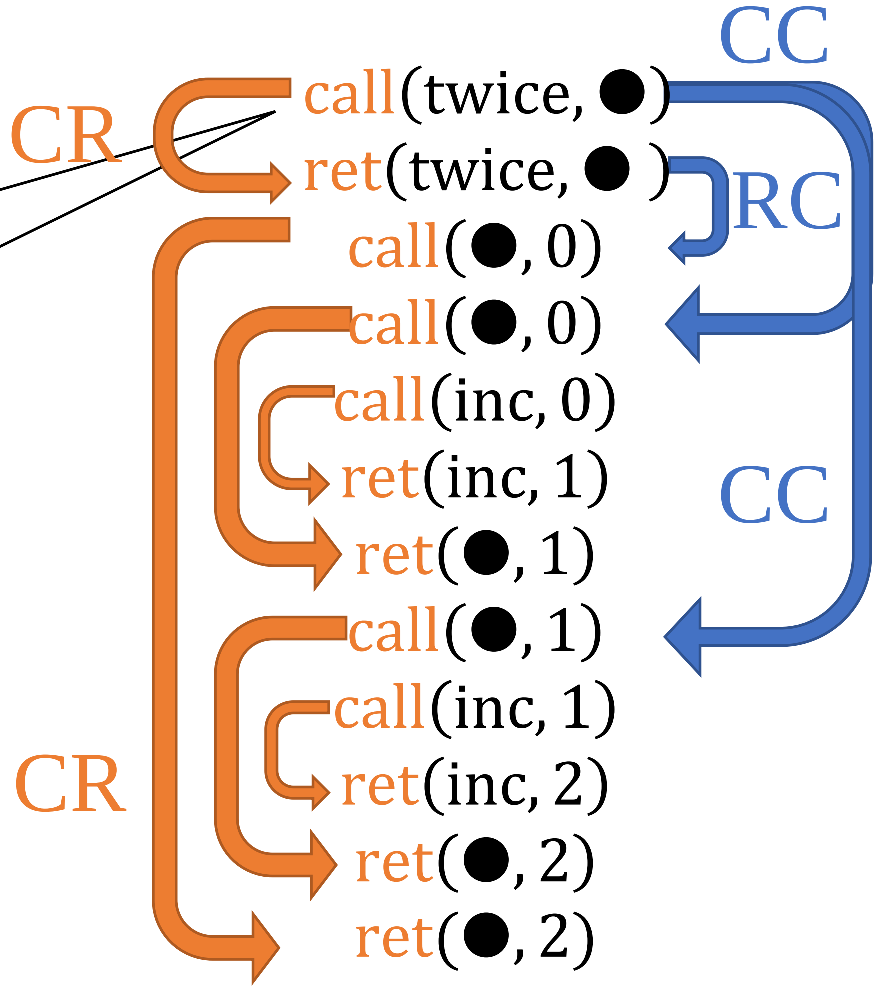
If the function returned by a partial application of twice to some function is called with some integer n, then the function argument passed to twice is eventually called with n

$$[\to^*] \bigwedge x \in \mathbb{Z}_b \left(\left(\mathbf{call}(\mathbf{twice}, \cdot) \wedge \langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right) \boxed{\Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x)} \right)$$

then the function argument passed to twice is eventually called with the argument x

Example : The property that cannot be expressed by the previous specification languages

$$[\to^*] \bigwedge_{x \in \mathbb{Z}_b} \left(\left(\mathbf{call}(\mathbf{twice},\cdot) \wedge \langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right) \Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right)$$

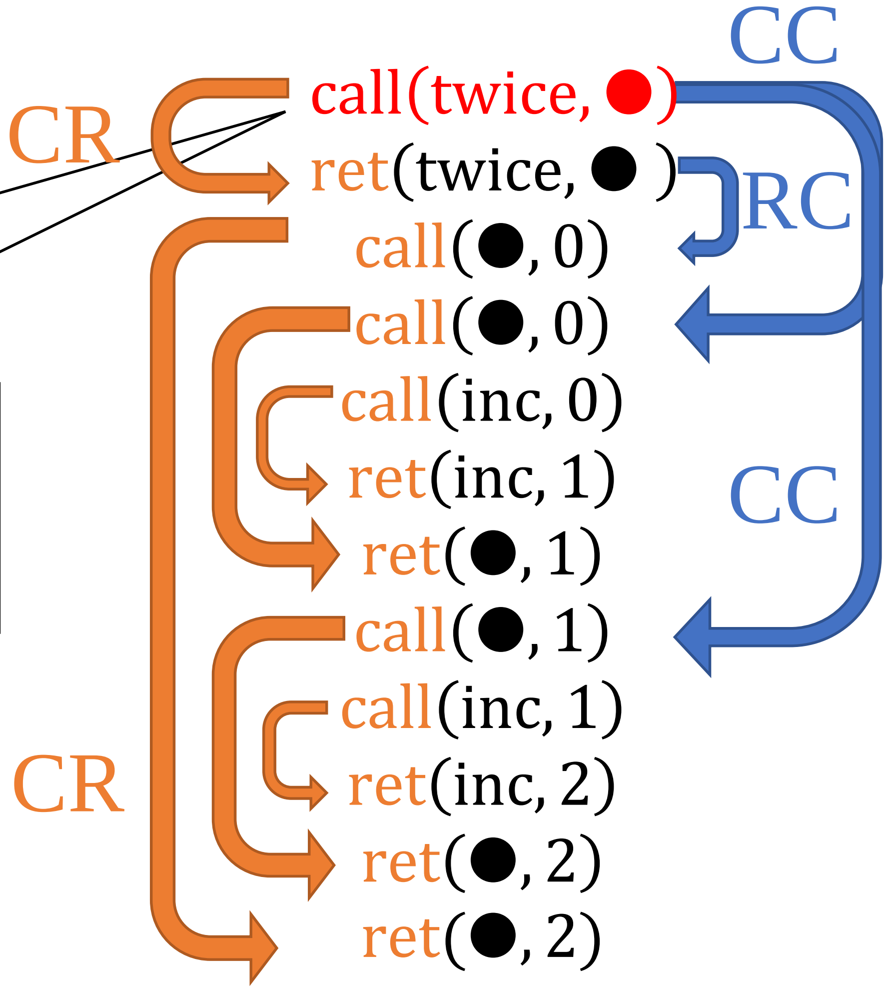


The formula holds at the node labeled with the event call(twice, ●)

Example : The property that cannot be expressed by the previous specification languages

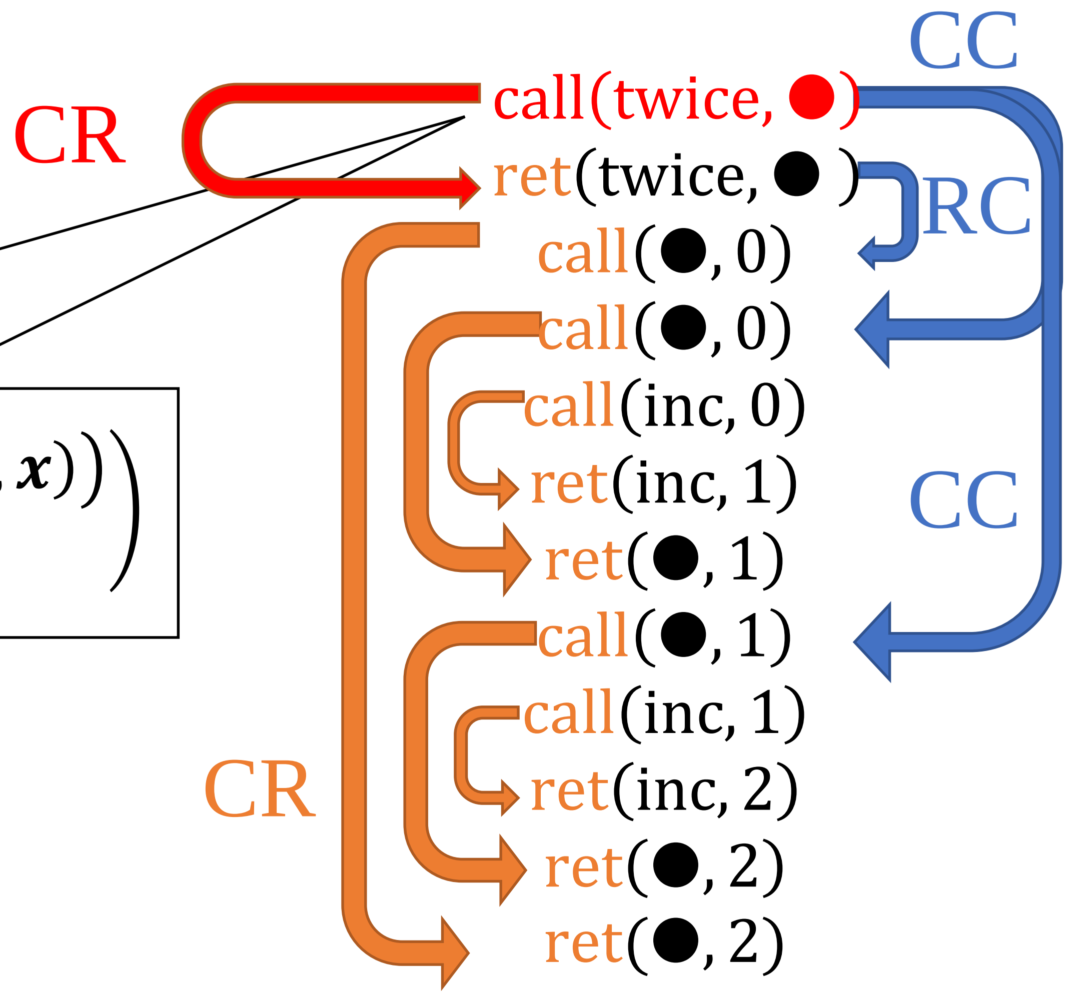
$$[\to^*]\bigwedge_{x\in\mathbb{Z}_b}\Big(\big(\boxed{\mathbf{call}(\mathbf{twice},\cdot)}\wedge\langle\to_{\mathbf{ret}}\cdot\to_{\mathbf{call}}\rangle\mathbf{call}(\cdot,x)\big)\Rightarrow\langle\to_{\mathbf{call}}\rangle\mathbf{call}(\cdot,x)\Big)$$

The formula holds at the node labeled with the event call(twice, ●)



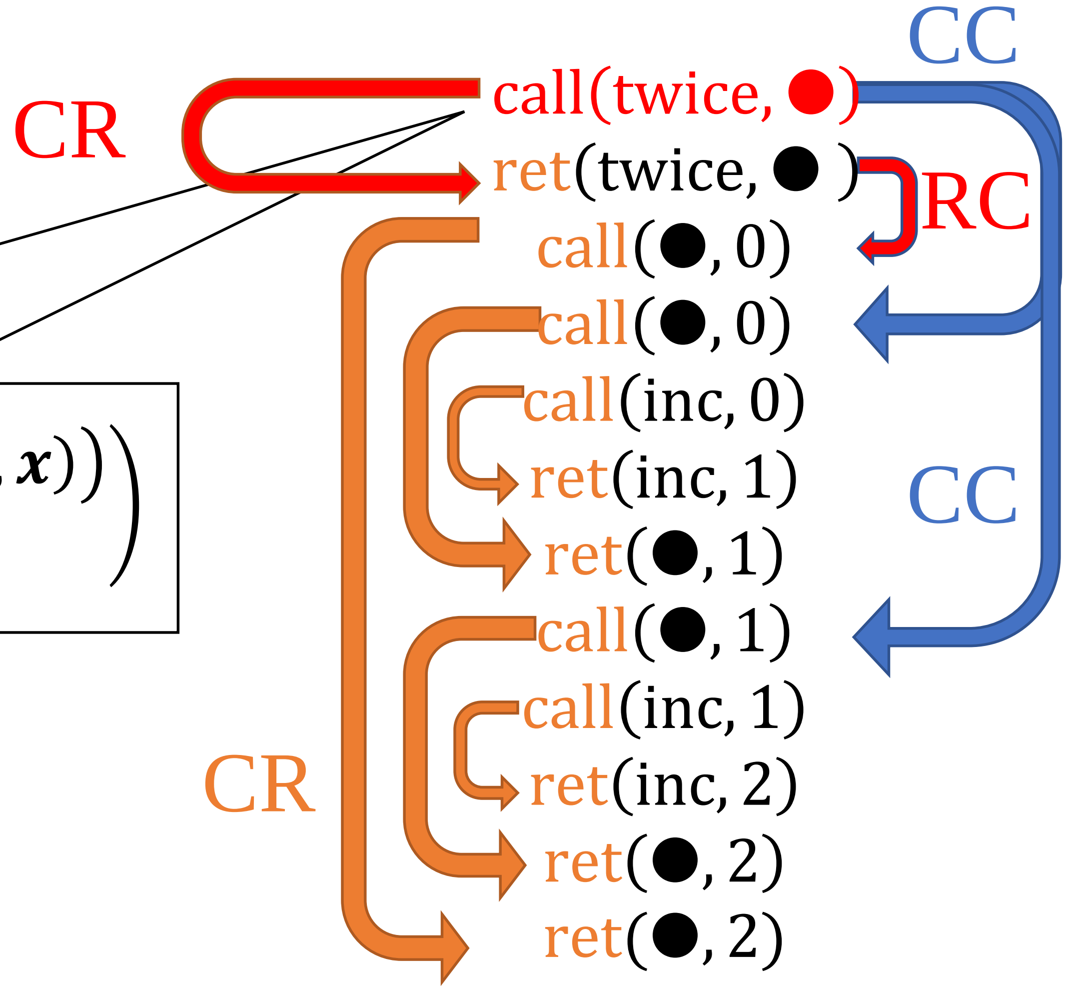
Example : The property that cannot be expressed by the previous specification languages

$$[\rightarrow^*] \bigwedge_{x \in \mathbb{Z}_b} \left(\left(\mathbf{call}(\mathbf{twice},\cdot) \wedge \langle \boxed{\rightarrow_{\mathbf{ret}}} \cdot \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right) \Rightarrow \langle \rightarrow_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right)$$



The formula holds at the node labeled with the event call(twice, ●)

Example : The property that cannot be expressed by the previous specification languages



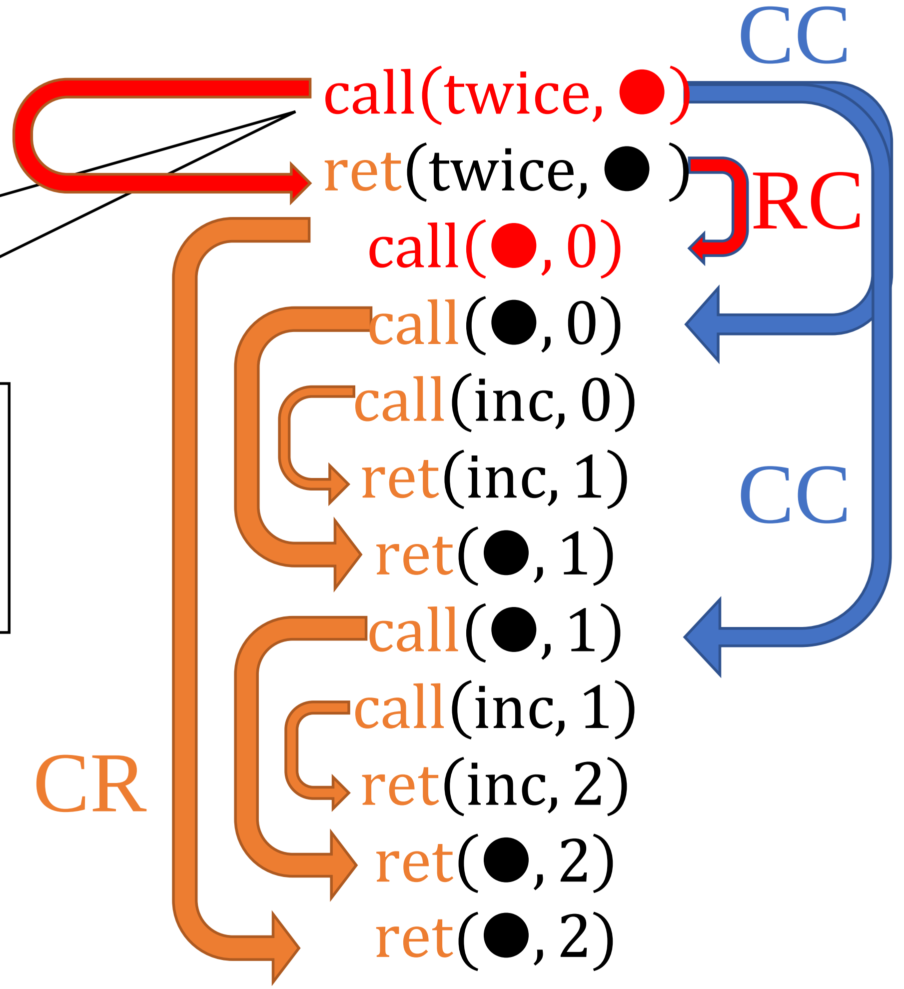
$$[\to^*] \bigwedge_{x \in \mathbb{Z}_b} \left(\left(\mathbf{call}(\mathbf{twice},\cdot) \wedge \boxed{\langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}}}\rangle \mathbf{call}(\cdot, x)\right) \Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x) \right)$$

The formula holds at the node labeled with the event call(twice, ●)

Example : The property that cannot be expressed by the previous specification languages

$x = 0$

$$[\to^*] \bigwedge_{x \in \mathbb{Z}_b} \left(\left(\mathbf{call}(\mathbf{twice},\cdot) \land \boxed{\langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, \boldsymbol{x})} \right) \Rightarrow \langle \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, \boldsymbol{x}) \right)$$



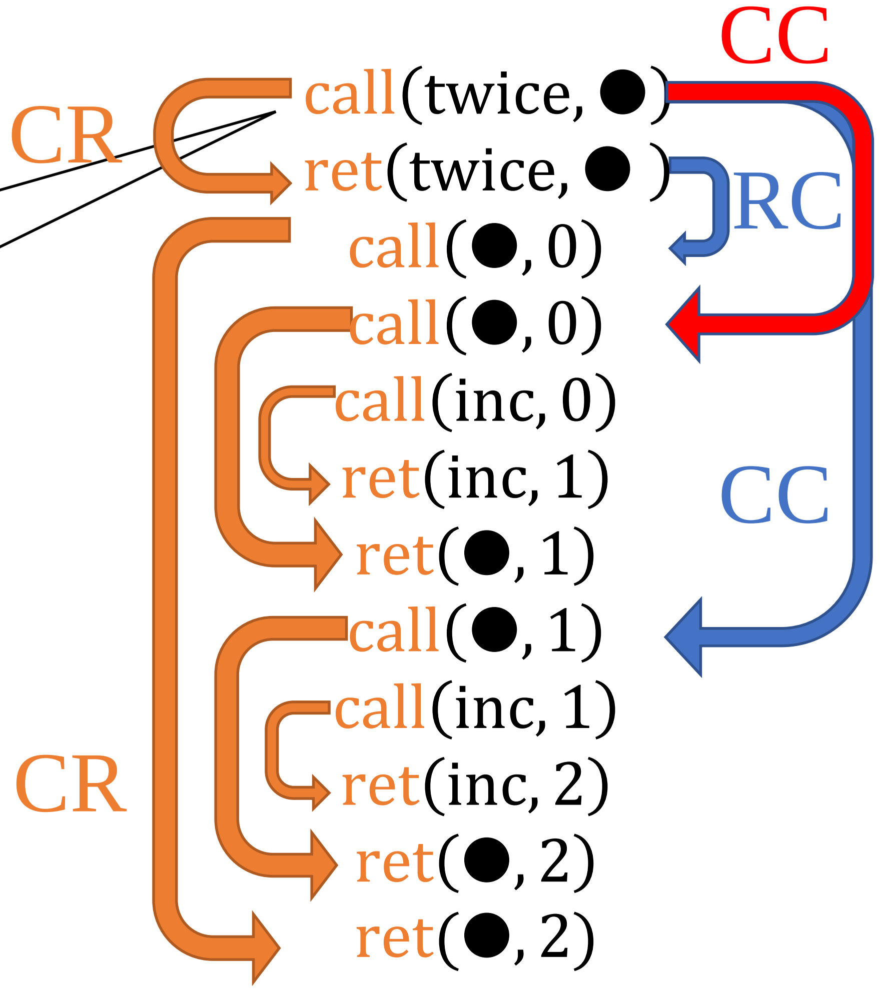
The formula holds at the node labeled with the event call(twice, ●)

Example : The property that cannot be expressed by the previous specification languages

$x = 0$

$$[\to^*] \bigwedge_{x \in \mathbb{Z}_b} \Big(\big(\mathbf{call}(\mathbf{twice}, \cdot) \wedge \langle \to_{\mathbf{ret}} \cdot \to_{\mathbf{call}} \rangle \mathbf{call}(\cdot, x)\big) \boxed{\Rightarrow \langle \to_{\mathbf{call}}} \rangle \mathbf{call}(\cdot, x) \Big)$$

The formula holds at the node labeled with the event call(twice, ●)



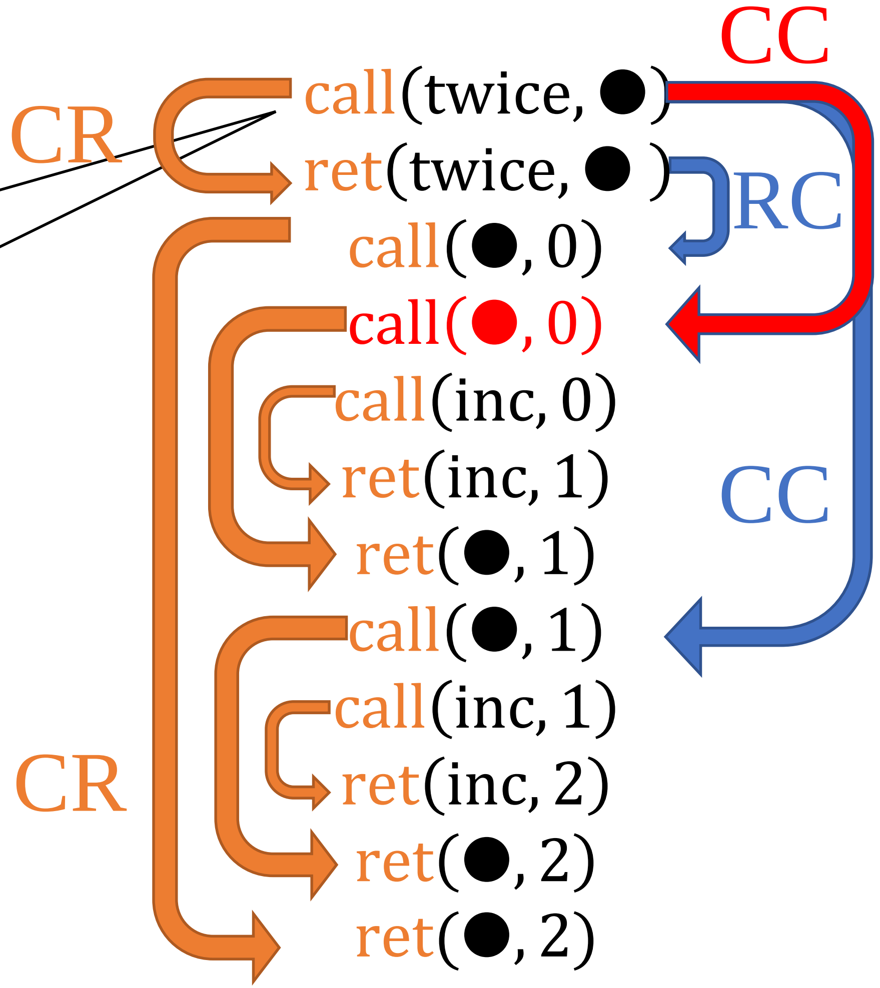
Example : The property that cannot be expressed by the previous specification languages

$x = 0$

$$[\to^*]\bigwedge_{x\in\mathbb{Z}_b}\Big(\big(\mathbf{call}(\mathbf{twice},\cdot)\wedge\langle\to_{\mathbf{ret}}\cdot\to_{\mathbf{call}}\rangle\mathbf{call}(\cdot,x)\big)\ \boxed{\Rightarrow\langle\to_{\mathbf{call}}\rangle\mathbf{call}(\cdot,x)}\Big)$$

$x = 0$

The formula holds at the node labeled with the event call(twice, ●)



Conclusion

- HOT captures the control flow of higher-order programs
- HOT-PDL is an extension of PDL defined over HOTs
  - Enables a precise specification of temporal trace properties for higher-order programs
  - Provides a foundation for specification in various application domains
    - stack-based access control properties
    - dependent refinement types
- HOT-PDL model checking of higher-order programs is shown decidable via a reduction to higher-order model checking
- **Future work:** extend HOTs with new kinds of events and pointers for capturing call-by-name and/or effectful computations by incorporating more ideas from game semantics