Propositional Dynamic Logic for Higher-Order Functional Programs

Yuki Satake and Hiroshi Unno

University of Tsukuba
{satake,uhiro}@logic.cs.tsukuba.ac.jp

Abstract. We present an extension of propositional dynamic logic called HOT-PDL for specifying temporal properties of higher-order functional programs. The semantics of HOT-PDL is defined over Higher-Order Traces (HOTs) that model execution traces of higher-order programs. A HOT is a sequence of events such as function calls and returns, equipped with two kinds of pointers inspired by the notion of justification pointers from game semantics: one for capturing the correspondence between call and return events, and the other for capturing higher-order control flow involving a function that is passed to or returned by a higher-order function. To allow traversal of the new kinds of pointers, HOT-PDL extends PDL with new path expressions. The extension enables HOT-PDL to specify interesting properties of higher-order programs, including stack-based access control properties and those definable using dependent refinement types. We show that HOT-PDL model checking of higher-order functional programs over bounded integers is decidable via a reduction to modal μ-calculus model checking of higher-order recursion schemes.

1 Introduction

Temporal verification of higher-order programs has been an emerging research topic [12, 14, 18, 22–24, 26, 27, 31, 34]. The specification languages used there are (ω-)regular word languages (that subsume LTL) [12, 18, 26] and modal μ-calculus (that subsumes CTL) [14, 24, 31], which are interpreted over sequences or trees consisting of events. (Extended) dependent refinement types are also used to specify temporal [23, 27] and branching properties [34]. These specification languages, however, cannot sufficiently express specifications of control flow involving (higher-order) functions. For example, let us consider the following simple higher-order program $D_{\texttt{tw}}$ (in OCaml syntax):

```
let tw f x = f (f x) in let inc x = x + 1 in let r = * in tw inc r
```

Here, $*$ denotes a non-deterministic integer, and the higher-order function $\texttt{tw} : (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$ applies its function argument $\texttt{f} : \texttt{int} \to \texttt{int}$ to the integer argument $\texttt{x}$ twice. For example, for $\texttt{r} = 0$, the program $D_{\texttt{tw}}$ exhibits the following call-by-value reduction sequence (with the redexes underlined).

$$\underline{\texttt{tw inc}}\ 0 \longrightarrow \underline{(\lambda x.\texttt{inc}\ (\texttt{inc}\ x))\ 0} \longrightarrow \texttt{inc}\ (\underline{\texttt{inc}\ 0}) \longrightarrow^* \underline{\texttt{inc}\ 1} \longrightarrow^* 2$$

Example properties of the program $D_{\texttt{tw}}$ that cannot be expressed by the previous specification languages are:

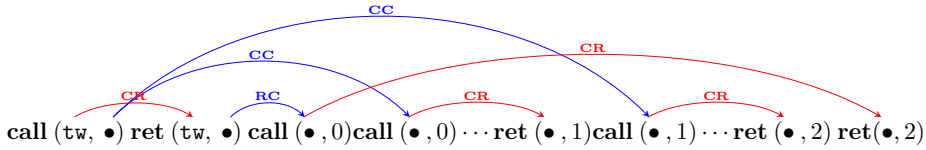
Prop.1 If the function returned by a partial application of `tw` to some function (e.g., $\lambda x.\texttt{inc}\ (\texttt{inc}\ x)$ in the above sequence) is called with some integer n, the function argument passed to `tw` (i.e., `inc`) is eventually called with n.

Prop.2 If the function returned by a partial application of `tw` to some function is never called, then the function argument passed to `tw` is never called.

To remedy the limitation, we introduce a notion of Higher-Order Trace (HOT) that captures the control flow of higher-order programs and propose a dynamic logic over HOTs called Higher-Order Trace Propositional Dynamic Logic (HOT-PDL) for specifying temporal properties of higher-order programs.

Intuitively, a HOT models a program execution trace which is a possibly infinite sequence of events such as function calls and returns with information about actual arguments and return values. Furthermore, HOTs are equipped with two kinds of pointers to enable precise specification of control flow: one for capturing the correspondence between call and return events, and the other for capturing higher-order control flow involving a function that is passed to or returned by a higher-order function. The two kinds of pointers are inspired by the notion of justification pointers from the game semantics of PCF [1, 2, 19, 20].

For the higher-order program $D_{\texttt{tw}}$, for $\texttt{r} = 0$, we get the following HOT $G_{\texttt{tw}}$:[1]



Here, $\bullet$ represents some function value, $\mathbf{call}(f, v)$ represents a call event of the function f with the argument v, and $\mathbf{ret}(f, v)$ represents a return event of the function f with the return value v. This trace corresponds to the previous reduction sequence: the call events $\mathbf{call}(\texttt{tw}, \bullet)$, $\mathbf{call}(\bullet, 0)$, $\mathbf{call}(\bullet, 0)$, and $\mathbf{call}(\bullet, 1)$ that occur in the trace in this order correspond respectively to the redexes $\texttt{tw}\ \texttt{inc}$, $(\lambda x.\texttt{inc}\ (\texttt{inc}\ x))\ 0$, $\texttt{inc}\ 0$, and $\texttt{inc}\ 1$. The three important points here are that (1) the call events have pointers labeled with **CR** to the corresponding return events $\mathbf{ret}(\texttt{tw}, \bullet)$, $\mathbf{ret}(\bullet, 2)$, $\mathbf{ret}(\bullet, 1)$, and $\mathbf{ret}(\bullet, 2)$, (2) the call event $\mathbf{call}(\texttt{tw}, \bullet)$ has two pointers labeled with **CC**, where $\bullet$ represents the function argument `f` of `tw` and the pointed call events $\mathbf{call}(\bullet, 0)$ and $\mathbf{call}(\bullet, 1)$ represent the two calls to `f` in `tw`, and (3) the return event $\mathbf{ret}(\texttt{tw}, \bullet)$ has a pointer labeled with **RC**, where $\bullet$ represents the partially-applied function $\lambda x.\texttt{inc}\ (\texttt{inc}\ x)$ and the pointed call event $\mathbf{call}(\bullet, 0)$ represents the call to the function.

To allow traversal of the pointers, HOT-PDL extends propositional dynamic logic with new path expressions (see Section 3 for details). The extension enables

[1] The symbol $\cdots$ indicates the omission of a subsequence. The two omitted subsequences are $\mathbf{call}(\texttt{inc}, 0)\ \mathbf{ret}(\texttt{inc}, 1)$ and $\mathbf{call}(\texttt{inc}, 1)\ \mathbf{ret}(\texttt{inc}, 2)$ in this order.

HOT-PDL to specify interesting properties of higher-order programs, including stack-based access control properties and those definable using dependent refinement types. Here, stack-based access control is a security mechanism implemented in runtimes like JVM for ensuring secure execution of programs that have components with different levels of trust: the mechanism ensures that a *security-critical* function (e.g., file access) is invoked only if all the (immediate and indirect) callers in the current call stack are *trusted*, or one of the callers is a *privileged* function and its callees are all *trusted*. We introduce a new variant of stack-based access control properties for higher-order programs, formalized in HOT-PDL from the point of view of interactions among callers and callees.

Compared to the previous specification languages with respect to the expressiveness, HOT-PDL subsumes (ω-)regular languages because PDL interpreted over words is already as expressive as them [15]. Temporal logics over nested words [6] such as CaRet [5] and NWTL [4] can capture the correspondence between call and return events (i.e., pointers labeled with **CR**) but cannot capture higher-order control flow (i.e., pointers labeled with **CC** and **RC**). Branching properties (expressible in, e.g., CTL), however, are out of the scope of the present paper, and such an extension of HOT-PDL remains an interesting future direction. Dependent refinement types are often used to specify properties of higher-order programs for partial- and total-correctness verification [29, 33, 39, 40]. For example, the following properties of the program $D_{\texttt{tw}}$ are expressible:

Prop.3 The function yielded by applying `tw` to a strictly increasing function is strictly increasing.

Prop.4 The function yielded by applying `tw` to a terminating function is terminating.

This paper shows that HOT-PDL can encode such dependent refinement types.

We also study HOT-PDL model checking: given a higher-order program D over bounded integers and a HOT-PDL formula ϕ, the problem is to decide whether ϕ is satisfied by all the execution traces of D modeled as HOTs. We show the decidability of HOT-PDL model checking via a reduction to modal μ-calculus model checking of higher-order recursion schemes [21, 28].

The rest of the paper is organized as follows. Section 2 formalizes HOTs and explains how to use them to model execution traces of higher-order functional programs. Section 3 defines the syntax and the semantics of HOT-PDL and Section 4 shows how to encode stack-based access control properties and dependent refinement types in HOT-PDL. Section 5 discusses HOT-PDL model checking. We compare HOT-PDL with related work in Section 6 and conclude the paper with remarks on future work in Section 7. Omitted proofs are given in the extended version of this paper [30].

2 Higher-Order Traces

This section defines the notion of Higher-Order Trace (HOT), which is used to model execution traces of higher-order programs. To this end, we first define (Σ, Γ)-*labeled directed graphs* and *DAGs*.

Definition 1 (**(Σ, Γ)-labeled directed graphs**). *Let Σ be a finite set of node labels and Γ be a finite set of edge labels. A* (Σ, Γ)-labeled directed graph *is defined as a triple (V, λ, ν), where V is a countable set of nodes, $\lambda : V \to \Sigma$ is a node labeling function, and $\nu : V \times V \to 2^{\Gamma}$ is an edge labeling function. We call a (Σ, Γ)-labeled directed graph that has no directed cycle (Σ, Γ)-labeled DAG.*

Note that an edge may have multiple labels. For nodes $u, u' \in V$, $\nu(u, u') = \emptyset$ means that there is no edge from u to u'. We use σ and γ as meta-variables ranging respectively over Σ and Γ. We write V_σ for the set $\{u \in V \mid \sigma = \lambda(u)\}$ of all the nodes labeled with σ. We also write V_Σ for the set $\bigcup_{\sigma \in \Sigma} V_\sigma$. For $u, u' \in V$, we write $u \prec_\gamma u'$ if $\gamma \in \nu(u, u')$. A binary relation $\prec_\gamma^+$ (resp. $\prec_\gamma^*$) denotes the transitive (resp. reflexive and transitive) closure of $\prec_\gamma$.

Definition 2 (**HOTs**). *A HOT is a (Σ, Γ)-DAG, $G = (V, \lambda, \nu)$ that satisfies:*

1. *$V \neq \emptyset$, $\Gamma = \{\mathbf{N}, \mathbf{CR}, \mathbf{CC}, \mathbf{RC}\}$, $\Sigma = \Sigma_{\mathbf{call}} \uplus \Sigma_{\mathbf{ret}}$, and $\Sigma_{\mathbf{call}} = \Sigma^T_{\mathbf{call}} \uplus \Sigma^A_{\mathbf{call}}$*
2. *$\prec_{\mathbf{CR}} \subseteq (V_{\Sigma_{\mathbf{call}}} \times V_{\Sigma_{\mathbf{ret}}})$, $\prec_{\mathbf{CC}} \subseteq (V_{\Sigma_{\mathbf{call}}} \times V_{\Sigma^A_{\mathbf{call}}})$, and $\prec_{\mathbf{RC}} \subseteq (V_{\Sigma_{\mathbf{ret}}} \times V_{\Sigma^A_{\mathbf{call}}})$.*
3. *The elements of V are linearly ordered by $\prec_{\mathbf{N}}$*
4. *If $u \prec_{\mathbf{CR}} u'$ and $u \prec_{\mathbf{CR}} u''$, then $u' = u''$.*
5. *For all $u' \in V_{\Sigma_{\mathbf{ret}}}$, there uniquely exists $u \in V_{\Sigma_{\mathbf{call}}}$ such that $u \prec_{\mathbf{CR}} u'$ holds.*
6. *For all $u' \in V_{\Sigma^A_{\mathbf{call}}}$, there uniquely exists $u \in V$ such that $u \prec_{\mathbf{CC}} u'$ or $u \prec_{\mathbf{RC}} u'$ holds.*

Intuitively, $\Sigma_{\mathbf{call}}$ (resp. $\Sigma_{\mathbf{ret}}$) represents a set of call (resp. return) events. $\Sigma^T_{\mathbf{call}}$ (resp. $\Sigma^A_{\mathbf{call}}$) represents a set of call events of top-level functions (resp. functions that are returned by or passed to (higher-order) functions). $u \prec_{\mathbf{N}} u'$ means that u' is the next event of u in the trace. $u \prec_{\mathbf{CR}} u'$ indicates that u' is the return event corresponding to the call event u. $u \prec_{\mathbf{CC}} u'$ represents that u' is a call event of the function argument passed at the call event u. $u \prec_{\mathbf{RC}} u'$ means that u' is a call event of the partially-applied function returned at the return event u. We call the minimum node of a HOT G with respect to $\prec_{\mathbf{N}}$ the *root node*, denoted by 0_G. For HOTs G_1 and G_2, we say G_1 is a *prefix* of G_2 and write $G_1 \preceq G_2$, if G_1 is a sub-graph of G_2 such that $0_{G_1} = 0_{G_2}$. Note that the HOT $G_{\mathtt{tw}}$ in Section 1, where $\mathbf{N}$-labeled edges are omitted, satisfies the above conditions, with $\{\mathbf{call}(\mathtt{tw}, \bullet), \mathbf{call}(\mathtt{inc}, 0), \mathbf{call}(\mathtt{inc}, 1)\} \subseteq \Sigma^T_{\mathbf{call}}$, $\{\mathbf{call}(\bullet, 0), \mathbf{call}(\bullet, 1)\} \subseteq \Sigma^A_{\mathbf{call}}$, and $\{\mathbf{ret}(\mathtt{tw}, \bullet), \mathbf{ret}(\mathtt{inc}, 1), \mathbf{ret}(\mathtt{inc}, 2), \mathbf{ret}(\bullet, 1), \mathbf{ret}(\bullet, 2)\} \subseteq \Sigma_{\mathbf{ret}}$.

2.1 Trace Semantics for Higher-Order Functional Programs

We now formalize our target language $\mathcal{L}$, which is an ML-like typed call-by-value higher-order functional language. The syntax is defined by

$$
\begin{array}{rl}
\text{(programs)} & D ::= \{f_1 \mapsto \lambda x.e_1, \ldots, f_m \mapsto \lambda x.e_m\} \\
\text{(expressions)} & e ::= x \mid f \mid \lambda x.e \mid e_1\ e_2 \mid n \mid \mathtt{op}(e_1, e_2) \mid \mathtt{ifz}\ e_1\ e_2\ e_3 \\
\text{(values)} & v ::= f \mid \lambda x.e \mid n \\
\text{(types)} & \tau ::= \mathtt{int} \mid \tau_1 \to \tau_2
\end{array}
$$

Here, x and f are meta-variables ranging respectively over term variables and names of top-level functions. The meta-variable n ranges over the set of bounded

Domains

$$
\begin{array}{rl}
\text{(configurations)} & C ::= (I, E[e]) \\
\text{(eval. contexts)} & E ::= [\,] \mid E\ e \mid v\ E \mid \mathtt{op}(E, e) \mid \mathtt{op}(v, E) \mid \mathtt{ifz}\ E\ e_1\ e_2 \mid \mathtt{ret}(h, i, E) \\
\text{(interfaces)} & I ::= \left\{ h_1 \overset{i_1}{\mapsto} v_1, \ldots, h_m \overset{i_m}{\mapsto} v_m \right\} \\
\text{(handles)} & h ::= n \mid f \mid \lfloor h \rfloor_i \mid \lceil h \rceil_i \\
\text{(events)} & \alpha ::= \mathbf{call}(h_1, i, h_2) \mid \mathbf{ret}(h_1, i, h_2)
\end{array}
$$

Derivation Rules

$$(I, E[(\lambda x.e)\ v]) \xrightarrow{\epsilon} (I, E[[v/x]e]) \quad (\textsc{App})$$

$$\frac{n = [\![\mathtt{op}]\!](n_1, n_2)}{(I, E[\mathtt{op}(n_1, n_2)]) \xrightarrow{\epsilon} (I, E[n])} \quad (\textsc{Op})$$

$$(I, E[\mathtt{ifz}\ 0\ e_1\ e_2]) \xrightarrow{\epsilon} (I, E[e_1]) \quad (\textsc{IfZ})$$

$$\frac{n \neq 0}{(I, E[\mathtt{ifz}\ n\ e_1\ e_2]) \xrightarrow{\epsilon} (I, E[e_2])} \quad (\textsc{IfN})$$

$$C \overset{\epsilon}{\Rightarrow} C \quad (\textsc{Refl})$$

$$\frac{C \xrightarrow{\varpi_1} C'' \qquad C'' \overset{\varpi_2}{\Rightarrow} C'}{C \overset{\varpi_1 \cdot \varpi_2}{\Longrightarrow} C'} \quad (\textsc{Tran})$$

$$\frac{C \xrightarrow{\varpi} C' \qquad C' \overset{\pi}{\Rightarrow} \bot}{C \overset{\varpi \cdot \pi}{\Longrightarrow} \bot} \quad (\textsc{Tran}\omega)$$

$$\frac{(h \overset{i}{\mapsto} v) \in I \qquad \alpha = \mathbf{call}(h, i, n) \qquad I' = I\left\{ h \overset{i+1}{\mapsto} v \right\}}{(I, E[h\ n]) \xrightarrow{\alpha} (I', E[\mathtt{ret}(h, i, v\ n)])} \quad (\textsc{CInt})$$

$$\frac{v \text{ is a function} \qquad (h \overset{i}{\mapsto} v') \in I \qquad \alpha = \mathbf{call}(h, i, \lfloor h \rfloor_i) \qquad I' = I\left\{ h \overset{i+1}{\mapsto} v', \lfloor h \rfloor_i \overset{0}{\mapsto} v \right\}}{(I, E[h\ v]) \xrightarrow{\alpha} (I', E[\mathtt{ret}(h, i, v'\ \lfloor h \rfloor_i)])} \quad (\textsc{CFun})$$

$$\frac{\alpha = \mathbf{ret}(h, i, n)}{(I, E[\mathtt{ret}(h, i, n)]) \xrightarrow{\alpha} (I, E[n])} \quad (\textsc{RInt})$$

$$\frac{v \text{ is a function} \qquad \alpha = \mathbf{ret}(h, i, \lceil h \rceil_i) \qquad I' = I\left\{ \lceil h \rceil_i \overset{0}{\mapsto} v \right\}}{(I, E[\mathtt{ret}(h, i, v)]) \xrightarrow{\alpha} (I', E[\lceil h \rceil_i])} \quad (\textsc{RFun})$$

Fig. 1. Labeled Transition Relations ($\overset{\varpi}{\Longrightarrow}$) and ($\overset{\pi}{\Rightarrow}$) for $\mathcal{L}$

integers $\mathbb{Z}_b = \{n_{\mathbf{min}}, \cdots, n_{\mathbf{max}}\} \subset \mathbb{Z}$. For simplicity of presentation, $\mathcal{L}$ has the type `int` of bounded integers as the only base type. `op` represents binary operators such as $+$, $-$, $\times$, $=$, and $>$. The binary relations $=$ and $>$ return an integer that encodes a boolean value (e.g., 1 for `true` and 0 for `false`). A program D maps each top-level function name f_i to its definition $\lambda x.e_i$. We write $\mathrm{dom}(D)$ for $\{f_1, \ldots, f_m\}$. We assume that D has the main function `main` of the type `int` $\rightarrow$ `int`. The functions in D can be mutually recursive. Expressions e comprise variables x, function names f, lambda abstractions $\lambda x.e$, function applications $e_1\ e_2$, bounded integers n, binary operations $\mathtt{op}(v_1, v_2)$, and conditional branches $\mathtt{ifz}\ e_1\ e_2\ e_3$. We assume that expressions are simply-typed. As usual, the simple type system guarantees that an evaluation of a typed expression never causes a runtime type mismatch like $1 + \lambda x.x$. An expression $\mathtt{ifz}\ e_1\ e_2\ e_3$ evaluates to e_2 (resp. e_3) if e_1 evaluates to 0 (resp. a non-zero integer). For example, the program $D_{\mathtt{tw}}$ in Section 1 is defined in $\mathcal{L}$ as follows:

$$D_{\mathtt{tw}} \triangleq \{\mathtt{tw} \mapsto \lambda f.\lambda x.f\ (f\ x), \mathtt{inc} \mapsto \lambda x.x + 1, \mathtt{main} \mapsto \lambda r.\mathtt{tw}\ \mathtt{inc}\ r\}$$

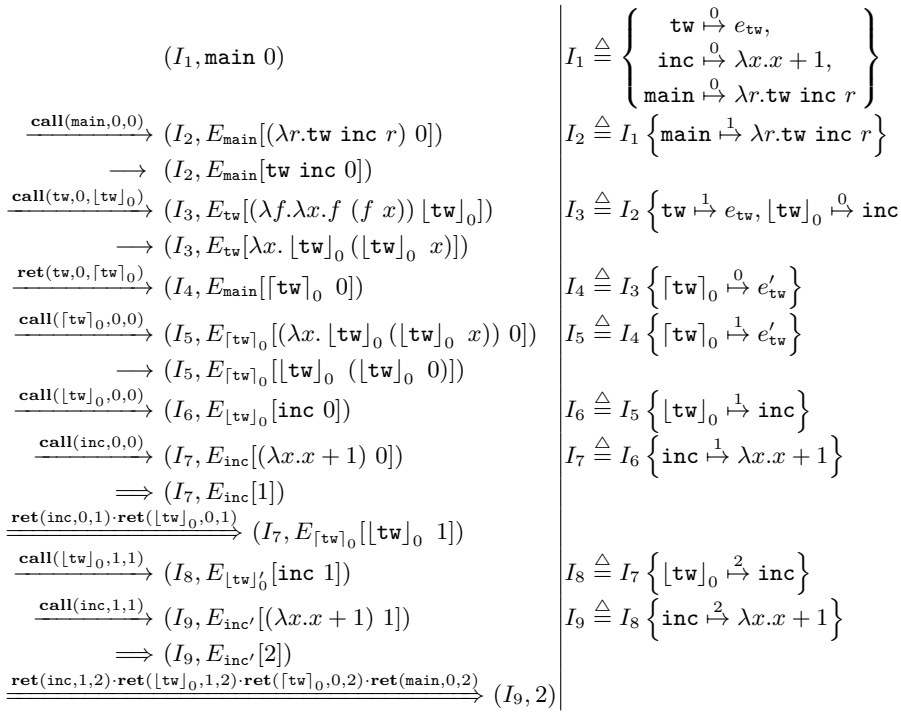


Fig. 2. Example Trace of $D_{\texttt{tw}}$

We now introduce a trace semantics of the language $\mathcal{L}$, which will be used in Section 5 to define our model checking problems of higher-order programs. In the trace semantics, a program execution trace is represented by a sequence of function call and return events without an explicit representation of pointers but with enough information to construct them. We will explain how to model traces of $\mathcal{L}$ as HOTs by presenting a translation.

The trace semantics $[\![D]\!]$ of the language $\mathcal{L}$ is defined as $[\![D]\!]_{\texttt{fin}} \cup [\![D]\!]_{\texttt{inf}}$ where $[\![D]\!]_{\texttt{fin}} = \left\{ \varpi \;\middle|\; (I, \texttt{main}\ n) \overset{\varpi}{\Rightarrow} C \right\}$ and $[\![D]\!]_{\texttt{inf}} = \left\{ \pi \;\middle|\; (I, \texttt{main}\ n) \overset{\pi}{\Rightarrow} \bot \right\}$ are respectively the sets of *finite* and *infinite* execution traces obtained by evaluating $\texttt{main}\ n$ for some integer n using *trace-labeled* multi-step reduction relations $\overset{\varpi}{\Rightarrow}$ and $\overset{\pi}{\Rightarrow}$, which are presented in Figure 1, under the program $I = \left\{ f \overset{0}{\mapsto} v \middle| (f \mapsto v) \in D \right\}$ annotated with the number of calls to each function oc-

curred so far (i.e., initialized to 0). There, we use ϖ (resp. π) as a meta-variable ranging over finite sequences $\alpha_1 \cdots \alpha_m$ (resp. infinite sequences $\alpha_1 \cdot \alpha_2 \cdots$) of events α_i. We write ϵ for the empty sequence, $\varpi_1 \cdot \varpi_2$ for the concatenation of the sequences ϖ_1 and ϖ_2, and $|\varpi|$ for the length of ϖ. An *event* α is either of the form $\mathbf{call}(h_1, i, h_2)$ or $\mathbf{ret}(h_1, i, h_2)$, where a *handle* h represents a top-level function or a runtime value exchanged among functions. An event $\mathbf{call}(h_1, i, h_2)$ represents the $(i+1)^{\text{th}}$ call to the function h_1 with the argument h_2. On the other hand, an event $\mathbf{ret}(h_1, i, h_2)$ represents the return of the $(i+1)^{\text{th}}$ call to the function h_1 with the return value h_2. We thus equip call and return events of h_1 with the information about (1) the number i of the calls to h_1 occurred so far and (2) the runtime value h_2 passed to or returned by h_1, so that we can construct pointers (see Definition 3 for details). Note here that handles h are also equipped with meta-information necessary for constructing pointers. More specifically, h is any of the following: a bounded integer n, a top-level function name $f \in \mathrm{dom}(D)$, the special identifier $\lfloor h \rfloor_i$ for the function argument of the $(i+1)^{\text{th}}$ call to the higher-order function h, or the special identifier $\lceil h \rceil_i$ for the partially-applied function returned by the $(i+1)^{\text{th}}$ call to h. We thus use handles to track for each function value where it is constructed and how many times it is called. We shall assume that the syntax of expressions e and values v is also extended with handles h. As we have seen, the finite traces $[\![D]\!]_{\mathtt{fin}}$ of a program D are collected using the *terminating* trace-labeled multi-step reduction relation $\stackrel{\varpi}{\Rightarrow}$ on configurations. A *configuration* $(I, E[e])$ is a pair of an interface I and an expression $E[e]$ consisting of an evaluation context E and a sub-expression e under evaluation. A special evaluation context $\mathtt{ret}(h, i, E)$ represents the calling context of the $(i+1)^{\text{th}}$ call to h that waits for the return value computed by E. An *interface* I is defined to be $\left\{h_1 \stackrel{i_1}{\mapsto} v_1, \ldots, h_m \stackrel{i_m}{\mapsto} v_m\right\}$ that maps each function handle h_j to its definition v_j, where i_j records the number of calls to the function h_j occurred so far. In the derivation rules for $\stackrel{\varpi}{\longrightarrow}$, $[\![\mathtt{op}]\!]$ represents the integer function denoted by $\mathtt{op}$, and $I\left\{h \stackrel{i}{\mapsto} v\right\}$ represents the interface obtained from I by adding (or replacing existing assignment to h with) the assignment $h \stackrel{i}{\mapsto} v$. In the rule CInt (resp. RInt) for function calls (resp. returns) with an integer n, the reduction relation is labeled with $\mathbf{call}(h, i, n)$ (resp. $\mathbf{ret}(h, i, n)$). By contrast, in the rule CFun (resp. RFun) for function calls (resp. returns) with a function value v, the special identifier $\lfloor h \rfloor_i$ (resp. $\lceil h \rceil_i$) for v is used in the label $\mathbf{call}(h, i, \lfloor h \rfloor_i)$ (resp. $\mathbf{ret}(h, i, \lceil h \rceil_i)$) of the reduction relation, and v in the expression is replaced by the identifier. For example, as shown in Figure 2, the following finite trace $\varpi_{\mathtt{tw}}$ is generated from the program $D_{\mathtt{tw}}$:

$$\mathbf{call}(\mathtt{main}, 0, 0) \cdot \mathbf{call}(\mathtt{tw}, 0, \lfloor \mathtt{tw} \rfloor_0) \cdot \mathbf{ret}(\mathtt{tw}, 0, \lceil \mathtt{tw} \rceil_0) \cdot \mathbf{call}(\lceil \mathtt{tw} \rceil_0, 0, 0) \cdot$$
$$\mathbf{call}(\lfloor \mathtt{tw} \rfloor_0, 0, 0) \cdot \mathbf{call}(\mathtt{inc}, 0, 0) \cdot \mathbf{ret}(\mathtt{inc}, 0, 1) \cdot \mathbf{ret}(\lfloor \mathtt{tw} \rfloor_0, 0, 1) \cdot \mathbf{call}(\lfloor \mathtt{tw} \rfloor_0, 1, 1) \cdot$$
$$\mathbf{call}(\mathtt{inc}, 1, 1) \cdot \mathbf{ret}(\mathtt{inc}, 1, 2) \cdot \mathbf{ret}(\lfloor \mathtt{tw} \rfloor_0, 1, 2) \cdot \mathbf{ret}(\lceil \mathtt{tw} \rceil_0, 0, 2) \cdot \mathbf{ret}(\mathtt{main}, 0, 2)$$

Similarly, the infinite traces $[\![D]\!]_{\mathtt{inf}}$ of a program D are collected using the *nonterminating* trace-labeled reduction relation $C \stackrel{\pi}{\Rightarrow} \bot$ on configurations. Intuitively, $C \stackrel{\pi}{\Rightarrow} \bot$ means that an execution from the configuration C diverges,

producing an infinite event sequence π. In the rule TRANω, the double horizontal line represents that the rule is interpreted co-inductively.

We now define the translation from traces $[\![D]\!]_{\mathtt{fin}}$ to HOTs with $\Sigma^T_{\mathbf{call}} = \{\mathbf{call}(f,n), \mathbf{call}(f,\bullet) \mid f \in \mathrm{dom}(D), n \in \mathbb{Z}_b\}$, $\Sigma^A_{\mathbf{call}} = \{\mathbf{call}(\bullet,n), \mathbf{call}(\bullet,\bullet) \mid n \in \mathbb{Z}_b\}$, and $\Sigma_{\mathbf{ret}} = \{\mathbf{ret}(f,n), \mathbf{ret}(f,\bullet), \mathbf{ret}(\bullet,n), \mathbf{ret}(\bullet,\bullet) \mid f \in \mathrm{dom}(D), n \in \mathbb{Z}_b\}$. We shall write $\Sigma(D)$ for $\Sigma^T_{\mathbf{call}} \cup \Sigma^A_{\mathbf{call}} \cup \Sigma_{\mathbf{ret}}$. Note that $\Sigma(D)$ is finite because $\mathrm{dom}(D)$ and $\mathbb{Z}_b$ are finite. We write $|\alpha|$ for the element of $\Sigma(D)$ obtained from the event α by dropping the second argument and replacing $\lfloor h \rfloor_i$ and $\lceil h \rceil_i$ by $\bullet$. For example, we get $|\mathbf{call}(\mathtt{tw}, 0, \lfloor \mathtt{tw} \rfloor_0)| = \mathbf{call}(\mathtt{tw}, \bullet)$.

Definition 3 (Finite Traces to HOTs). *Given a finite trace $\varpi = \alpha_1 \cdots \alpha_m \in [\![D]\!]_{\mathtt{fin}}$ with $m > 0$, the corresponding HOT $G_\varpi = (V_\varpi, \lambda_\varpi, \nu_\varpi)$ is defined by:*

- $V_\varpi = \{1, \ldots, m\}$,
- $\lambda_\varpi = \{j \mapsto |\alpha_j| \mid j \in V_\varpi\}$, *and*
- ν_ϖ *is the smallest relation that satisfies: for any $j_1, j_2 \in V_\varpi$,*
  - $j_1 \prec_{\mathbf{N}} j_2$ *if* $j_2 = j_1 + 1$,
  - $j_1 \prec_{\mathbf{CR}} j_2$ *if* $\exists h, h', h'', i.\ \alpha_{j_1} = \mathbf{call}(h,i,h') \wedge \alpha_{j_2} = \mathbf{ret}(h,i,h'')$,
  - $j_1 \prec_{\mathbf{CC}} j_2$ *if* $\exists h, h', h'', i, i'.\ \alpha_{j_1} = \mathbf{call}(h',i,h) \wedge \alpha_{j_2} = \mathbf{call}(h,i',h'')$,
  - $j_1 \prec_{\mathbf{RC}} j_2$ *if* $\exists h, h', h'', i, i'.\ \alpha_{j_1} = \mathbf{ret}(h',i,h) \wedge \alpha_{j_2} = \mathbf{call}(h,i',h'')$.

For example, the HOT $G_{\mathtt{tw}}$ in Section 1 is translated from the finite trace $\varpi_{\mathtt{tw}}$ defined above (with the call and return events of `main` omitted).

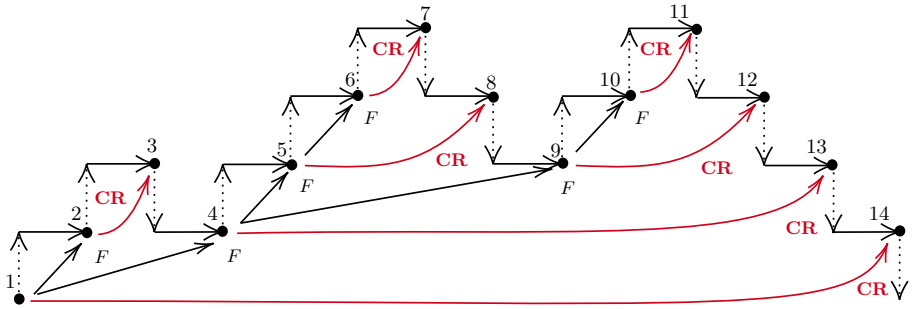
For an infinite trace $\pi = \alpha_1 \cdot \alpha_2 \cdots \in [\![D]\!]_{\mathtt{inf}}$, the HOT $G_\pi = (V_\pi, \lambda_\pi, \nu_\pi)$ is defined similarly for $V_\pi = \{j \in \mathbb{N} \mid j \geq 1\}$ and $\lambda_\pi = \{j \mapsto |\alpha_j| \mid j \in V_\pi\}$.

3 Propositional Dynamic Logic over Higher-Order Traces

This section presents HOT-PDL, a propositional dynamic logic (PDL) defined over HOTs (see [16] for a general exposition of PDL). HOT-PDL extends path expressions of PDL with $\rightarrow_{\mathbf{ret}}$ and $\rightarrow_{\mathbf{call}}$ for traversing edges of HOTs labeled respectively with **CR** and **CC**/**RC**. The syntax is defined by:

$$\begin{array}{rl} \text{(formulas)} & \phi ::= p \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid [\pi]\,\phi \\ \text{(path expressions)} & \pi ::= \rightarrow \mid \rightarrow_{\mathbf{call}} \mid \rightarrow_{\mathbf{ret}} \mid \{\phi\}? \mid \pi_1 \cdot \pi_2 \mid \pi_1 + \pi_2 \mid \pi^* \end{array}$$

Here, p is a meta-variable ranging over atomic propositions $\mathcal{AP}$. Let $\top$ and $\bot$ denote tautology and contradiction, respectively. Path expressions π are defined using a syntax based on regular expressions: we have concatenation $\pi_1 \cdot \pi_2$, alternation $\pi_1 + \pi_2$, and Kleene star π^*. We write π^+ for $\pi \cdot \pi^*$. Path expressions $\rightarrow$, $\rightarrow_{\mathbf{ret}}$, and $\rightarrow_{\mathbf{call}}$ are for traversing edges labeled with **N**, **CR**, and **CC** or **RC**, respectively. A path expression $\{\phi\}?$ is for testing if ϕ holds at the current node. A formula $[\pi]\,\phi$ means that ϕ always holds if one moves along any path represented by the path expression π. The dual formula $\langle\pi\rangle\,\phi$ is defined by $\neg\,[\pi]\,\neg\phi$ and means that there is a path represented by π such that ϕ holds if one moves along the path. $\langle\pi\rangle$ and $[\pi]$ have the same priority as $\neg$.



$1 : \mathbf{call}(\mathtt{main}, 0), 2 : \mathbf{call}(\mathtt{tw}, \bullet), 3 : \mathbf{ret}(\mathtt{tw}, \bullet), 4 : \mathbf{call}(\bullet, 0), 5 : \mathbf{call}(\bullet, 0),$
$6 : \mathbf{call}(\mathtt{inc}, 0), 7 : \mathbf{ret}(\mathtt{inc}, 1), 8 : \mathbf{ret}(\bullet, 1), 9 : \mathbf{call}(\bullet, 1), 10 : \mathbf{call}(\mathtt{inc}, 1),$
$11 : \mathbf{ret}(\mathtt{inc}, 2), 12 : \mathbf{ret}(\bullet, 2), 13 : \mathbf{ret}(\bullet, 2), 14 : \mathbf{ret}(\mathtt{main}, 2)$

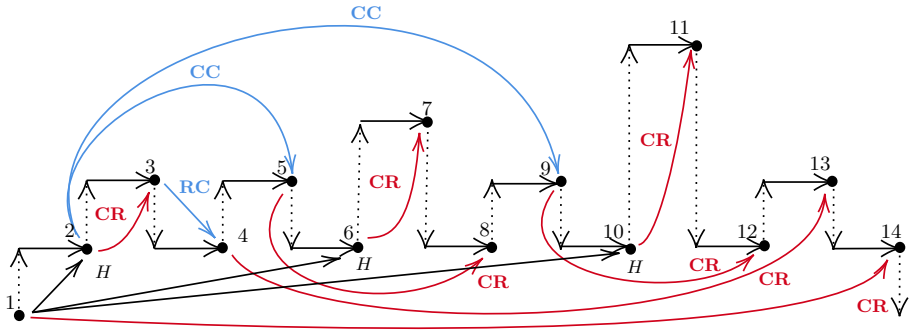
Fig. 3. The pairs of nodes in $G_{\mathtt{tw}}$ related by **CR** or $\nearrow_F$

We now define the semantics of HOT-PDL. For a given HOT $G = (V, \lambda, \nu)$ with $\Sigma = \mathcal{AP}$, $\lambda(u)$ represents the atomic proposition satisfied at the node $u \in V$. We define the semantics $[\![\phi]\!]_G$ of a formula ϕ as the set of all nodes $u \in V$ where ϕ is satisfied, and the semantics $[\![\pi]\!]_G$ of a path expression π as the set of all pairs $(u_1, u_2) \in V \times V$ such that one can move along π from u_1 to u_2.

$$[\![p]\!]_G = \{u \in V \mid p = \lambda(u)\} \qquad [\![\phi_1 \wedge \phi_2]\!]_G = [\![\phi_1]\!]_G \cap [\![\phi_2]\!]_G \qquad [\![\neg\phi]\!]_G = V \setminus [\![\phi]\!]_G$$
$$[\![[\pi]\,\phi]\!]_G = \{u \in V \mid \forall u'.\, ((u, u') \in [\![\pi]\!]_G \Rightarrow u' \in [\![\phi]\!]_G)\}$$
$$[\![\rightarrow]\!]_G = \prec_{\mathbf{N}} \qquad [\![\rightarrow_{\mathbf{ret}}]\!]_G = \prec_{\mathbf{CR}} \qquad [\![\rightarrow_{\mathbf{call}}]\!]_G = \prec_{\mathbf{CC}} \cup \prec_{\mathbf{RC}}$$
$$[\![\{\phi\}?]\!]_G = \{(u, u) \in V \times V \mid u \in [\![\phi]\!]_G\}$$
$$[\![\pi_1 \cdot \pi_2]\!]_G = \{(u_1, u_3) \in V \times V \mid \exists u_2 \in V.\, (u_1, u_2) \in [\![\pi_1]\!]_G \wedge (u_2, u_3) \in [\![\pi_2]\!]_G\}$$
$$[\![\pi_1 + \pi_2]\!]_G = [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G \qquad [\![\pi^*]\!]_G = \bigcup_{m \geq 0} [\![\pi]\!]_G^m$$

Here, for a binary relation R, R^m denotes the m-th power of R. Note that this semantics can interpret a given HOT-PDL formula over both finite and infinite HOTs. $[\![p]\!]_G$ consists of all nodes labeled by p. $[\![[\pi]\,\phi]\!]_G$ contains all nodes from which we always reach to a node in $[\![\phi]\!]_G$ if we take a path represented by π. $[\![\rightarrow]\!]_G$, $[\![\rightarrow_{\mathbf{ret}}]\!]_G$, and $[\![\rightarrow_{\mathbf{call}}]\!]_G$ contain the pairs of nodes linked by an edge labeled by **N**, **CR**, and **CC** or **RC**, respectively. We write $G \models \phi$ if $0_G \in [\![\phi]\!]_G$. For example, let us consider the HOT $G_{\mathtt{tw}}$ and $\mathcal{AP} = \Sigma(D_{\mathtt{tw}})$. Then, $[\![\langle\rightarrow\rangle\, \mathbf{ret}(\mathtt{tw}, \bullet)]\!]_{G_{\mathtt{tw}}}$ consists of the node labeled by $\mathbf{call}(\mathtt{tw}, \bullet)$. $[\![\langle\rightarrow_{\mathbf{ret}}\rangle\, \mathbf{ret}(\bullet, 2)]\!]_{G_{\mathtt{tw}}}$ consists of a node labeled by $\mathbf{call}(\bullet, 0)$ and the node labeled by $\mathbf{call}(\bullet, 1)$. $[\![\langle\rightarrow_{\mathbf{call}}\rangle\, \mathbf{call}(\bullet, 0)]\!]_{G_{\mathtt{tw}}}$ consists of the two nodes respectively labeled by $\mathbf{call}(\mathtt{tw}, \bullet)$ and $\mathbf{ret}(\mathtt{tw}, \bullet)$. The example properties of $D_{\mathtt{tw}}$ discussed in Section 1 can be expressed as follows:

Prop.1: $[\rightarrow^*] \bigwedge_{x \in \mathbb{Z}_b} ((\mathbf{call}(\mathtt{tw}, \bullet) \wedge \langle\rightarrow_{\mathbf{ret}} \cdot \rightarrow_{\mathbf{call}}\rangle\, \mathbf{call}(\bullet, x)) \Rightarrow \langle\rightarrow_{\mathbf{call}}\rangle\, \mathbf{call}(\bullet, x))$
Prop.2: $[\rightarrow^*]\, ((\mathbf{call}(\mathtt{tw}, \bullet) \wedge \neg\, \langle\rightarrow_{\mathbf{ret}} \cdot \rightarrow_{\mathbf{call}}\rangle\, \top) \Rightarrow \neg\, \langle\rightarrow_{\mathbf{call}}\rangle\, \top)$



$1 : \mathbf{call}(\mathtt{main}, 0), 2 : \mathbf{call}(\mathtt{tw}, \bullet), 3 : \mathbf{ret}(\mathtt{tw}, \bullet), 4 : \mathbf{call}(\bullet, 0), 5 : \mathbf{call}(\bullet, 0),$
$6 : \mathbf{call}(\mathtt{inc}, 0), 7 : \mathbf{ret}(\mathtt{inc}, 1), 8 : \mathbf{ret}(\bullet, 1), 9 : \mathbf{call}(\bullet, 1), 10 : \mathbf{call}(\mathtt{inc}, 1),$
$11 : \mathbf{ret}(\mathtt{inc}, 2), 12 : \mathbf{ret}(\bullet, 2), 13 : \mathbf{ret}(\bullet, 2), 14 : \mathbf{ret}(\mathtt{main}, 2)$

Fig. 4. The pairs of nodes in $G_{\mathtt{tw}}$ related by **CR**, **CC**, **RC**, or $\nearrow_H$

Here, $\bigwedge_{x \in \mathbb{Z}_b} \phi$ abbreviates $[n_{\mathbf{min}}/x]\,\phi \wedge \cdots \wedge [n_{\mathbf{max}}/x]\,\phi$.

In Section 4, we show further examples that express interesting properties of higher-order programs, including stack based access control properties and those definable using dependent refinement types. We here prepare notations used there. First, we overload the symbols $\Sigma_{\mathbf{call}}$, $\Sigma_{\mathbf{ret}}$, and $\Sigma^T_{\mathbf{call}}$ to denote the path expressions $\{\bigvee \Sigma_{\mathbf{call}}\}?$, $\{\bigvee \Sigma_{\mathbf{ret}}\}?$, and $\{\bigvee \Sigma^T_{\mathbf{call}}\}?$, respectively. We write $\rightarrow_F$ for the path expression $\rightarrow_{\mathbf{ret}} \cdot \rightarrow$, which is used to move from a call event to the next event of the caller (by skipping to the next event of the corresponding return event). We also write $\nearrow_F$ for the path expression $\Sigma_{\mathbf{call}} \cdot \rightarrow \cdot \rightarrow^*_F \cdot \Sigma_{\mathbf{call}}$, which is used to move from a call event to any call event invoked by the callee. Figure 3 illustrates the pairs of nodes in $G_{\mathtt{tw}}$ related by $\nearrow_F$. To capture control flow of higher-order programs, where function callers and callees may exchange functions as values, we need to use **CC**- and **RC**-labeled edges. For example, an event raised by the function argument f_{arg} of a higher-order function f could be regarded as an event of the caller g of f, because f_{arg} is constructed by g. Similarly, an event raised by the (partially-applied) function f_{ret} returned by a function f could be regarded as an event of f. To formalize the idea, we introduce variants $\rightarrow_H$ and $\nearrow_H$ of $\rightarrow_F$ and $\nearrow_F$ with higher-order control flow taken into consideration: $\rightarrow_H$ denotes $(\rightarrow_{\mathbf{ret}} \cdot \rightarrow) + (\rightarrow_{\mathbf{call}} \cdot \rightarrow)$ and $\nearrow_F$ denotes $\Sigma^T_{\mathbf{call}} \cdot \rightarrow \cdot \rightarrow^*_H \cdot \Sigma^T_{\mathbf{call}}$. Note that the source and the target of $\nearrow_H$ are restricted to call events of top-level functions. Figure 4 illustrates the pairs of nodes in $G_{\mathtt{tw}}$ related by $\nearrow_H$, where nodes labeled with events of the same function (in the sense discussed above) are arranged in the same horizontal line.

4 Applications of HOT-PDL

We show how to encode dependent refinement types and stack-based access control properties using HOT-PDL.

4.1 Dependent Refinement Types

HOT-PDL can specify pre- and post-conditions of higher-order functions, by encoding dependent refinement types τ for partial [29, 33, 40] and total [23, 27, 34, 36, 39] correctness verification, defined as: $\tau ::= \{\nu \mid \psi\} \mid (x : \tau_1) \to \tau_2^Q$. Here, Q is either $\forall$ or $\exists$. An integer refinement type $\{\nu \mid \psi\}$ is the type of bounded integers ν that satisfy the refinement formula ψ over bounded integers. A dependent function type $(x : \tau_1) \to \tau_2^\forall$ is the type of functions that, for any argument x conforming to the type τ_1, *if terminating*, return a value conforming to the type τ_2. By contrast, $(x : \tau_1) \to \tau_2^\exists$ is the type of functions that, for any argument x conforming to τ_1, *always terminate* and return a value conforming to τ_2. For example, Prop.3 and Prop.4 of $D_{\mathtt{tw}}$ are expressed by the following types of `tw`:

Prop.3: $(f : (x : \mathtt{int}) \to \{\nu \mid \nu > x\}^\forall) \to \left((x : \mathtt{int}) \to \{\nu \mid \nu > x\}^\forall\right)^\forall$

Prop.4: $(f : (x : \mathtt{int}) \to \mathtt{int}^\exists) \to \left((x : \mathtt{int}) \to \mathtt{int}^\exists\right)^\forall$

We here write `int` for $\{\nu \mid \top\}$. These types can be encoded in HOT-PDL as:

Prop.3: $\mathbf{call}(\mathtt{tw}, \bullet) \Rightarrow ([\to_{\mathbf{call}}]\, \mathtt{incr}(\bullet)) \wedge [\to_{\mathbf{ret}}] \left(\mathbf{ret}(\mathtt{tw}, \bullet) \Rightarrow [\to_{\mathbf{call}}]\, \mathtt{incr}(\bullet)\right)$

Prop.4: $\mathbf{call}(\mathtt{tw}, \bullet) \Rightarrow ([\to_{\mathbf{call}}]\, \mathtt{term}(\bullet)) \wedge [\to_{\mathbf{ret}}] \left(\mathbf{ret}(\mathtt{tw}, \bullet) \Rightarrow [\to_{\mathbf{call}}]\, \mathtt{term}(\bullet)\right)$

Here, $\mathtt{incr}(g) = \bigwedge_{x \in \mathbb{Z}_b} \mathbf{call}(g, x) \Rightarrow [\to_{\mathbf{ret}}] \bigwedge_{y \in \mathbb{Z}_b} (\mathbf{ret}(g, y) \Rightarrow y > x)$ and $\mathtt{term}(g) = \bigwedge_{x \in \mathbb{Z}_b} (\mathbf{call}(g, x) \Rightarrow \langle \to_{\mathbf{ret}} \rangle \top)$ for $g \in \{\bullet\} \cup \{f \mid f \in \mathrm{dom}(D)\}$. We now define a translation F from types to HOT-PDL formulas as follows:

$$F(g, (x : \tau_1) \to \tau_2^Q) = \bigwedge_{x \in |\tau_1|} \left(\mathbf{call}(g, x) \Rightarrow F_{arg}(x, \tau_1) \wedge F_{ret}(g, \tau_2^Q)\right)$$

$$|(x : \tau_1) \to \tau_2^Q| = \{\bullet\} \qquad |\{x \mid \psi\}| = \mathbb{Z}_b$$

$$F_{arg}(\bullet, \tau) = [\to_{\mathbf{call}}]\, F(\bullet, \tau) \qquad F_{arg}(n, \{x \mid \psi\}) = \begin{cases} \top & (\text{if } \models [n/x]\psi) \\ \bot & (\text{if } \not\models [n/x]\psi) \end{cases}$$

$$F_{ret}(g, \tau^\forall) = [\to_{\mathbf{ret}}] \bigwedge_{x \in |\tau|} (\mathbf{ret}(g, x) \Rightarrow F(x, \tau))$$

$$F_{ret}(g, \tau^\exists) = (\langle \to_{\mathbf{ret}} \rangle \top) \wedge F_{ret}(g, \tau^\forall)$$

4.2 Stack-Based Access Control Properties

As briefly summarized in Section 1, stack-based access control [13] ensures that a *security-critical* function (e.g., file access) is invoked only if all the (immediate and indirect) callers in the current call stack are *trusted*, or one of the callers

is a *privileged* function and its callees are all *trusted*. We here use HOT-PDL to specify stack-based access control properties for higher-order programs. Let **Critical**, **Trusted**, and **Priv** be HOT-PDL formulas that tell whether the current node is labeled with a call event of security-critical, trusted, and privileged functions, respectively. We assume that **Critical**, **Priv**, and $\neg$**Trusted** do not overlap each other, and a function in **Priv** can be directly called only from a function in **Trusted**. Then, one may think we can express the specification as:

$$\neg \left\langle \nearrow_F^* \cdot \{\neg \mathbf{Trusted}\}? \cdot (\nearrow_F \cdot \{\neg \mathbf{Priv}\}?)^+ \right\rangle \mathbf{Critical}$$

Here, the path expression $\nearrow_F$ introduced in Section 3 is used to traverse the call stack bottom-up. The above formula says that an invalid call stack never occurs, where a call stack is called *invalid* if it contains a call to an untrusted function (represented by the part $\nearrow_F^* \{\neg \mathbf{Trusted}\}?$), followed by a call to a critical function (represented by **Critical**), with no intervening call to a privileged function (represented by $(\nearrow_F \cdot \{\neg \mathbf{Priv}\}?)^+$).

This definition, however, is not sufficient for our higher-order language. Let us consider the following program D_{pa}, which involves a partial application:

$$\begin{aligned} &\texttt{let untrusted } () = \lambda u.\texttt{critical } u \\ &\texttt{let main } () = \texttt{untrusted } () \ () \end{aligned}$$

Here, $\texttt{untrusted} \notin \mathbf{Trusted}$ and $\texttt{critical} \in \mathbf{Critical}$. Intuitively, D_{pa} should be regarded as *unsafe* because `critical` in the body of `untrusted` is called. However, D_{pa} satisfies the specification above (under the assumption that anonymous functions are in **Trusted**), because the partial application `untrusted ()` never causes a call to `critical` but just returns the anonymous (and trusted) function $\lambda u.\texttt{critical } u$. The following higher-order program D_{ho} is yet another unsafe example that satisfies the specification:

$$\begin{aligned} &\texttt{let privileged } f = f \ () \\ &\quad \texttt{let trusted } f = \texttt{if test } () \texttt{ then privileged } f \texttt{ else } () \\ &\texttt{let untrusted } () = \texttt{trusted } (\lambda x.\texttt{crash } (); \texttt{critical } ()) \\ &\quad \texttt{let main } () = \texttt{untrusted } () \end{aligned}$$

Here, $\texttt{privileged} \in \mathbf{Priv}$, $\texttt{trusted} \in \mathbf{Trusted}$, $\texttt{untrusted} \notin \mathbf{Trusted}$, and $\texttt{critical} \in \mathbf{Critical}$. Note that `critical` in the body of `untrusted` is called as follows: the anonymous function $\lambda x.\texttt{crash } (); \texttt{critical } ()$ is first passed to `trusted` and then to `privileged` (if `test ()` returns `true`), and is finally called by `privileged`, causing a call to `critical`.

To remedy the limitation, we introduce a new refined variant of stack-based access control properties for higher-order programs, formalized in HOT-PDL from the point of view of interactions among callers and callees as follows:

$$\neg \left\langle \nearrow_H^* \cdot \{\neg \mathbf{Trusted}\}? \cdot (\nearrow_H \cdot \{\neg \mathbf{Priv}\}?)^+ \right\rangle \mathbf{Critical}$$

Note that this is obtained from the previous version by just replacing $\nearrow_F$ with $\nearrow_H$, which takes into account which function constructed each function value

exchanged among functions. The refined version rejects the unsafe D_{pa} and D_{ho} as intended: D_{pa} (resp. D_{ho}) is rejected because the call event of $\lambda u.\texttt{critical}\ u$ (resp. $\lambda x.\texttt{crash}\ ();\texttt{critical}\ ()$) is regarded as an event of `untrusted`.

Fournet et al. [13] have studied variants of stack-based access control properties for a call-by-value higher-order language. We conclude this section by comparing ours with one of theirs called "stack inspection with frame capture".[2] The ideas behind the two are similar but what follows illustrates the difference:

$$\begin{array}{l}\texttt{let untrusted}\ f = \texttt{crash}\ ();\ f\ () \\ \quad \texttt{let trusted}\ x = \texttt{untrusted}\ (\lambda x.\texttt{if test}\ ()\ \texttt{then critical}\ ()\ \texttt{else}\ ()) \\ \quad\quad \texttt{let main}\ () = \texttt{trusted}\ ()\end{array}$$

This program satisfies ours but violates theirs. Note that ours allows a function originally constructed by a trusted function to invoke a critical function even if the function is passed around by an untrusted function. By contrast, in their definition, a trusted function value gets "contaminated" (i.e., disabled to invoke a critical function) once it is passed to or returned by an untrusted function. In some cases, their conservative policy is useful, but we believe ours would be more semantically robust (e.g., even works well with the CPS transformation).

5 HOT-PDL Model Checking

In this section, we define HOT-PDL model checking problems for higher-order functional programs over bounded integers and sketch a proof of the decidability.

Definition 4 (HOT-PDL model checking). *Given a program D and a HOT-PDL formula ϕ with $\mathcal{AP} = \Sigma(D)$, HOT-PDL model checking is the problem of deciding whether $G_\varpi \models \phi$ and $G_\pi \models \phi$ for all $\varpi \in [\![D]\!]_{\texttt{fin}}$ and $\pi \in [\![D]\!]_{\texttt{inf}}$.*

Theorem 1 (Decidability). *HOT-PDL model checking is decidable.*

We show this by a reduction to modal μ-calculus (μ-ML) model checking of higher-order recursion schemes (HORSs), which is known decidable [21, 28]. A HORS is a grammar for generating a (possibly infinite) ranked tree, and HORSs are essentially simply-typed lambda calculus with general recursion, tree constructors, and finite data domains such as booleans and bounded integers.

In the reduction, we encode the set of HOTs that are generated from the given program D as a single tree (generated by a HORS). For example, Figure 5 shows such a tree that encodes the HOTs of $D_{\texttt{tw}}$.[3] There, a node labeled with `end` represents the termination of the program. Note that the branching at the root node is due to the input to the function `main`. The subtree with the root node labeled with $\textbf{call}(\texttt{main}, 0)$ is obtained from the HOT $G_{\texttt{tw}}$ by appending a special node labeled with `end`, adding, for each edge with the label $\gamma \in \{\mathbf{N}, \mathbf{CR}, \mathbf{CC}, \mathbf{RC}\}$,

[2] We do not compare with the other variants in [13] because they are too syntactic to be preserved by simple program transformations like inlining.

[3] There, for simplicity, we illustrate an *unranked* tree and omit the label of branching nodes. In the formalization, we express an unranked tree as a binary tree using a special node label **br** of the arity 2 representing a binary branching.

a new node labeled with γ, and expanding the resulting DAG into a tree. Thus, the edge labels of $G_{\mathtt{tw}}$ are turned into node labels of the tree.

It is also worth mentioning here that we are allowed to expand DAGs into trees because the truth value of a HOT-PDL formula is not affected by node-sharing in the given HOT. This nice property is lost if we extend the path expressions of HOT-PDL, for example, with intersections. Thus, the decidability of model checking for extensions of HOT-PDL is an open problem.

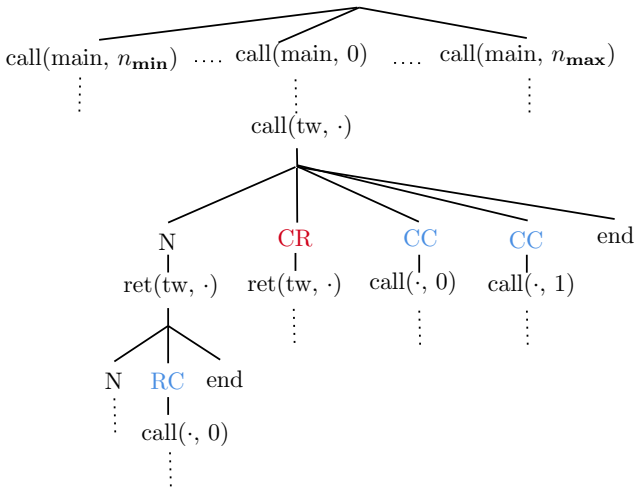


Fig. 5. A tree encoding the HOTs generated from $D_{\mathtt{tw}}$

We next explain our translation from a HOT-PDL formula into a μ-ML formula interpreted over trees that encode HOTs. Our translation is based on an existing one for ordinary PDL [11]. The syntax of μ-ML is defined as follows:

$$\varphi ::= X \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box\varphi \mid \nu X.\varphi \mid \mu X.\varphi$$

Here, X represents a propositional variable and p represents an atomic proposition. A formula $\Box\varphi$ means that φ holds for any child of the current node. A formula $\mu X.\varphi$ (resp. $\nu X.\varphi$) represents the least (resp. greatest) fixpoint of the function $\lambda X.\varphi$. Here, we assume X occurs only positively in φ. For example, the HOT-PDL formulas $[\rightarrow]\,p$, $[\rightarrow_{\mathbf{ret}}]\,p$, and $[\rightarrow_{\mathbf{call}}]\,p$ are respectively translated to μ-ML formulas: $\Box(\nu X.(\mathbf{N} \Rightarrow \Box p) \wedge (\mathbf{br} \Rightarrow \Box X))$, $\Box(\nu X.(\mathbf{CR} \Rightarrow \Box p) \wedge (\mathbf{br} \Rightarrow \Box X))$, and $\Box(\nu X.((\mathbf{CC} \vee \mathbf{RC}) \Rightarrow \Box p) \wedge (\mathbf{br} \Rightarrow \Box X))$, where the greatest fixpoints are used to skip the branching nodes labeled with **br** (that may repeat infinitely).

Finally, we explain how to obtain a HORS for generating a tree that encodes the set of HOTs generated from the given program D. We here need to simulate pointer traversals of HOT-PDL by using purely functional features of HORSs because μ-ML does not support pointers. Intuitively, we obtain the desired HORS from D by embedding an event monitor and an event handler. Whenever the monitor detects a function call or return event during the execution of D, the handler creates a new node labeled with the event or ignores the event until a certain event is detected by the monitor, depending on the current mode of the handler. The handler has the following three modes:

$m_{\mathbf{N}}$: The handler always creates and links two new nodes $u_{\mathbf{N}}$ and u_α labeled respectively with $\mathbf{N}$ and the event α observed. The handler then continues as follows, depending on the form of the event α:

- $\mathbf{call}(g, n)$: Spawns a new handler with the mode $m_{\mathbf{ret}}$. Then, the two handlers of the modes $m_{\mathbf{N}}$ and $m_{\mathbf{ret}}$ continue to create subtrees of u_α.
- $\mathbf{call}(g, \bullet)$: Spawns two new handlers with the modes $m_{\mathbf{ret}}$ and $m_{\mathbf{call}}$. The three handlers of $m_{\mathbf{N}}$, $m_{\mathbf{ret}}$, and $m_{\mathbf{call}}$ continue to create subtrees of u_α.
- $\mathbf{ret}(g, n)$: The handler of the mode $m_{\mathbf{N}}$ continues to create a subtree of u_α.
- $\mathbf{ret}(g, \bullet)$: Spawns a new handler with the mode $m_{\mathbf{call}}$. Then, the two handlers of the modes $m_{\mathbf{N}}$ and $m_{\mathbf{call}}$ continue to create subtrees of u_α.

$m_{\mathbf{ret}}$: The handler ignores all events but the return event corresponding to the call event that caused the spawn of the handler. If not ignored, the handler creates and links new nodes $u_{\mathbf{CR}}$ and u_α labeled with **CR** and the event α. The handler changes its mode to $m_{\mathbf{N}}$ and continues creating a subtree of u_α.

$m_{\mathbf{call}}$: The handler ignores all events but the call event of the function passed to or returned by the call or return event that caused the spawn of the handler. If not ignored, the handler creates and links new nodes u and u_α labeled respectively with **CC** or **RC** and the event α, duplicates itself, and changes the mode of the original to $m_{\mathbf{N}}$. The handler of the mode $m_{\mathbf{N}}$ (resp. $m_{\mathbf{call}}$) continues to create a subtree of u_α (resp. the parent of u).

For simplicity of the construction, we assume that D is in the Continuation-Passing Style (CPS). This does not lose generality because we can enforce this form by the CPS transformation. Because CPS explicates the order of function call and return events, it simplifies event monitoring, handling, and tracking of the current mode of the monitors, which often changes as monitoring proceeds.

6 Related Work

HOT-PDL can specify temporal trace properties of higher-order programs. An extension for specifying branching properties, however, remains a future work.

There have been proposed logics and formal languages on richer structures than words. Regular languages of nested words, or equivalently, Visibly Pushdown Languages (VPLs) have been introduced by Alur and Madhusudan [7]. An (ω-)nested word is a (possibly infinite) word with additional well-nested pointers from call events to the corresponding return events. Compared to temporal logics CaRet [5] and NWTL [4] over (ω-)nested words, HOT-PDL is defined over HOTs that have richer structures. Recall that a HOT is equipped with two kinds of pointers: one kind with the label **CR**, which is the same as the pointers of nested words, and the other kind with the label **CC** or **RC**, which is newly introduced to capture higher-order control flow. Bollig et al. proposed nested traces as a generalization of nested words for modeling traces of concurrent (first-order) recursive programs, and presented temporal logics over nested traces [8]. Nested traces, however, cannot model traces of higher-order programs. We expect a combination of our work with theirs enables us to specify temporal trace properties of concurrent and higher-order recursive programs. Cyriac et

al. have recently introduced an extension of PDL defined over traces of *order-2* collapsible pushdown systems (CPDS) [3]. Interestingly, their traces are also equipped with two kinds of pointers: one kind of pointers captures the correspondence between ordinary push and pop stack operations, and the other captures the correspondence between order-2 push and pop operations for second-order stacks. Our work deals with higher-order programs that correspond to order-n CPDS for arbitrary n.

Finally, we compare HOT-PDL with existing logics defined over words. It is well known that LTL is less expressive than ω-regular languages [38]. To remedy the limitation of LTL, Wolper introduced ETL [38] that allows users to define new temporal operators using right-linear grammars. Henriksen and Thiagarajan proposed DLTL [17] that generalizes the until operator of LTL using regular expressions. Leucker and Sánchez proposed RLTL [25] that combines LTL and regular expressions. Vardi and Giacomo have introduced Linear Dynamic Logic (LDL), a variant of PDL interpreted over infinite words [15, 35]. LDL_f, a variant of PDL interpreted over finite words, has also been studied in [15]. ETL, DLTL, RLTL, and LDL are as expressive as ω-regular languages. Note that HOT-PDL subsumes (ω-)regular languages because LDL and LDL_f can be naturally embedded in HOT-PDL. (ω-)VPLs strictly subsume (ω-)regular languages. Though CaRet [5] and NWTL [4] are defined over nested words, they do not capture the full class of VPLs [10]. To remedy the limitation, VLTL [10] combines LTL and VRE [9] in the style of RLTL, where VRE is a generalization of regular expressions for VPLs. VLDL [37] extends LDL by replacing the path expressions with VPLs over finite words. VLTL and VLDL exactly characterize ω-VPLs. Because VPLs and HOT-PDL are incomparable, it remains future work to extend HOT-PDL to subsume (ω-)VPLs.

7 Conclusion and Future Work

We have presented HOT-PDL, an extension of PDL defined over HOTs that model execution traces of call-by-value and higher-order programs. HOT-PDL enables a precise specification of temporal trace properties of higher-order programs and consequently provides a foundation for specification in various application domains including stack-based access control and dependent refinement types. We have also studied HOT-PDL model checking and presented a reduction method to modal μ-calculus model checking of higher-order recursion schemes.

To further widen the scope of our approach, it is worth investigating how to adapt HOTs and HOT-PDL to call-by-name and/or effectful languages. To this end, it is natural to incorporate more ideas from achievements of game semantics [1, 20, 32] and extend HOTs with new kinds of events and pointers for capturing call-by-name and/or effectful computations.

Acknowledgments We would like to thank anonymous referees for their useful comments. This work was supported by JSPS KAKENHI Grant Numbers 15H05706, 16H05856, 17H01720, and 17H01723.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Information and Computation **163**, 409–470 (2000)
2. Abramsky, S., McCusker, G.: Call-by-value games. In: CSL '98. LNCS, vol. 1414, pp. 1–17. Springer (1998)
3. Aiswarya, C., Gastin, P., Saivasan, P.: Nested words for order-2 pushdown systems. arXiv:1609.06290 (2016)
4. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. Logical Methods in Computer Science **Volume 4, Issue 4** (2008)
5. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: TACAS '04. pp. 467–481. Springer (2004)
6. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC '04. pp. 202–211. ACM (2004)
7. Alur, R., Madhusudan, P.: Adding nesting structure to words. Journal of the ACM **56**(3), 16:1–16:43 (2009)
8. Bollig, B., Cyriac, A., Gastin, P., Zeitoun, M.: Temporal logics for concurrent recursive programs: Satisfiability and model checking. Journal of Applied Logic **12**(4), 395 – 416 (2014)
9. Bozzelli, L., Sánchez, C.: Visibly rational expressions. In: FSTTCS '12. LIPIcs, vol. 18, pp. 211–223. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2012)
10. Bozzelli, L., Sánchez, C.: Visibly linear temporal logic. In: IJCAR '14. LNCS, vol. 8562, pp. 418–433. Springer (2014)
11. Carreiro, F., Venema, Y.: PDL inside the μ-calculus: A syntactic and an automata-theoretic characterization. Advances in Modal Logic **10**, 74–93 (2014)
12. Disney, T., Flanagan, C., McCarthy, J.: Temporal higher-order contracts. In: ICFP '11. pp. 176–188. ACM (2011)
13. Fournet, C., Gordon, A.D.: Stack inspection: Theory and variants. In: POPL '02. pp. 307–318. ACM (2002)
14. Fujima, K., Ito, S., Kobayashi, N.: Practical alternating parity tree automata model checking of higher-order recursion schemes. In: APLAS '13. LNCS, vol. 8301, pp. 17–32. Springer (2013)
15. Giacomo, G.D., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI '13. pp. 854–860. AAAI Press (2013)
16. Harel, D., Tiuryn, J., Kozen, D.: Dynamic Logic. MIT Press (2000)
17. Henriksen, J.G., Thiagarajan, P.: Dynamic linear time temporal logic. Annals of Pure and Applied Logic **96**(1), 187 – 207 (1999)
18. Hofmann, M., Chen, W.: Abstract interpretation from Büchi automata. In: CSL-LICS '14. pp. 51:1–51:10. ACM (2014)
19. Honda, K., Yoshida, N.: Game theoretic analysis of call-by-value computation. In: ICALP '97. LNCS, vol. 1256, pp. 225–236. Springer (1997)
20. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II, and III. Information and Computation **163**, 285–408 (2000)
21. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: LICS '09. pp. 179–188. IEEE (2009)
22. Kobayashi, N., Tsukada, T., Watanabe, K.: Higher-order program verification via HFL model checking. In: ESOP '18. pp. 711–738. Springer (2018)

23. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: CSL-LICS '14. pp. 59:1–59:10. ACM (2014)
24. Lester, M.M., Neatherway, R.P., Ong, C.H.L., Ramsay, S.J.: Model checking liveness properties of higher-order functional programs. URL `http://mjolnir.comlab.ox.ac.uk/papers/thors.pdf` (2011)
25. Leucker, M., Sánchez, C.: Regular linear temporal logic. In: ICTAC '07. pp. 291–305. Springer (2007)
26. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: POPL '16. pp. 57–68. ACM (2016)
27. Nanjo, Y., Unno, H., Koskinen, E., Terauchi, T.: A fixpoint logic and dependent effects for temporal property verification. In: LICS '18. ACM (2018)
28. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS '06. pp. 81–90. IEEE (2006)
29. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI '08. pp. 159–169. ACM (2008)
30. Satake, Y., Unno, H.: Propositional dynamic logic for higher-order functional programs. Extended version, available from `http://www.cs.tsukuba.ac.jp/~uhiro/` (2018)
31. Suzuki, R., Fujima, K., Kobayashi, N., Tsukada, T.: Streett automata model checking of higher-order recursion schemes. In: FSCD '17. LIPIcs, vol. 84, pp. 32:1–32:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017)
32. Tzevelekos, N.: Nominal game semantics. Ph.D. thesis, University of Oxford (2008)
33. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP '09. pp. 277–288. ACM (2009)
34. Unno, H., Satake, Y., Terauchi, T.: Relatively complete refinement type system for verification of higher-order non-deterministic programs. Proc. ACM Program. Lang. **2**(POPL), 12:1–12:29 (Dec 2017)
35. Vardi, M.Y.: The rise and fall of LTL. GandALF (2011)
36. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton Jones, S.L.: Refinement types for Haskell. In: ICFP '14. pp. 269–282. ACM (2014)
37. Weinert, A., Zimmermann, M.: Visibly linear dynamic logic. In: FSTTCS '16. LIPIcs, vol. 65, pp. 28:1–28:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2016)
38. Wolper, P.: Temporal logic can be more expressive. Information and Control **56**(1), 72–99 (1983)
39. Xi, H.: Dependent types for program termination verification. In: LICS '01. pp. 231–242. IEEE (2001)
40. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL '99. pp. 214–227. ACM (1999)