A Temporal Logic for Higher-Order Functional Programs

Yuya Okuyama, Takeshi Tsukada(✉), and Naoki Kobayashi

The University of Tokyo, Tokyo, Japan
{okuyama,tsukada,koba}@kb.is.s.u-tokyo.ac.jp

Abstract. We propose an extension of linear temporal logic that we call Linear Temporal Logic of Calls (LTLC) for describing temporal properties of higher-order functions, such as "the function calls its first argument before any call of the second argument." A distinguishing feature of LTLC is a new modal operator, the call modality, that checks if the function specified by the operator is called in the current step and, if so, describes how the arguments are used in the subsequent computation. We demonstrate expressiveness of the logic, by giving examples of LTLC formulas describing interesting properties. Despite its high expressiveness, the model checking problem is decidable for deterministic programs with finite base types.

1 Introduction

Specifications of programs (or other systems) are often described by temporal or modal logics such as linear temporal logic (LTL), computational tree logic (CTL) and modal μ-calculus [2,8,12,16,19,22]. Formulas of these logics are built from atomic propositions representing basic properties of run-time states, e.g. whether the control is at a certain program point and whether the value of a certain global variable is positive. The set of atomic propositions in these logics is *static* in the sense that it remains unchanged during evaluation of programs.

We are interested in verification of higher-order functional programs, and logics suitable for describing temporal properties of such programs. For example, consider a function $g : (\texttt{unit} \to \texttt{string}) \to (\texttt{int} \to \texttt{int})$, which takes (the reader function of) a read-only file and creates a function on integers. The following is a possible specification for g:

– g is allowed to access the reader function only until it returns.

The implementation

$$\mathbf{let}\, g\, r \;=\; \left(\mathbf{let}\, w \;=\; \mathit{int\_of\_string}\,(r\,())\,\mathbf{in}\,\lambda x.x + w\right)$$

meets the specification, whereas

$$\mathbf{let}\, g\, r \;=\; \lambda x.x + (\mathit{int\_of\_string}\,(r\,()))$$

B.-Y. E. Chang (Ed.): SAS 2019, LNCS 11822, pp. 437–458, 2019.
https://doi.org/10.1007/978-3-030-32304-2_21

violates it.

Properly describing this specification is difficult because the property refers to *dynamic* notions such as "the argument of the first call of g." The program

$$\begin{array}{l}\mathbf{let}\, g\, r \;=\; \left(\mathbf{let}\, w \;=\; \mathit{int\_of\_string}\,(r\,())\,\mathbf{in}\,\lambda x.x + w\right)\,\mathbf{in}\\ \mathbf{let}\, \mathit{fp} = \mathit{open}\ \mathit{filename}\,\mathbf{in}\\ \mathbf{let}\, r = \lambda\_.\mathit{read\_line}\ \mathit{fp}\,\mathbf{in}\\ \mathbf{let}\, v_1 = g\, r\,\mathbf{in}\\ \mathbf{let}\, v_2 = g\, r\,\mathbf{in}\\ \quad \mathbf{print}\,(v_1\, 1 + v_2\, 2)\end{array}$$

is an example that calls g twice. The sequence of call and return events of g and r in the evaluation is written as

$$\mathbf{call}\, g_1 \;\rightarrow\; \mathbf{call}\, r_1 \;\rightarrow\; \mathbf{ret}\, r_1 \;\rightarrow\; \mathbf{ret}\, g_1 \;\rightarrow\; \mathbf{call}\, g_2 \;\rightarrow\; \mathbf{call}\, r_2 \;\rightarrow\; \mathbf{ret}\, r_2 \;\rightarrow\; \mathbf{ret}\, g_2,$$

where g_1 means the first call of g and r_1 is its argument and similarly for g_2 and r_2. The above program satisfies the property since there is no $\mathbf{call}\, r_i$ after $\mathbf{ret}\, g_i$ for $i = 1, 2$. However, by ignoring the subscripts, i.e. confusing the first call of g with the second call, the program may seem to violate the specification since there is $\mathbf{call}\, r_2$ after $\mathbf{ret}\, g_1$. This means that references to dynamic notions like the subscripts in the above sequence are inevitable for precise description of the specification. For this reason, the specification does not seem to be expressible in the standard logics listed above.

This paper proposes a new temporal logic, named *Linear Temporal Logic of Calls* (*LTLC* for short), by which one can describe the above property. The logic is an extension of Linear Temporal Logic (LTL) with a new operator $\mathit{call}\, f(x_1, \ldots, x_n).\varphi$, the *call operator*, where the occurrences of $x_1, \ldots, x_n$ are binding occurrences. Intuitively this formula is true just if the function f is called in the current step (i.e. the current expression is of the form $E[f\, e_1\, \ldots\, e_n]$) and $\varphi[e_1/x_1, \ldots, e_n/x_n]$ holds at the next step.[1] We shall see in Example 2 an LTLC formula describing the above property.

LTLC is expressive. One can describe properties written by dependent refinement types in the form of [21], relational properties [1,5] (e.g. monotonicity of a given function) and some examples in resource usage analysis [10].

Furthermore LTLC is tractable. The LTLC model-checking problem is not more difficult than the standard temporal verification problem, because the LTLC model-checking problem is effectively reducible to the standard temporal verification of programs. In particular, for programs over finite types,[2] the LTLC model-checking is reducible to higher-order model checking [12,17] and thus decidable.

[1] Here is a subtlety. We should distinguish different occurrences of the same expression, and here e_i means the occurrence of e_i as the i-th argument of this function call. See Sects. 3.2 and 3.3.

[2] This finiteness condition is obviously necessary, because most verification problems are undecidable for programs with infinite types, such as integers.

Organisation of this Paper. After defining the target language in Sect. 2, we present the syntax and semantics of LTLC in Sect. 3. Section 4 shows examples of properties expressible by LTLC. Section 5 proves the reducibility result. Section 6 gives a brief discussion on other topics. After discussing related work in Sect. 7, Sect. 8 concludes the paper.

2 Target Language

This section describes the target language of this paper, which is a simply-typed call-by-name higher-order functional programming language with recursion.

We assume a set of *base types*, ranged over by b, as well as a set V_b of values for each base type b. We require that $V_b \cap V_{b'} = \emptyset$ whenever $b \neq b'$. Examples of base types are boolean type and (bounded or unbounded) integer type. We assume the set of base types contains the boolean type $\mathtt{Bool}$ and $V_{\mathtt{Bool}} = \{ tt, ff \}$.

We also assume a set of binary operators Op. Each binary operator $op \in Op$ is associated with their *sort* $b_1, b_2 \to b_3$, meaning that it takes two arguments of types b_1 and b_2 and yields a value of type b_3. Examples of binary operations are $+, -, \times$ (of sort $\mathtt{Int}, \mathtt{Int} \to \mathtt{Int}$) and $=_{\mathtt{Int}}$ (of sort $\mathtt{Int}, \mathtt{Int} \to \mathtt{Bool}$).

Most results of this paper are independent of the choice of basic types and operators. The only exception is the decidability of model checking (Theorem 4) for which we assume base types are finite (i.e. V_b is finite for each base type b).

The set of *types* is given by:

$$\tau := \star \mid \sigma \to \tau \qquad\qquad \sigma := b \mid \tau.$$

A type is the unit type $\star$, a base type b or a function type $\sigma \to \tau$. For a technical convenience, the above syntax requires that the return type of a function is not a base type. Therefore a type τ must be of the form $\sigma_1 \to \cdots \to \sigma_n \to \star$ for some $n \geq 0$ and σ_i, $1 \leq i \leq n$. This does not lose generality because this requirement can be fulfilled by applying the CPS translation.

Assume disjoint sets V of variables and F of function names. The set of *expressions* is defined by the following grammar:

$$e \quad := \quad () \mid c \mid x \mid f \mid e_1\, e_2 \mid \mathbf{op} \mid \mathbf{if}$$

where x and f are meta-variables ranging respectively over variables and function names. The expression $()$ is the unit value and $c \in \bigcup_b V_b$ is a constant of a base type. Each binary operation $op \in Op$ has the associated constructor $\mathbf{op}$, which is in CPS (see the type system below) for a technical convenience. We also have a constructor $\mathbf{if}$ of conditional branching.

A *function definition* $\mathcal{P}$ is a finite set of equations of the form $f\, x_1 \ldots x_n = e$, where f is the name of the function, $x_1, \ldots, x_n$ $(n \geq 0)$ are formal parameters and e is the body of the function. We assume that a function definition $\mathcal{P}$ contains at most one equation for each function name f. A *program* is a pair $(\mathcal{P}, e)$ of a function definition and an expression.

We shall consider only well-typed programs. A *type environment* is a finite set of type bindings of the form $x : \sigma$ or $f : \tau$. We shall use Δ for type environments

of function names, and Γ for variables. A *type judgement* is a tuple $\Delta \mid \Gamma \vdash e : \sigma$. The typing rules for expressions are straightforward, e.g.,

$$\frac{f\colon \tau \in \Delta}{\Delta \mid \Gamma \vdash f : \tau} \qquad \frac{}{\Delta \mid \Gamma \vdash \mathbf{if} : \mathtt{Bool} \to \star \to \star \to \star}$$

$$\frac{op \in Op \text{ has sort } b_1, b_2 \to b_3}{\Delta \mid \Gamma \vdash \mathbf{op} : b_1 \to b_2 \to (b_3 \to \star) \to \star}.$$

Some notable points are (1) the then- and else-branches of an if-expression have to be of unit type, and (2) the binary operation $\mathbf{op}\, e_1\, e_2\, e_3$ in CPS takes two arguments $e_1 : b_1$ and $e_2 : b_2$ of base types and a continuation $e_3 : b_3 \to \star$. We say that a function definition $f\, x_1 \ldots x_n = e$ defines a function of type τ under the type environment Δ, written $\Delta \vdash f\, x_1 \ldots x_n = e : \tau$, if $\tau = \sigma_1 \to \cdots \to \sigma_n \to \star$ and $\Delta \mid x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n \vdash e : \star$. Note that the function body e is required to have type $\star$ (this restriction can be fulfilled by η-expanding the definition, which does not change the meaning of a function in the call-by-name setting). A function definition $\mathcal{P} = \{\, f_i\, \widetilde{x}_i = e_i \}_{1 \le i \le m}$ is *well-typed* under $\Delta = \{\, f_1\colon \tau_1, \ldots, f_m\colon \tau_m \,\}$, written $\Delta \vdash \mathcal{P}$, if $\Delta \vdash f_i\, \widetilde{x}_i = e_i : \tau_i$ for every i. A program $(\mathcal{P}, e)$ is *well-typed* if there exists Δ such that $\Delta \vdash \mathcal{P}$ and $\Delta \mid \emptyset \vdash e : \star$.

The operational semantics of the language is fairly straightforward. We define the *small-step reduction relation* $\longrightarrow$ by the following rules:

$$\frac{c_1\ op\ c_2 = c}{\mathbf{op}\, c_1\, c_2\, e \longrightarrow e\, c} \qquad \frac{}{\mathbf{if}\ tt\, e_1\, e_2 \longrightarrow e_1} \qquad \frac{}{\mathbf{if}\ ff\, e_1\, e_2 \longrightarrow e_2}$$

$$\frac{(f\, x_1 \ldots x_n = e) \in \mathcal{P}}{f\, e_1 \ldots e_n \longrightarrow [e_1/x_1, \ldots, e_n/x_n]e} \qquad \frac{}{() \longrightarrow ()}$$

The last rule is an artificial rule, which ensures that every well-typed expression has an infinite reduction sequence, somewhat simplifying some definitions in the next section. We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

If V_b is finite and the equality $=_b$ on b is in Op, the case analysis of values of type b is definable in the language. We write

$$\mathbf{case}\, e\, \mathbf{of}\, c_1 \to e_1 \mid \cdots \mid c_n \to e_n,$$

where $e, c_1, \ldots, c_n : b$ and $e_1, \ldots, e_n : \star$, for the expression

$$\mathbf{if}\, (=_b\ e\, c_1)\, e_1 (\mathbf{if}\, (=_b\ e\, c_2)\, e_2\, (\ldots (\mathbf{if} (=_b\ e\, c_n)\, e_n\, ())\ldots)).$$

Example 1. Recall the example in Introduction, which was written in direct style. By abstracting unimportant details and transforming it into CPS, we obtain the following program:

$$\begin{array}{l}
\mathbf{let}\, g\, r\, k\ =\ r\, (\lambda w.k\, (\lambda x h.h\, (x + w)))\ \mathbf{in} \\
\mathbf{let}\, r\, k\ =\ k\, 0\ \mathbf{in} \\
\mathbf{let}\, p\, x\ =\ ()\ \mathbf{in} \\
\quad g\, r\, (\lambda v_1.g\, r\, (\lambda v_2.v_1\, 1\, (\lambda u_1.v_2\, 2\, (\lambda u_2.p\, (u_1 + u_2)))))
\end{array}$$

Here r is a function reading the value from a file (consisting of only 0 s) and passing it to the continuation k; p is the function that prints the argument (but formally does nothing). This program can be seen as a program in our language,[3] of which functions definitions are given by sequences of **let** and the initial expression e is that in the last line. The type environment for functions is given by

$$r\colon (\mathtt{Int} \to \star) \to \star, \qquad p\colon \mathtt{Int} \to \star,$$
$$g\colon ((\mathtt{Int} \to \star) \to \star) \to ((\mathtt{Int} \to (\mathtt{Int} \to \star) \to \star) \to \star) \to \star$$

The evaluation of the program is

$$\begin{array}{lllll} g\,r\,k_1 & \longrightarrow & r\,(\lambda w.k_1\,(\lambda x h.h(x+w))) & \longrightarrow^* & k_1\,(\lambda x h.h(x+0)) \longrightarrow \\ g\,r\,k_2 & \longrightarrow & r\,(\lambda w.k_2\,(\lambda x h.h(x+w))) & \longrightarrow^* & k_2\,(\lambda x h.h(x+0)) \longrightarrow \cdots \end{array}$$

where

$$k_1 = (\lambda v_1.g\,r\,(\lambda v_2.v_1\,1\,(\lambda u_1.v_2\,2\,(\lambda u_2.p\,(u_1+u_2)))))$$
$$k_2 = [(\lambda x h.h(x+0))/v_1]\,(\lambda v_2.v_1\,1\,(\lambda u_1.v_2\,2\,(\lambda u_2.p\,(u_1+u_2)))).$$

Note that r is called twice: The first (resp. the second) call of r is between the first (resp. the second) call of g and the call of the corresponding continuation k_1 (resp. k_2).

3 Linear Temporal Logic of Calls

This section defines a novel temporal logic that we call *Linear Temporal Logic of Calls* (*LTLC* for short). It is an extension of the standard linear temporal logic (LTL) by a modal operator, called the *call operator*, which describes a property on function calls. Let $\mathcal{P}$ be a function definitions and Δ be the type environment for functions, fixed in the sequel.

3.1 Syntax

Assume a set L of *variables* that is disjoint from the sets of function names and of variables in expressions. We use α, β and γ for variables in L. The set of *LTLC formulas* is defined by the following grammar:

$$\varphi := \mathit{true} \mid \mathit{false} \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \varphi\,\mathbf{R}\,\varphi \mid \mathit{call}\,\xi(\widetilde{\beta}).\varphi \mid p(\widetilde{\beta})$$

where ξ is either a function name f or a variable $\alpha \in L$. It is the standard LTL with next $\bigcirc$, (strong) until $\mathbf{U}$ and release $\mathbf{R}$ extended by the *call operator* $\mathit{call}\,\xi(\widetilde{\beta}).\varphi$ and predicates $p(\widetilde{\beta})$ on values of base types (such as the order $<$

[3] Strictly speaking, we need to do lambda-lifting as the lambda abstraction is not in the syntax.

and equivalence $=$ of integers). The occurrences of variables $\widetilde{\beta}$ in $\mathit{call}\,\xi(\widetilde{\beta}).\varphi$ are binding occurrences, and ξ is free.

The meaning of formulas should be clear except for $\mathit{call}\,\xi(\widetilde{\beta}).\varphi$. Intuitively $\mathit{call}\,f(\widetilde{\beta}).\varphi$ is true just if the current expression is of the form $f\,\widetilde{e}$ and $\varphi\{\widetilde{e}/\widetilde{\beta}\}$ holds in the next step $e_f\{\widetilde{e}/\widetilde{x}\}$ (where $f\,\widetilde{x} = e_f \in \mathcal{P}$), although here is a subtle point that we shall discuss in the next subsection.

Each variable β in a formula naturally has its type, and a formula should respect the types to make sense. For example, $\mathit{call}\,f(\beta).\varphi$ would be nonsense if the function f has two arguments. We use a type system to filter out such meaningless formulas. We write Θ for a type environment for variables in L. A type judgement is of the form $\Delta \mid \Theta \vdash \varphi$, meaning that φ is well-formed under Δ and Θ. Here Δ and Θ declares the types for function names and variables in L, respectively. Examples of typing rules are as follows:

$$\frac{}{\Delta \mid \Theta \vdash \mathit{false}} \qquad \frac{\Delta \mid \Theta \vdash \varphi}{\Delta \mid \Theta \vdash \bigcirc\varphi} \qquad \frac{\Delta \mid \Theta \vdash \varphi_1 \qquad \Delta \mid \Theta \vdash \varphi_2}{\Delta \mid \Theta \vdash \varphi_1\,\mathbf{U}\,\varphi_2}$$

$$\frac{f : \sigma_1 \to \cdots \to \sigma_n \to \star \in \Delta \qquad \Delta \mid \Theta \cup \{\beta_i : \sigma_i\}_{1 \le i \le n} \vdash \varphi}{\Delta \mid \Theta \vdash \mathit{call}\,f(\beta_1, \ldots, \beta_n).\varphi}.$$

We shall use the following abbreviations. As usual, the temporal operators "future" $\mathbf{F}$ and "always" $\mathbf{G}$ are introduced as derived operators, defined by

$$\mathbf{F}\varphi := \mathit{true}\,\mathbf{U}\,\varphi \qquad \text{and} \qquad \mathbf{G}\varphi := \mathit{false}\,\mathbf{R}\,\varphi.$$

We also use a derived operator *ifcall* given by

$$\mathit{ifcall}\,\xi(\widetilde{\beta}).\varphi \quad := \quad \neg\mathbf{F}\,\mathit{call}\,\xi(\widetilde{\beta}).(\neg\varphi)$$

meaning that $\mathit{call}\,\xi(\widetilde{\beta}).\varphi$ holds for every call of ξ in the future. This operator can alternatively be defined by

$$\mathit{ifcall}\,\xi(\widetilde{\beta}).\varphi \quad = \quad \mathbf{G}(\mathit{call}\,\xi(\widetilde{\beta}).\varphi \vee \neg\mathit{call}\,\xi(\widetilde{\beta}).\mathit{true}).$$

We write $\bigcirc^n\varphi$ for $\underbrace{\bigcirc\cdots\bigcirc}_{n}\varphi$, meaning that φ holds after n steps.

Example 2. Recall the program in Example 1. The formula meaning "g does not call its first argument after returning the value" can be written as follows:

$$\mathit{ifcall}\,g(\alpha, \beta)\;.\;\mathit{ifcall}\,\beta(\gamma)\;.\;(\neg\mathbf{F}\,\mathit{call}\,\alpha(\delta)\;.\;\mathit{true}).$$

Since the programs are now written in CPS, "returning the value" means "calling the continuation β." The above formula says that, for every call of g, if it returns the value (via the continuation β), then it will never call the first argument α.

3.2 Naïve Semantics and a Problem

Every closed expression $\Delta \mid \emptyset \vdash e : \star$, possibly using functions in $\mathcal{P}$, induces the unique infinite reduction sequence

$$e = e_0 \longrightarrow e_1 \longrightarrow e_2 \longrightarrow \ldots \longrightarrow e_n \longrightarrow \cdots .$$

An LTLC formula φ describes a property on such infinite reduction sequences. Thus the *satisfaction relation* $e \models \varphi$ is defined as a relation between an expression $\Delta \vdash e : \star$ and an LTLC formula $\Delta \vdash \varphi$.

The definition of the relation for logical connectives from the standard LTS is straightforward. For example, $\bigcirc\varphi$ means that φ holds in the next step, and thus $e \models \bigcirc\varphi$ if and only if $e' \models \varphi$ for the unique e' such that $e \longrightarrow e'$.

The main issue is how to define the semantics of $\mathit{call}\, f(\widetilde{\beta}).\varphi$. Intuitively $e \models \mathit{call}\, f(\beta_1, \ldots, \beta_n).\varphi$ holds if and only if $e = f\, e_1 \ldots e_n$ and $e_f[e_1/x_1, \ldots, e_n/x_n] \models \varphi[e_1/\beta_1, \ldots, e_n/\beta_n]$, where e_f is the body of the definition of f. However this naïve definition has a problem.

Let us explain the problem by using an example. Consider the function *doTask* defined by

$$\mathit{doTask}\, f\, g \quad = \quad f\,(g\,()).$$

It would be natural to expect that "*doTask* does not call the second argument unless it does not call the first argument" should be true independent of the context in which *doTask* is used. Formally we expect $C[\mathit{doTask}] \models \varphi$ for every context C, where φ is the formula given by

$$\varphi \quad = \quad \mathit{call}\, \mathit{doTask}(\alpha, \beta)\ .\ \big((\neg\, \mathit{call}\, \beta(\gamma).\mathit{true})\ \mathbf{U}\ (\mathit{call}\, \alpha(\delta).\mathit{true})\big).$$

However it is not true. Consider, for example, $C = []\, h\, h$ where h is an arbitrary function. Then

$$\mathit{doTask}\, h\, h \models \varphi \quad \Leftrightarrow \quad h\,(h\,()) \models (\neg\, \mathit{call}\, h(\gamma).\mathit{true})\ \mathbf{U}\ (\mathit{call}\, h(\delta).\mathit{true})$$

but the right-hand-side is false because $h\,(h\,()) \models \mathit{call}\, h(\gamma).\mathit{true}$ and thus $h\,(h\,()) \not\models \neg\, \mathit{call}\, h(\gamma).\mathit{true}$.

The problem is caused by confusion between h as the first argument and h as the second argument. In the formula φ, the first and second arguments of *doTask* are distinguished by their names, α and β. However they become indistinct by the substitution $[h/\alpha, h/\beta]$.

3.3 Formal Semantics

We use labels to correctly keep track of expressions. A *label* is just a variable $\alpha \in L$ in a formula. *Labelled expressions* are those obtained by extending expressions with the *labelling* construct, as follows:

$$e ::= \cdots \mid e^{\alpha}, \qquad \alpha \in L.$$

$$
\begin{array}{lcl}
e, \rho \models \mathit{true} & & \text{always holds} \\
e, \rho \models \mathit{false} & & \text{never holds} \\
e, \rho \models \neg\varphi & \iff & e, \rho \not\models \varphi \\
e, \rho \models \varphi_1 \vee \varphi_2 & \iff & e, \rho \models \varphi_1 \text{ or } e, \rho \models \varphi_2 \\
e, \rho \models \varphi_1 \wedge \varphi_2 & \iff & e, \rho \models \varphi \text{ and } e, \rho \models \varphi_2 \\
e, \rho \models \bigcirc\varphi & \iff & (\exists e')[e \longrightarrow e' \text{ and } e', \rho \models \varphi] \\
e, \rho \models \varphi_1 \,\mathbf{U}\, \varphi_2 & \iff & (\exists j)[e, \rho \models \bigcirc^j \varphi_2 \text{ and } (\forall i < j)[e, \rho \models \bigcirc^i \varphi_1]] \\
e, \rho \models \varphi_1 \,\mathbf{R}\, \varphi_2 & \iff & (\forall j)[e, \rho \models \bigcirc^j \varphi_2 \text{ or } (\exists i < j)[e, \rho \models \bigcirc^i \varphi_1]] \\
e, \rho \models p(\alpha_1, \ldots, \alpha_k)a & \iff & p(\natural\rho(\alpha_1), \ldots, \natural\rho(\alpha_k)) \text{ is true} \\
e, \rho \models \mathit{call}\, \alpha(\beta_1, \ldots, \beta_k).\varphi & \iff & e = (\ldots (e_0^{S_0}\, e_1)^{S_1} \ldots e_k)^{S_k} \text{ and } \alpha \in S_0 \text{ and} \\
 & & e_0\, e_1^{\beta_1} \ldots e_k^{\beta_k} \longrightarrow e' \text{ and } e', \rho \cup \{\beta_i \mapsto e_i\}_{1 \le i \le k} \models \varphi \\
e, \rho \models \mathit{call}\, f(\beta_1, \ldots, \beta_k).\varphi & \iff & e = (\ldots (f^{S_0}\, e_1)^{S_1} \ldots e_k)^{S_k} \text{ and} \\
 & & e_0\, e_1^{\beta_1} \ldots e_k^{\beta_k} \longrightarrow e' \text{ and } e', \rho \cup \{\beta_i \mapsto e_i\}_{1 \le i \le k} \models \varphi
\end{array}
$$

Fig. 1. Semantics of formulas. The operation $\natural$ removes labels from a given expression.

For a possibly empty sequence $S = \alpha_1 \ldots \alpha_n$, we write e^S for $((e^{\alpha_1}) \ldots)^{\alpha_n}$. Given a labelled expression e, we write $\natural e$ for the (ordinary) expression obtained by removing labels in e.

The labels do not affect reduction. For example,

$$(((f^{S_0}\, e_1)^{S_1}\, e_2)^{S_2} \ldots\, e_n)^{S_n} \quad \longrightarrow \quad e_f[e_1/x_1, \ldots, e_n/x_n]$$

provided that $f\, x_1 \ldots x_n = e_f \in \mathcal{P}$. Therefore, if $e \longrightarrow e'$ as labelled expressions, then $\natural e \longrightarrow \natural e'$.

Now we formally define the satisfaction relation $\models$. It is a ternary relation $e, \rho \models \varphi$ on a labelled expression $e : \star$, a valuation map ρ from free variables in φ to labelled expressions, and an LTLC formula φ. It is defined by induction on the complexity[4] of formulas by the rules in Fig. 1.

Remark 1. Given a judgement $e, \rho \models \varphi$, one can remove the following data without changing the meaning of the judgement:

- mapping $\alpha \mapsto e$ from ρ if α is not of a base type, and
- label β in $(d)^\beta$ from e if d is an expression of a base type.

This is because the information on a base-type variable β is recorded in ρ, and the information on a non-base-type variable α is tracked by labels in the expression. We put both information to both ρ and e just to simplify the definition (by avoiding the case split by types).

The main difference from the naïve semantics is the meaning of the call operator. Instead of substituting β_i in the formula to the actual argument e_i in the expression, we annotate the actual argument e_i by β_i.

[4] We define the *complexity* of a formula φ as the pair of numbers (n, m) ordered by the lexicographic ordering, where n is the sum of the numbers of occurrences of $\mathbf{U}$ and $\mathbf{R}$ in φ and m is the size of φ.

We see how the labelling works by using the example in the previous subsection. By the labelling semantics, we have

$$(\mathit{doTask}\, h\, h), \emptyset \models \varphi \quad \Leftrightarrow \quad h^{\alpha}\,(h^{\beta}\,()), \rho \models (\neg\, \mathit{call}\, \beta(\gamma).\mathit{true})\, \mathbf{U}\, (\mathit{call}\, \alpha(\delta).\mathit{true})$$

for some ρ (whose contents are not important here). Notice that h as the first argument of *doTask* can be distinguished from h as the second argument of *doTask*: the former has the label α whereas the latter is annotated by β. Now $h^{\alpha}\,(h^{\beta}\,()), \rho \not\models \mathit{call}\, \beta(\gamma).\mathit{true}$ and $h^{\alpha}\,(h^{\beta}\,()), \rho \models \neg \mathit{call}\, \beta(\gamma).\mathit{true}$ as expected. It is not difficult to see that $\mathit{doTask}\, h\, h, \emptyset \models \varphi$ indeed holds, whatever h is.

3.4 Negation Normal Form

The negation $\neg$ in a formula can be pushed inwards in many cases, without changing the meaning of the formula. For example,

$$\neg \mathit{true} = \mathit{false} \qquad \neg(\varphi_1\, \mathbf{U}\, \varphi_2) = (\neg\varphi_1)\, \mathbf{R}\, (\neg\varphi_2) \quad \text{and} \quad \neg\bigcirc\varphi = \bigcirc\neg\varphi.$$

Unfortunately the negation of the call operator $\neg \mathit{call}\, \xi(\widetilde{\beta}).\varphi$ cannot be pushed inwards in general, but we can restrict the shape of the formula to which the negation is applied. The formula $\mathit{call}\, \xi(\widetilde{\beta}).\varphi$ does *not* hold if either (a) ξ is now called but the following computation violates φ or (b) ξ is not called in the current step. This observation can be expressed by the equation

$$\neg \mathit{call}\, \xi(\widetilde{\beta}).\varphi \quad = \quad \mathit{call}\, \xi(\widetilde{\beta}).(\neg\varphi) \;\vee\; \neg \mathit{call}\, \xi(\widetilde{\beta}).\mathit{true}.$$

We shall abbreviate $\neg \mathit{call}\, \xi(\widetilde{\beta}).\mathit{true}$ as $\neg \mathit{call}\, \xi$.

The above argument gives an effective rewriting process, yielding a formula in the following syntax that we call the *negation normal form*:

$$\begin{aligned} \varphi := {} & \mathit{true} \mid \mathit{false} \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\, \mathbf{U}\, \varphi \mid \varphi\, \mathbf{R}\, \varphi \\ & \mid \mathit{call}\, \xi(\widetilde{\beta}).\varphi \mid \neg \mathit{call}\, \xi \mid p(\widetilde{\beta}) \mid \neg p(\widetilde{\beta}). \end{aligned}$$

We shall use this normal form in the following section.

4 Expressiveness

This section briefly explains the expressiveness of LTLC.

4.1 Dependent Refinement Types

Properties described by dependent refinement types in the form of [21] are expressible by LTLC formulas.

Example 3. Consider the type

$$T_0 \quad := \quad (x\colon \{\mathtt{Int} \mid \nu \geq 0\}) \to \{\mathtt{Int} \mid \nu > x\}$$

for call-by-value programs. This is the type for functions f on integers such that $f(x) > x$ for every positive x. As the target language of this paper is call-by-name, we need to apply the CPS translation to call-by-value programs of interest and the corresponding translation to dependent types. The resulting type is

$$T \quad := \quad (x\colon \{\mathtt{Int} \mid \nu \geq 0\}) \to \big(\{\mathtt{Int} \mid \nu \geq x\} \to \star\big) \to \star.$$

The LTLC formula φ_T corresponding to the judgement $\vdash f : T$ is

$$\varphi_T \quad := \quad \mathit{ifcall}\, f(\alpha, \beta)\ .\ \big(\alpha \geq 0 \Rightarrow \mathit{ifcall}\, \beta(\gamma)\ .\ \alpha < \gamma\big).$$

We explain the general rule of the translation, focusing on the image of function types by the call-by-value CPS translation. The syntax of dependent refinement types is given by

$$T, S \ ::= \ (\alpha : U) \to \big(V \to \star\big) \to \star \qquad U, V \ ::= \ \{\mathtt{Int} \mid \vartheta(\nu)\} \quad | \quad T$$

where ν is a distinguished variable and $\vartheta(\nu)$ is a formula of the underlying logic. The occurrence of α is a binding occurrence and ϑ may contain variables other than ν. The LTLC formula Φ_U is defined by the following rules:

$$\begin{aligned} \Phi_{\{\mathtt{Int} \mid \vartheta(\nu)\}}(\alpha) &:= \vartheta(\alpha) \\ \Phi_{(\beta:U)\to(V\to\star)\to\star}(\alpha) &:= \mathit{ifcall}\, \alpha(\beta, \kappa).\ \big(\Phi_U(\beta) \Rightarrow \mathit{ifcall}\, \kappa(\gamma)\ .\ \Phi_V(\gamma)\big). \end{aligned}$$

A judgement $\vdash f : T$ corresponds to the LTLC formula $\Phi_T(f)$.

4.2 Relational Property

Some *relational properties* [1,5], such as the relationship between two functions and that between two calls of a function, can be described by LTLC. An example of relational property is monotonicity; if a given function f is not monotone, one can find two inputs $x \leq y$ such that $f(x) \not\leq f(y)$. Monotonicity can be naturally expressed by LTLC.

Example 4 (Monotonicity). Assume a function $f\colon \mathtt{Int} \to (\mathtt{Int} \to \star) \to \star$. This function is *monotone* if $n \leq n'$, $f\, n\, k$ calls the continuation k with the value m and $f\, n'\, k'$ calls k' with m', then $m \leq m'$. (Recall that f is in CPS and thus "calling k with m" can be understood as "returning m".) If f is assumed to be non-recurrent, this property can be written as

$$\mathit{ifcall}\, f(\alpha, \beta)\ .\ \mathit{ifcall}\, \beta(\gamma)\ .\ \mathit{ifcall}\, f(\alpha', \beta')\ .\ \mathit{ifcall}\, \beta'(\gamma')\ .\ \psi(\alpha, \gamma, \alpha', \gamma')$$

where $\psi(\alpha, \gamma, \alpha', \gamma') = (\alpha \leq \alpha' \Rightarrow \gamma \leq \gamma') \wedge (\alpha \geq \alpha' \Rightarrow \gamma \geq \gamma')$. The meaning of this formula can be expressed by a natural language as follows:

Let α be the argument to the first call of f, and γ be the "return value" of the first call. Similarly let α' be the argument to the second call of f, and γ' be the "return value" of the second call. We require that $\alpha \leq \alpha'$ implies $\gamma \leq \gamma'$, and that $\alpha \geq \alpha'$ implies $\gamma \geq \gamma'$.

A formula applicable to the case of f being recurrent is a bit complicated, since the order of two calls and returns is not determined. The formula applicable to the general case is

$$\begin{aligned} & \mathit{ifcall}\, f(\alpha, \beta)\, .\, \mathit{ifcall}\, f(\alpha', \beta')\, .\, \mathit{ifcall}\, \beta(\gamma)\, .\, \mathit{ifcall}\, \beta'(\gamma')\, .\, \psi(\alpha, \gamma, \alpha', \gamma') \\ \wedge\ & \mathit{ifcall}\, f(\alpha, \beta)\, .\, \mathit{ifcall}\, f(\alpha', \beta')\, .\, \mathit{ifcall}\, \beta'(\gamma')\, .\, \mathit{ifcall}\, \beta(\gamma)\, .\, \psi(\alpha, \gamma, \alpha', \gamma') \\ \wedge\ & \mathit{ifcall}\, f(\alpha, \beta)\, .\, \mathit{ifcall}\, \beta(\gamma)\, .\, \mathit{ifcall}\, f(\alpha', \beta')\, .\, \mathit{ifcall}\, \beta'(\gamma')\, .\, \psi(\alpha, \gamma, \alpha', \gamma') \end{aligned}$$

The conjunction enumerates all possible orders of two calls and returns of f.

4.3 Resource Usage Verification

The final example is verification/analysis of programs using resources, known as *resource usage analysis* [10]. An example of resource is read-only files. For simplicity, we focus on the verification of usage of read-only files.

Let us first consider the simplest case in which a program generates a unique resource only at the beginning. In this case, a target is a program with distinguished functions $r, c : \star \to \star$ for reading and closing the file. The specification requires (1) the program does not read the file after closing it, and (2) the file must be closed before the termination of the program. The specification can be described by an LTLC formula:

$$\varphi(r, c) \quad := \quad \mathbf{G}(\mathit{call}\, c \Rightarrow \neg \mathbf{F}\, \mathit{call}\, r) \wedge (\neg \mathtt{end}\ \mathbf{U}\ \mathit{call}\, c),$$

where `end` is the event meaning the termination. Indeed this is an LTL formula when one regards $\mathit{call}\, c$ and $\mathit{call}\, r$ as atomic propositions.

In the general case, a program can dynamically create read-only file resources. The target program has a distinguished type `File` for file resources and a distinguished function $\mathit{gen} : (\mathtt{File} \to \star) \to \star$. Since the possible operations for `File` is read and close, we identify the type `File` as $(\star \to \star) \times (\star \to \star)$, the pair of reading and closing functions. The specification requires that, for each call of gen, the created resource should be used in the manner following φ; this specification can be written by an LTLC formula as

$$\mathit{ifcall}\, \mathit{gen}(\alpha)\, .\, \mathit{ifcall}\, \alpha(r, c)\, .\, \varphi(r, c).$$

Note that $\mathit{ifcall}\, \alpha(r, c)$ intuitively means that "if the function gen returns the value (r, c)" since gen is in CPS and α is the continuation.

5 LTLC Model Checking

This section focuses on the LTLC model-checking problem, i.e. the problem to decide, given a program $\mathcal{P}$, an expression e and an LTLC formula φ, whether

$e, \emptyset \models \varphi$ (where $\emptyset$ is the empty valuation). The main result of this section is that the LTLC model-checking problem is effectively reducible to the standard temporal verification problem, which could be solved by model checkers for higher-order programs. In particular, for programs and expressions over finite types, this reduction yields an instance of higher-order model checking [12,17], for which several model-checkers are available [6,20].

5.1 Preliminaries: Higher-Order Model Checking

Higher-order model checking is a problem to decide, given a higher-order tree grammar and an ω-regular tree property, whether the (possibly infinite) tree generated by the grammar satisfies the property. Higher-order model checking has been proved to be decidable by Ong [17][5] and applied to many verification problems of higher-order programs (see, e.g., [12]). We prove that LTLC model checking is decidable by reducing it to higher-order model checking.

This subsection briefly reviews higher-order model checking, tailored to our purpose. See [12,17] for formal definitions and general results.

Let Σ be a ranked alphabet defined by

$$\Sigma := \{ \top, \bot \mapsto 0,\ \sqcup, \sqcap \mapsto 2,\ \mathtt{U}, \mathtt{R} \mapsto 3 \}.$$

This means that $\top$ is a leaf and $\sqcup$ (resp. $\mathtt{U}$) is a binary (resp. ternary) branching tree constructor, and so on. A *Σ-labelled tree* is a possibly infinite tree of which each node is labelled by a symbol in Σ. We shall consider only *well-ranked trees*: we require that the number of children of a node labelled by $\sqcap$ is 2, for example. We shall often use the infix notation for $\sqcup$ and $\sqcap$, e.g. $T_1 \sqcup T_2$ is the tree whose root is $\sqcup$ and its children are T_1 and T_2.

A *nondeterministic Büchi automaton* is a tuple (Q, q_0, δ, F), where Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, $\delta\colon \prod_{a \in \mathrm{dom}(\Sigma)} (Q \to \mathcal{P}(Q^{\Sigma(a)}))$ is a *transition function* and $F \subseteq Q$ is the set of *accepting states*. A *run-tree* of $\mathcal{A}$ over a tree T is an association of states $q \in Q$ to nodes in T that respects the transition function in a certain sense. A run-tree is *accepting* if each infinite branch contains infinitely many occurrences of an accepting state. A tree T is *accepted* by $\mathcal{A}$ if there is an accepting run-tree over T.

A *tree-generating program* is a variant of programs introduced in Sect. 2, but has different set of operators on type $\star$. The syntax of expressions is

$$e \quad := \quad \top \mid \bot \mid \sqcup \mid \sqcap \mid \mathtt{U} \mid \mathtt{R} \mid c \mid x \mid f \mid e_1\, e_2 \mid \mathbf{op}\, e_1\, e_2\, e_3 \mid \mathbf{if}\, e_1\, e_2\, e_3,$$

obtained by replacing () with the tree constructors in Σ. Their types are

$$\top, \bot\colon \star \qquad \sqcup, \sqcap\colon \star \to \star \to \star \qquad \text{and} \qquad \mathtt{U}, \mathtt{R}\colon \star \to \star \to \star \to \star.$$

[5] The original definition (as in [17]) considers only programs without data types, but the decidability result can be easily extended to programs with finite data types. We shall consider a generalised version, in which programs may contain infinite data types. Of cause, the decidability result fails for the generalised version.

So $\star$ should be now regarded as the type for *trees*. The notion of function definition remains unchanged, except that the body of a function is now an expression with tree constructors.

The operational semantics is basically the same as before. The only difference is that reduction may occur under tree constructors, i.e. the following rules

$$\frac{e_1 \longrightarrow e_1'}{(e_1 \sqcup e_2) \longrightarrow (e_1' \sqcup e_2)} \quad \text{and} \quad \frac{e_2 \longrightarrow e_2'}{(e_1 \sqcup e_2) \longrightarrow (e_1 \sqcup e_2')}$$

are added, as well as similar rules for other constructors. A program $(\mathcal{P}, e)$ generates a possibly infinite tree as a result of possibly infinite steps of reduction.

Higher-order model checking asks to decide, given a program $(\mathcal{P}, e)$ and a nondeterministic Büchi automaton $\mathcal{A}$, whether the tree generated by $(\mathcal{P}, e)$ is accepted by $\mathcal{A}$.

Theorem 1 (Ong [17]). *Given a tree-generating program $(\mathcal{P}, e)$, of which all basic data types are finite, and a nondeterministic Büchi automaton $\mathcal{A}$, one can effectively decide whether the tree generated by $(\mathcal{P}, e)$ is accepted by $\mathcal{A}$.*

5.2 Satisfaction Tree

Let $\mathcal{P}$ be a function definition, fixed in this subsection. Given an expression $e : \star$, a valuation ρ and an LTLC formula φ in negation normal form, we define a tree $\mathcal{T}(\varphi, \rho, e)$, called the *satisfaction tree*, which represents the process evaluating $e, \rho \models \varphi$. This subsection shows that the satisfaction tree correctly captures the satisfaction relation, in the sense that $e, \rho \models \varphi$ if and only if $\mathcal{T}(\varphi, \rho, e)$ belongs to a certain ω-regular tree language.

A satisfaction tree is a Σ-labelled tree. The meaning of $\top, \bot, \sqcup$ and $\sqcap$ should be obvious. The trees $\top$ and $\bot$ represent immediate truth and falsity. The tree $T_1 \sqcap T_2$ means that the evaluation process invokes two subprocess, represented by T_1 and T_2, and the result is true just if the results of both subprocesses are true. The meaning of $\sqcup$ is similar.

The constructors `U` and `R`, corresponding respectively to $\mathbf{U}$ and $\mathbf{R}$, require some expositions. The meaning of `U` is based on a classical but important observation: whether $e, \rho \models \varphi_1 \,\mathbf{U}\, \varphi_2$ or not is completely determined by three judgements, namely $e, \rho \models \varphi_1$, $e, \rho \models \varphi_2$ and $e, \rho \models \bigcirc(\varphi_1 \,\mathbf{U}\, \varphi_2)$. That means, $e, \rho \models \varphi_1 \,\mathbf{U}\, \varphi_2$ if and only if either (a) $e, \rho \models \varphi_1$ and $e, \rho \models \varphi_2$, or (b) $e, \rho \models \varphi_1$ and $e, \rho \models \bigcirc(\varphi_1 \,\mathbf{U}\, \varphi_2)$. So the process of checking $e, \rho \models \varphi_1 \,\mathbf{U}\, \varphi_2$ invokes three subprocesses; the three subtrees of `U` correspond to these judgements. A similar observation applies to $\mathbf{R}$.

The definition of satisfaction trees is co-inductively defined by the rules in Figs. 2, 3, 4 and 5. The meaning of the rules in Fig. 2 should now be clear. For example, the rule for $\varphi_1 \,\mathbf{U}\, \varphi_2$ says that $e, \rho \models \varphi_1 \,\mathbf{U}\, \varphi_2$ depends on satisfaction of $e, \rho \models \bigcirc(\varphi_1 \,\mathbf{U}\, \varphi_2)$, $e, \rho \models \varphi_1$ and $e, \rho \models \varphi_2$.

Figure 3 defines the rules for the call modality $\mathit{call}\, \alpha(\widetilde{\beta}).\varphi$. The first two rules check if e is calling an expression labelled by α. If one finds the label α, then

$$
\begin{aligned}
\mathcal{T}[e, \rho \models \mathit{true}] &:= \top \\
\mathcal{T}[e, \rho \models \mathit{false}] &:= \bot \\
\mathcal{T}[e, \rho \models \varphi_1 \vee \varphi_2] &:= \mathcal{T}[e, \rho \models \varphi_1] \sqcup \mathcal{T}[e, \rho \models \varphi_2] \\
\mathcal{T}[e, \rho \models \varphi_1 \wedge \varphi_2] &:= \mathcal{T}[e, \rho \models \varphi_1] \sqcap \mathcal{T}[e, \rho \models \varphi_2] \\
\mathcal{T}[e, \rho \models \varphi_1 \,\mathbf{U}\, \varphi_2] &:= \mathbf{U}(\mathcal{T}[e, \rho \models \bigcirc(\varphi_1 \,\mathbf{U}\, \varphi_2)],\ \mathcal{T}[e, \rho \models \varphi_1],\ \mathcal{T}[e, \rho \models \varphi_2]) \\
\mathcal{T}[e, \rho \models \varphi_1 \,\mathbf{R}\, \varphi_2] &:= \mathbf{R}(\mathcal{T}[e, \rho \models \bigcirc(\varphi_1 \,\mathbf{R}\, \varphi_2)],\ \mathcal{T}[e, \rho \models \varphi_1],\ \mathcal{T}[e, \rho \models \varphi_2]) \\
\mathcal{T}[e, \rho \models p(\alpha_1, \ldots, \alpha_n)] &:= \begin{cases} \top & (\text{if } p(\natural\rho(\alpha_1), \ldots, \natural\rho(\alpha_n)) \text{ is true}) \\ \bot & (\text{if } p(\natural\rho(\alpha_1), \ldots, \natural\rho(\alpha_n)) \text{ is false}) \end{cases} \\
\mathcal{T}[e, \rho \models \neg p(\alpha_1, \ldots, \alpha_n)] &:= \begin{cases} \top & (\text{if } p(\natural\rho(\alpha_1), \ldots, \natural\rho(\alpha_n)) \text{ is false}) \\ \bot & (\text{if } p(\natural\rho(\alpha_1), \ldots, \natural\rho(\alpha_n)) \text{ is true}) \end{cases}
\end{aligned}
$$

Fig. 2. Satisfaction tree: (1) Boolean connectives, until and release.

the arguments are recorded to ρ and labelled by $\widetilde{\beta}$ as required. In this case, we also change the target formula to $\bigcirc\varphi$. In the second rule, a label other than α should be simply ignored. The last three rules deal with the case of α not being annotated; then $e, \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi$ is immediately false.

$$
\begin{aligned}
\mathcal{T}[e^{\alpha}\, e_1\, \ldots\, e_n,\ \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi] &:= \mathcal{T}[e\, e_1^{\beta_1}\, \ldots\, e_n^{\beta_n},\ \rho \cup \{\beta_i \mapsto e_i\}_{1 \le i \le n} \models \bigcirc\varphi] \\
\mathcal{T}[e^{\gamma}\, e_1\, \ldots\, e_n,\ \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi] &:= \mathcal{T}[e\, e_1\, \ldots\, e_n,\ \rho \models \mathit{call}\, \ell(\widetilde{x}).\varphi] \\
\mathcal{T}[f\, e_1\, \ldots\, e_n,\ \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi] &:= \bot \\
\mathcal{T}[\mathbf{if}\, e_1\, e_2\, e_3,\ \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi] &:= \bot \\
\mathcal{T}[\mathbf{op}\, e_1\, e_2\, e_3,\ \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi] &:= \bot \\
\mathcal{T}[(), \rho \models \mathit{call}\, \alpha(\widetilde{\beta}).\varphi] &:= \bot
\end{aligned}
$$

Fig. 3. Satisfaction tree: (2) Call modality. We assume $\gamma \neq \alpha$. The satisfaction tree for $\mathit{call}\, f(\widetilde{\beta}).\varphi$ is similar; we omit the rules here.

Figure 4 defines the rules for the negation of call. If e is calling an expression labelled by α, then $e, \rho \models \neg \mathit{call}\, \alpha(\_)$ is obviously false. The last three rules describe the case of α not being found, in which case $\neg \mathit{call}\, \alpha(\_)$ holds.

Figure 5 defines the rules for the next modality. It simply ignores labels and reduces the expression in one step.

We omit the rules for $\mathit{call}\, f(\widetilde{\beta}).\varphi$ and $\neg \mathit{call}\, f(\_)$, which are basically the same as those for $\mathit{call}\, \alpha(\widetilde{\beta}).\varphi$ and $\neg \mathit{call}\, \alpha(\_)$.

We formalise the meaning of a satisfaction tree by giving a nondeterministic Büchi automaton. The definition of the automaton is basically straightforward, but there is a subtlety in the meaning of `U`. Recall that $\varphi_1 \,\mathbf{U}\, \varphi_2$ holds if either

$$
\begin{aligned}
\mathcal{T}[e^{\alpha}\, e_1\, \ldots\, e_n,\ \rho \models \neg\, \mathit{call}\, \alpha(\_)] &:= \bot \\
\mathcal{T}[e^{\gamma}\, e_1\, \ldots\, e_n,\ \rho \models \neg\, \mathit{call}\, \alpha(\_)] &:= \mathcal{T}[e\, e_1\, \ldots\, e_n,\ \rho \models \neg \mathit{call}\, \ell(\_)] \\
\mathcal{T}[f\, e_1\, \ldots\, e_n,\ \rho \models \neg \mathit{call}\, \alpha(\_),\ \rho,] &:= \top \\
\mathcal{T}[\mathbf{if}\, e_1\, e_2\, e_3,\ \rho \models \neg \mathit{call}\, \alpha(\_)] &:= \top \\
\mathcal{T}[\mathbf{op}\, e_1\, e_2\, e_3,\ \rho \models \neg \mathit{call}\, \alpha(\_)] &:= \top \\
\mathcal{T}[(), \rho \models \neg \mathit{call}\, \alpha(\_)] &:= \top
\end{aligned}
$$

Fig. 4. Satisfaction tree: (3) Negation of call modality. We assume $\gamma \neq \alpha$. The satisfaction tree for $\neg \mathit{call}\, f(\_)$ is similar; we omit the rules here.

$$
\begin{aligned}
\mathcal{T}[e^{\alpha}\, e_1\, \ldots\, e_n,\ \rho \models \bigcirc\varphi] &:= \mathcal{T}[e\, e_1\, \ldots\, e_n,\ \rho \models \bigcirc\varphi] \\
\mathcal{T}[f\, e_1\, \ldots\, e_n,\ \rho \models \bigcirc\varphi] &:= \mathcal{T}[e_f[e_1/x_1, \ldots, e_n/e_n],\ \rho \models \varphi] \\
\mathcal{T}[\mathbf{if}\ \mathit{tt}\, e\, e,\ \rho \models \bigcirc\varphi] &:= \mathcal{T}[e, \rho \models \varphi] \\
\mathcal{T}[\mathbf{if}\ \mathit{ff}\, e\, e',\ \rho \models \bigcirc\varphi] &:= \mathcal{T}[e', \rho \models \varphi] \\
\mathcal{T}[\mathbf{op}\, e_1\, e_2\, e_3,\ \rho \models \bigcirc\varphi] &:= \mathcal{T}[e_3\, c',\ \rho \models \varphi] \qquad \text{where } c' = (\natural e_1)\ \mathit{op}\ (\natural e_2) \\
\mathcal{T}[(), \rho \models \bigcirc\varphi] &:= \mathcal{T}[(), \rho \models \varphi]
\end{aligned}
$$

Fig. 5. Satisfaction tree: (4) Next modality. It ignores labels and reduces the expression in one step. We assume that $(f\, x_1\, \ldots\, x_n = e_f) \in \mathcal{P}$.

1. both φ_1 and φ_2 hold, or
2. both φ_1 and $\bigcirc(\varphi_1\, \mathbf{U}\, \varphi_2)$ hold.

Similarly $\varphi_1\, \mathbf{R}\, \varphi_2$ holds if and only if

1. both φ_1 and φ_2 hold, or
2. both φ_2 and $\bigcirc(\varphi_1\, \mathbf{U}\, \varphi_2)$ hold.

The condition for **U** quite resembles that for **R**, but there is a crucial difference which cannot be captured by the above descriptions. That is, $\varphi_1\, \mathbf{U}\, \varphi_2$ requires that φ_2 eventually holds, but $\varphi_1\, \mathbf{R}\, \varphi_2$ is true even if φ_1 never becomes true. This difference should be captured by the acceptance condition of the Büchi automaton.

The Büchi automaton $\mathcal{A}$ has three states, q_0, q_1 and $*$. The states q_0 and q_1 have the same behaviour except that q_0 is accepting and q_1 is not accepting. The state $*$ accepts every tree; this state is used to describe a rule which ignores some of children. The set of accepting states is $\{q_0, *\}$ and the initial state is q_0. The transition rules are given by:

$$
\begin{aligned}
\delta_{\top}(q) &:= \{()\} & \delta_{\bot}(q) &:= \{()\} \\
\delta_{\sqcap}(q) &:= \{(q_0, q_0)\} & \delta_{\sqcup}(q) &:= \{(q_0, *), (*, q_0)\} \\
\delta_{\mathtt{U}}(q) &:= \{(q_1, q_0, *), (*, q_0, q_0)\} & \delta_{\mathtt{R}}(q) &:= \{(q_0, *, q_0), (*, q_0, q_0)\},
\end{aligned}
$$

where $q = q_0$ or q_1. We omit the rules for the state $*$, which accepts every tree. The tree $\mathtt{U}(T_1, T_2, T_3)$ is accepted from q_0 if T_1 is accepted from q_1 and T_3 is

accepted from q_0; note that we assign q_1 to T_1, instead of q_0, because the until formula $\varphi_1 \mathbf{U} \varphi_2$ expects φ_2 eventually holds.

The following theorem is the first half of the reduction.

Theorem 2. *$e, \rho \models \varphi$ if and only if $\mathcal{T}[e, \rho \models \varphi]$ is accepted by $\mathcal{A}$.*

5.3 A Tree-Generating Program Generating the Satisfaction Tree

The previous subsection introduced satisfaction trees, which concern only about LTL features of LTLC, i.e. the tree does not have any information on the call nor next modality, which are related to reduction of programs.

This subsection discusses a way to deal with these features missing in satisfaction trees. Technically, given a program $(\mathcal{P}, e)$ and a formula φ, we construct a tree-generating program $(\mathcal{P}^\#, e')$ that generates the satisfaction tree $\mathcal{T}[e, \emptyset \models \varphi]$. The construction of $(\mathcal{P}^\#, e')$ is effective, and if the original program and the formula use only finite base types, then so does $(\mathcal{P}^\#, e')$. Therefore this construction, together with the result of the previous subsection, shows that the LTLC model checking is decidable for programs and formulas over finite data types.

We first give the formal statement of the theorem, which shall be proved in the rest of this subsection.

Theorem 3. *Given a program $(\mathcal{P}, e_0)$ and an LTLC formula φ, one can effectively construct a tree-generating program $(\mathcal{P}^\#, e'_0)$ that generates the satisfaction tree $\mathcal{T}[e, \emptyset \models \varphi]$. Furthermore, if both the program $(\mathcal{P}, e)$ and the formula φ contain only finite base types, then so does $(\mathcal{P}^\#, e'_0)$.*

Let φ_0 be a formula of interest, fixed in the sequel. By renaming bound variables if necessary, we can assume without loss of generality that different variables in φ_0 have different names. Let $L_0 \subseteq L$ be the finite set of bound variables in φ_0. Note that each $\alpha \in L_0$ is associated to its type in φ_0.

The idea of the translation, written $\#$, is as follows. Recall that the satisfaction tree $\mathcal{T}[e, \rho \models \varphi]$ is determined by the three data, namely an expression $e : \star$, a valuation ρ and a formula φ. Hence the translation $e^\#$ of the expression e should take two extra arguments ρ and φ to compute $\mathcal{T}[e, \rho \models \varphi]$.

Let us first consider the translation of types. Because the translation of an expression e of unit type $\star$ takes two additional arguments, namely a formula and a valuation, the translation of the unit type should be given by

$$\star \quad \overset{\#}{\longmapsto} \quad (\mathit{valuation} \to \mathit{formula} \to \star),$$

where *valuation* and *formula* are the "types" for valuations and formulas, which shall be described below. The translation can be naturally extended to base types and function types by

$$b^\# := b \quad \text{and} \quad (\sigma \to \tau)^\# := \sigma^\# \to \tau^\#.$$

The "type" *formula* can be defined as an additional finite base type. An important observation is that only finitely various formulas are reachable by unfolding the definition of $\mathcal{T}[e, \emptyset \models \varphi_0]$. It is easy to see that the following set

$$\{\, \psi, \bigcirc\psi \mid \psi \text{ is a subformula of } \varphi_0 \,\}$$

is an overapproximation. So we define the values in $V_{formula}$ as this set. We shall write $\lfloor\psi\rfloor$ for the formula ψ seen as a value in $V_{formula}$. We assume an operation $=_{formula}$ to compare formulas. Since *formula* is now a finite base type, one can define a function by using pattern matching of formulas.

The "type" *valuation* can be implemented as a tuple. Note that valuations ρ reachable from $\mathcal{T}[e, \emptyset \models \varphi_0]$ have subsets of L_0 as their domains. So a reachable valuation ρ can be represented as a tuple of length $|L_0|$, where $|L_0|$ is the number of elements in L_0. If $\rho(\alpha)$ is undefined for some $\alpha \in L_0$, one can fill the corresponding place in a tuple by an arbitrary expression.

Summarising the above argument, the translation of the unit type is

$$\star \quad \overset{\#}{\longmapsto} \quad (\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n \to \mathit{formula} \to \star)$$

if the set of variables L_0 in φ_0 is $\{\, \alpha_1, \ldots, \alpha_n \,\}$ and σ_i is the type for α_i, $1 \le i \le n$. We shall fix the enumeration $\alpha_1, \ldots, \alpha_n$ of L_0 in the sequel.

We give the translation of expressions. The function definition $\mathcal{P}^\#$ after translation defines the following functions:

- $f^\# : \tau^\#$ for each function f defined in $\mathcal{P}$,
- $\alpha^\# : \tau^\# \to \tau^\#$ for each variable $\alpha \in L_0$ of type τ,
- $\mathbf{op}^\# : b_1 \to b_2 \to (b_3 \to \star^\#) \to \star^\#$ for each operation $op \in Op$,
- $\mathbf{if}^\# : \mathtt{Bool} \to \star^\# \to \star^\# \to \star^\#$, the translation of **if**, and
- $()^\# : \star^\#$, the translation of the unit value.

Note that $\alpha \in L_0$ does not have the translation if α has a base type; the label $(-)^\alpha$ is simply ignored by the translation (see Remark 1). Using these functions, the translation of expressions is given as follows:

$$c^\# := c \qquad x^\# := x \qquad (e_1\, e_2)^\# := e_1^\#\, e_2^\# \qquad (e^\alpha)^\# := \alpha^\#\, e^\# \quad \text{and} \quad (e^\beta)^\# := e^\#$$

where α (resp. β) is a variable in L_0 of a non-base type (resp. a base type). Other cases have already given by $\mathcal{P}^\#$: for example, $() \overset{\#}{\mapsto} ()^\#$ and $f \overset{\#}{\mapsto} f^\#$. A notable point is that the label annotation e^α is translated to application to $\alpha^\#$.

The translation of valuations should now be clear. A valuation is translated to a sequence of expressions, defined by

$$\rho \quad \overset{\#}{\longmapsto} \quad \rho(\alpha_1)^\# \ \ldots\ \rho(\alpha_n)^\#.$$

If $\rho(\alpha_i)$ is undefined, then $\rho(\alpha_i)^\#$ can be arbitrary (but fixed a priori) expression of the required type. We use ϱ for sequences of this kind. We write $\varrho[\alpha_i \mapsto e]$ for the sequence obtained by replacing the ith element in ϱ with e.

What remains is to give definitions of functions $\mathcal{P}^{\#}$ so that the value tree of $e^{\#}\,\rho^{\#}\,\lfloor\varphi\rfloor$ will coincide with the satisfaction tree $\mathcal{T}[e, \rho \models \varphi]$. Each function definition is of the form $h\,\widetilde{x}\,\varrho\,\lfloor\varphi\rfloor = e$, where $\widetilde{x}$ is a sequence of arguments in the original definition, ϱ is a sequence representation of a tuple of type *valuation*, and $\lfloor\varphi\rfloor \in V_{formula}$ is the value of type *formula*. All functions in $\mathcal{P}^{\#}$ are defined by pattern matching on the final argument $\lfloor\varphi\rfloor$. For example, consider the case of the final argument being $\lfloor\psi_1 \wedge \psi_2\rfloor$. Because

$$\mathcal{T}[h\,\widetilde{e}, \rho \models \psi_1 \wedge \psi_2] \quad = \quad \mathcal{T}[h\,\widetilde{e}, \rho \models \psi_1] \sqcap \mathcal{T}[h\,\widetilde{e}, \rho \models \psi_2],$$

the definition[6] of h for this case has to be

$$h\,\widetilde{x}\,\varrho\,\lfloor\psi_1 \wedge \psi_2\rfloor \quad = \quad (h\,\widetilde{x}\,\varrho\,\lfloor\psi_1\rfloor) \sqcap (h\,\widetilde{x}\,\varrho\,\lfloor\psi_2\rfloor).$$

As an example of more complicated case, let us consider the rule

$$\mathcal{T}[e^{\alpha}\,e_1\,\ldots\,e_n,\ \rho \models \mathit{call}\,\alpha(\beta_1, \ldots, \beta_n).\varphi] \quad = \quad \mathcal{T}[e\,e_1^{\beta_1}\,\ldots\,e_n^{\beta_n},\ \rho' \models \bigcirc\varphi]$$

where $\rho' = \rho \cup \{\beta_i \mapsto e_i\}_{1 \le i \le n}$. Because

$$(e^{\alpha}\,e_1\,\ldots\,e_n)^{\#} \quad = \quad \alpha^{\#}\,e^{\#}\,e_1^{\#}\,\ldots\,e_n^{\#},$$

the above rule can be seen as (a part of) the definition of $\alpha^{\#}$:

$$\alpha^{\#}\,g\,\widetilde{x}\,\varrho\,\lfloor\mathit{call}\,\alpha(\beta_1, \ldots, \beta_n).\varphi\rfloor \quad = \quad g\,(\beta_1^{\#}\,x_1)\,\ldots\,(\beta_n^{\#}\,x_n)\,\varrho'\,\lfloor\bigcirc\varphi\rfloor$$

where $\varrho' = \varrho[\beta_1 \mapsto x_1]\ldots[\beta_n \mapsto x_n]$. It is easy to check that

$$(e^{\alpha}\,e_1\,\ldots\,e_n)^{\#}\,\rho^{\#}\,\lfloor\mathit{call}\,\alpha(\beta_1, \ldots, \beta_n).\varphi\rfloor \quad \longrightarrow^{*} \quad (e\,e_1^{\beta_1}\,\ldots\,e_n^{\beta_n})^{\#}\,\rho'^{\#}\,\lfloor\bigcirc\varphi\rfloor$$

as expected. All other cases are given in the same way.

Now the definition of the translation has been given in sufficient detail, we believe. It is not difficult to establish the following lemma.

Lemma 1. *The value tree of $e^{\#}\,\rho^{\#}\,\lfloor\varphi\rfloor$ is equivalent to $\mathcal{T}[e, \rho \models \varphi]$, provided that ρ and φ are reachable from the definition of $\mathcal{T}[e_1, \emptyset \models \varphi_0]$ for some expression e_1 of type $\star$.*

Theorem 3 is a consequence of this lemma: e_0' can be defined as $e_0^{\#}\,\emptyset^{\#}\,\lfloor\varphi_0\rfloor$.

The decidablity result is a corollary of Theorems 2 and 3.

Theorem 4. *Let $(\mathcal{P}, e)$ is a program and φ is an LTLC formula. If $(\mathcal{P}, e)$ and φ contain only finite base types, then one can effectively decide whether $e, \emptyset \models \varphi$.*

[6] Strictly speaking, the "function definition" here does not precisely follow the syntax of function definition in our language, as we do not allow pattern matching on the left-hand-side of a definition, but we expect that the reader can fill the gap.

Remark 2. Let us briefly discuss the time complexity of the algorithm. The cost of the translation is negligible; we estimate the running time of the higher-order model checking. If we fix the property automaton to $\mathcal{A}$ in Theorem 2, the higher-order model checking be solved in time $O(P^2\,\mathbf{exp}_N(poly(A\,D)))$ for some polynomial *poly* ([13, Section 5] adopted to our setting), where P, N, A and D are parameters determined by the program after translation; P is the size, N is the order, A is the maximum arity of types and D is the maximum number of values in base types. Easy calculation shows that

$$P = O(|(\mathcal{P}, e)| \times |\varphi|) \qquad N \leq order(\mathcal{P}, e) + 2 \qquad N = O(|\varphi|) \qquad A = O(|\varphi|)$$

where $|(\mathcal{P}, e)|$ and $|\varphi|$ are the sizes of the program and of the formula.

6 Discussions

Compositional Reasoning. LTLC model checking is a kind of whole-program verification. Actually $C[f], \rho \models \Phi_T(f)$, where Φ_T is the LTLC formula corresponding to a dependent type T (see Sect. 4.1), only means that the behaviour of f *in the context* C does not violate the specification T; it does not ensure that f meets T in other contexts as well.

This is in contrast to a compositional approach such as a type-based one, in which $\vdash t : T$ means that t satisfies the specification T in whatever the context t is used. In this sense $\Phi_T(f)$ is not like a type judgement but like dynamic monitoring of a contract [9].

A way to fill the gap is to consider all possible contexts. That means, we define $\models f : T$ to mean that $C[f], \emptyset \models \Phi_T(f)$ for every context C. In a sufficiently expressive programming language, $\forall C.\big(\, C[f], \emptyset \models \Phi_T(f)\,\big)$ is equivalent to $C_0[f], \emptyset \models \Phi_T(f)$ for a certain "worst" context C_0; this observation gives us a way to reduce compositional reasoning to whole-program analysis. This strategy is actually used in [23], for example.

A typical way to construct the "worst" context C_0 is to use nondeterminism [23]; intuitively C_0 is a "maximally" nondeterministic context, which may do anything allowed. Unfortunately this construction is not directly applicable to our case, since our reducibility result (in particular, Theorem 2) essentially relies on the determinism of programs.

Non-deterministic Programs. Determinism of programs is essential to our reducibility result. In fact, even the definition of the satisfaction relation becomes "incorrect" in the presence of non-determinism.

To see the reason, consider an LTLC formula

$$\varphi \quad := \quad \mathit{ifcall}\ f \vee \neg \mathit{ifcall}\ f,$$

which is obviously true for every program. By definition, we have

$$e, \rho \models \mathit{ifcall}\ f \vee \neg \mathit{ifcall}\ f \qquad \text{iff} \qquad e, \rho \models \mathit{ifcall}\ f \quad \text{or} \quad e, \rho \models \neg \mathit{ifcall}\ f,$$

for every expression e. This rule is problematic in the presence of nondeterminism. For example, consider $e = (f\,()) \oplus ()$ where $\oplus$ is the nondeterministic branching. This expression decides nondeterministically whether it calls f or not. Then $e, \rho \models \mathit{ifcall}\ f \vee \neg \mathit{ifcall}\ f$ but neither $e, \rho \models \mathit{ifcall}\ f$ nor $e, \rho \models \neg \mathit{ifcall}\ f$.

This problem can be easily fixed by changing the definition of the satisfaction relation. It should be a relation $\pi, \rho \models \varphi$ on an infinite reduction sequence π (instead of an expression e), a valuation and a formula in the presence of nondeterminism.

However Theorem 2 cannot be modified accordingly to the new definition. The definition of $\mathcal{T}[e, \rho \models \varphi]$ is so deeply related to the current definition of the satisfaction that we cannot obtain a variant of Theorem 2 applicable to nondeterministic setting.

Actually we conjecture that LTLC model-checking for nondeterministic programs is undecidable even for programs with only finite data types. The proof of the conjecture is left for future work.

7 Related Work

LTLC model checking is a kind of temporal verification of higher-order programs, which has been extensively studied [11,12,14,15,22]. The temporal properties of higher-order programs have also been studied in the context of contracts, named *temporal higher-order contracts* [7].

Alur et al. proposed a linear temporal logic called CARET [2], which is designed for specifying properties for first-order programs modeled by Recursive State Machines [3] and Pushdown Sytems [16]. Neither CARET nor LTLC subsumes the other. On the one hand, CARET cannot describe properties of higher-order functions, such as "a function argument of some function is eventually called." On the other hand, CARET can describe *caller* properties such as "a caller function of the function currently invoded is never returned," which cannot be expressed in LTLC. An extension of LTLC for specifying caller properties is left for future work. Alur and Madhusudan proposed Visibly Pushdown Languages (VPL) [4], which can specify properties of function calls and returns, and subsumes CARET. Like CARET, VPL is for first-order programs, not for higher-order programs.

Recently Satake and Unno proposed a dynamic logic for higher-order programs, named HOT-PDL [22]. Their logic is not directly comparable to ours, as their logic is for call-by-value programs. The gap can be partially filled by applying the CPS translation, and the formulas in their logic can be translated to LTLC formulas in many cases, although we need to extend LTLC by *anonymous call operator* $\mathit{call}\ \_(\widetilde{\beta}).\varphi$ to fully capture their logic. Many LTLC formulas such as those in Example 2 and Sect. 4.3 cannot be expressed in HOT-PDL.

Applications of HORS model checking to program verification has been studied [11,12,14,15,17,18,22]. Decidability of resource usage verification has been proved in [12] by using a program translation tailor-made for the resource usage verification problem. The argument in Sect. 4.3 together with Theorem 4 gives

another, more principled proof of the decidability result, although the current argument proves only a partial result of [12].

8 Conclusion

We have proposed a temporal logic called LTLC, which is an extension of LTL and can specify properties for call-by-name higher-order programs. Thanks to the call operator, LTLC can describe properties of arguments of function currently called. For example, LTLC can specify the order of function calls such as "the first argument passed to the function f is called before the call of the second argument passed to f." We have shown that LTLC model checking is decidable for a finite-data deterministic programs via a reduction to HORS model checking.

The most important future work is to prove the undecidability (possibly, the decidability) of LTLC model checking for non-deterministic programs. To further widen the scope of our method, it is worth extending LTLC for specifying branching properties by embedding the call operator into CTL* or modal μ-calculus.

References

1. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A relational logic for higher-order programs. PACMPL **1**(ICFP), 21:1–21:29 (2017)
2. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_35
3. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 207–220. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_18
4. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Babai, L. (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13–16, 2004. pp. 202–211. ACM (2004)
5. Asada, K., Sato, R., Kobayashi, N.: Verifying relational properties of functional programs by first-order refinement. Sci. Comput. Program. **137**, 2–62 (2017)
6. Broadbent, C.H., Kobayashi, N.: Saturation-based model checking of higher-order recursion schemes. In: Rocca, S.R.D. (ed.) Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2–5, 2013, Torino, Italy. LIPIcs, vol. 23, pp. 129–148. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
7. Disney, T., Flanagan, C., McCarthy, J.: Temporal higher-order contracts. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011, pp. 176–188. ACM (2011)
8. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM (JACM) **33**(1), 151–178 (1986)
9. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Wand, M., Peyton Jones, S.L. (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2002), Pittsburgh, Pennsylvania, USA, October 4–6, 2002, pp. 48–59. ACM (2002)

10. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. **27**(2), 264–313 (2005)
11. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009. pp. 416–428. ACM (2009)
12. Kobayashi, N.: Model checking higher-order programs. J. ACM **60**(3), 20:1–20:62 (2013)
13. Kobayashi, N., Ong, C.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11–14 August 2009, Los Angeles, CA, USA. pp. 179–188. IEEE Computer Society (2009)
14. Lester, M.M., Neatherway, R.P., Ong, C.L., Ramsay, S.: Model checking liveness properties of higher-order functional programs. In: Proceedings of ML Workshop, vol. 2011 (2011)
15. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 57–68. ACM (2016)
16. Nguyen, H., Touili, T.: CARET model checking for pushdown systems. In: Seffah, A., Penzenstadler, B., Alves, C., Peng, X. (eds.) Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3–7, 2017, pp. 1393–1400. ACM (2017)
17. Ong, C.L.: On model-checking trees generated by higher-order recursion schemes. In: 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12–15 August 2006, Seattle, WA, USA, Proceedings, pp. 81–90. IEEE Computer Society (2006)
18. Ong, C.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011, pp. 587–598. ACM (2011)
19. Pnueli, A.: The temporal logic of programs. In: In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
20. Ramsay, S.J., Neatherway, R.P., Ong, C.L.: A type-directed abstraction refinement approach to higher-order model checking. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20–21, 2014, pp. 61–72. ACM (2014)
21. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008, pp. 159–169. ACM (2008)
22. Satake, Y., Unno, H.: Propositional dynamic logic for higher-order functional programs. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 105–123. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_6
23. Sato, R., Asada, K., Kobayashi, N.: Refinement type checking via assertion checking. JIP **23**(6), 827–834 (2015)