

A Higher Order Modal Fixed Point Logic

Mahesh Viswanathan¹ and Ramesh Viswanathan²

¹ University of Illinois at Urbana-Champaign
vmahesh@uiuc.edu

² Bell Laboratories
rv@research.bell-labs.com

Abstract. We present a higher order modal fixed point logic (HFL) that extends the modal μ -calculus to allow predicates on states (sets of states) to be specified using recursively defined higher order functions on predicates. The logic HFL includes negation as a first-class construct and uses a simple type system to identify the monotonic functions on which the application of fixed point operators is semantically meaningful. The model checking problem for HFL over finite transition systems remains decidable, but its expressiveness is rich. We construct a property of finite transition systems that is not expressible in the Fixed Point Logic with Chop [1] but which can be expressed in HFL. Over infinite transition systems, HFL can express bisimulation and simulation of push down automata, and any recursively enumerable property of a class of transition systems representing the natural numbers.

1 Introduction

An attractive methodology for compositional or hierarchical verification is the assumption-guarantee paradigm [2], in which a component of a system is specified in terms of assumptions it makes about its environment (other components), and properties it guarantees about its behavior, provided the assumptions hold. Using $\varphi \triangleright \psi$ to syntactically denote the property that under the assumptions φ , the property ψ is guaranteed, the semantics of the assume guarantee property needs to accommodate the following circular compositional rule: for a system $P = P_1 || P_2$, if P_1 satisfies the property $\varphi_2 \triangleright \varphi_1$ and P_2 satisfies the property $\varphi_1 \triangleright \varphi_2$ then P satisfies $\varphi_1 \wedge \varphi_2$. That such a semantics can be defined, for certain properties, was first observed by Misra and Chandy [3], and later formalized by Abadi and Lamport [4, 5]. Subsequently, it has been extended to other concurrency models and richer classes of properties [6, 7]. A unifying framework was provided in [8], in which the assume guarantee semantics was defined for properties expressible as fixed points; previously proposed rules then arise as instances of this framework.

To utilize the assume-guarantee paradigm in developing a formal system for compositional reasoning of concurrent programs, an obvious necessity is a logic or language in which the assume-guarantee semantics can be expressed. A natural candidate logic is the modal μ -calculus [9] which contains almost all other

temporal specification logics [10], and for which model-checking is decidable [11]. The distinguishing feature of the μ -calculus is the presence of fixed points that allow recursive definitions of predicates on states (subsets of states), thus corresponding exactly to the class of properties considered in [8]. But, as we show in Section 2, there are assume-guarantee properties that cannot be expressed in the μ -calculus, *i.e.*, assume-guarantee properties between two recursively defined subsets of states are not necessarily, in turn, expressible recursively. However, as also detailed in Section 2, there is a simple way to express all assume-guarantee properties by using a recursively defined function that takes as arguments subsets of states and returns a subset of states. Consequently, we were interested in a logic in which we can express recursively defined higher order functions.

A natural way to extend the modal μ -calculus to include higher order functions is to add the operations of λ -calculus, together with higher-order fixed points. The main technical difficulty with such an extension is a suitable accounting of nonmonotonic operators such as negation. In the modal μ -calculus, it can be assumed (without any loss of expressiveness) that negation is applied only to propositional constants and not to arbitrary formulas — in particular, negation cannot be applied to variables. This automatically ensures that formulas are monotonic in all their free variables thus assuring the semantic well-definedness of recursively binding them. Making such an assumption in the higher-order case would however be overly restrictive. The reason is that while the only intended use of variables in the modal μ -calculus is for recursive definitions, the extension with λ -abstraction also includes variables that are λ -bound, *i.e.*, used as formal parameters of functions which are semantically well-defined independent of being monotonic. Thus, restricting the use of negation, would force all definable functions to be monotonic, and it may lead to a loss in expressivity. In particular, the function that we use in expressing assume-guarantee properties is antimonotonic in one of its arguments, and furthermore requires the application of negation to a variable that stands for a formal function parameter. However, allowing the free use of negation means that formulas expressible in the logic can be of arbitrary monotonicity in their free variables and the logic needs to incorporate a systematic means of distinguishing semantically meaningful recursive definitions from invalid ones.

We present a logic that we call higher order fixed point logic (HFL). At the level of terms, HFL is a simple full-fledged union of propositional logic, modality operators, and λ -calculus with fixed point operators. It thus allows arbitrary use of negation and can accommodate recursive definitions of functions of arbitrary monotonicity. However, we formulate a type system that is an enrichment of the simply-typed λ -calculus which identifies the monotonicity of terms in their free variables and assures all well-typed terms to be semantically well-defined.

While the ability to define higher-order functions may be interesting, their impact on the class of definable predicates on states is not obvious *a priori*. The formulation of the logic HFL allows us to precisely explore this question. We consider the Fixed Point Logic with Chop (FLC), first proposed in [1], and further studied in [12, 13]. Formulas in FLC denote unary functions from sets of

states to sets of states and the logic includes fixed points allowing such predicate transformers to be expressed recursively. FLC is strictly more expressive than μ -calculus and can express, for example, context-sensitive languages of finite linear processes. We exhibit a translation of FLC into HFL that preserves the semantics of formulas, thus showing that HFL is as expressive as FLC. We then construct a property of finite transition systems that is not expressible in FLC but which is shown to be definable in HFL. This shows that definable functions of more than one argument and higher-order contribute to increased expressivity. Similarly, it can be proved that HFL is strictly more expressive than every sublogic of HFL in which the number of arguments or order of functional arguments is restricted to any finite level, *i.e.*, the increased expressivity resulting from the order of the functions used continues through at all levels. In spite of this richer expressiveness even over finite transition systems, model checking for the full logic HFL is decidable. Furthermore, properties of transition systems expressible in HFL are shown to be closed upto bisimulation. However, the satisfiability and validity problems are undecidable for FLC [1], and are therefore *a fortiori* undecidable for HFL as well.

The property shown to be provably not expressible in FLC is constructed by encoding FLC formulas as finite transition systems and diagonalizing over them. To our knowledge, the encoding is novel and provides the first inexpressibility result for FLC. On the other hand, this construction does not yield a particularly natural property although it is inherently dictated by the fact that FLC can express context sensitive languages and the only known proofs that certain languages are not context sensitive are ultimately based on diagonalization (*c.f.* [14]). Examples of more directly presentable properties expressible in HFL are: (a) Simulation and bisimulation of Push Down Automata (PDA) processes [15–17], and (b) Partial recursive functions and recursively enumerable properties over a class of infinite transition systems representing the natural number; due to space limitations, these constructions are not detailed in this paper. The expressibility of both these properties (as well as assume-guarantee properties) rely on recursively defined functions that take multiple arguments, and would therefore not be (directly) describable in FLC. Besides these examples, HFL may be well-suited for reasoning about higher-order concurrent programs that include procedural or object-oriented abstractions.

The rest of the paper is organized as follows. Section 2 details the motivating context of assume-guarantee properties. The syntax and semantics of HFL are defined in Section 3, and expressivity results for HFL are detailed in Section 4.

2 Motivation: Assume Guarantee Properties

Let S be the set of states of a transition system (formally defined in Section 3). In μ -calculus, the semantics of formulas are subsets of states, *i.e.*, for a μ -calculus formula φ , its semantics $\llbracket \varphi \rrbracket \in 2^S$ where 2^S denotes the powerset of S ; a transition system satisfies a formula φ if its initial state belongs to $\llbracket \varphi \rrbracket$. Such subsets of states can be defined recursively as least or greatest fixed points of mono-

tonic functions $F: 2^S \rightarrow 2^S$; we use $\mu X.F(X)$ to denote the least fixed point and $\nu X.F(X)$ for the greatest fixed point. The Tarski-Knaster [18] construction approximates the fixed points through repeated iterations of F whose limit yields the desired fixed point. In the case of greatest fixed points, the k 'th approximation, denoted by $[\nu X.F(X)]^k$, is defined inductively as $[\nu X.F(X)]^0 = S$, and $[\nu X.F(X)]^{k+1} = F([\nu X.F(X)]^k)$. Assume-guarantee specifications will be syntactically denoted by $\nu X.A(X) \triangleright \nu X.G(X)$, and their informal reading is that the guarantee specification $\nu X.G(X)$ is satisfied whenever the assumption specification $\nu X.A(X)$ is satisfied. The semantics of an assume-guarantee property $\nu X.A(X) \triangleright \nu X.G(X)$, given in Definition 1³ below (from [8]), requires that if the environment ensures that the k 'th approximation of A is satisfied, then the $k + 1$ 'st approximation of G must be satisfied.

Definition 1. For monotonic functions $A, G: 2^S \rightarrow 2^S$, $\nu X.A(X) \triangleright \nu X.G(X) \in 2^S$ is defined as $s \in \nu X.A(X) \triangleright \nu X.G(X)$ iff $\forall k \geq 0. s \in [\nu X.A(X)]^k \Rightarrow s \in [\nu X.G(X)]^{k+1}$.

The salient property of the semantics given by Definition 1 is that for any state s such that $s \in \nu(X).A(X) \triangleright \nu X.G(X)$ and $s \in \nu X.G(X) \triangleright \nu X.A(X)$ we have that $s \in \nu(X).A(X) \cap \nu(X).G(X)$ — this can be shown using induction [8].

We now show that the μ -calculus is not closed under assume-guarantee specifications. Consider the μ -calculus formulas $\varphi = \nu X.X \wedge \langle a \rangle X$ and $\psi = \nu X.X \wedge \langle b \rangle X$ which assert that there is a path where an a and b transition respectively is always enabled. A transition system then satisfies $\varphi \triangleright \psi$ iff it has the property that for every n , if there is a path of length n of a transitions from the initial state then there is a path of length $n + 1$ of b transitions from the initial state. Viewing this as a property of computation trees (the unrolling of a transition system to yield a possibly infinite tree) we show that this is not a regular tree language. Since the set of computation trees associated with models satisfying a μ -calculus formula define a regular language, it follows that the property $\varphi \triangleright \psi$ cannot be expressed in μ -calculus. The intuition behind why a tree automaton cannot recognize this property is because the tree automaton will need to “remember” the length of the longest a -sequence and use this to check against the length of a b sequence in another part of the transition system. Since the a sequence can be of arbitrary length, the automaton does not have enough “memory” to do the necessary checks. Formalizing this argument yields the following proposition.

Proposition 1. There exist properties φ and ψ expressible in μ -calculus, such that $\varphi \triangleright \psi$ is not expressible in μ -calculus.

The properties φ, ψ that we have exhibited are in fact expressible in CTL as well. Hence Proposition 1 actually demonstrates that all the classical branching-time logics, *i.e.*, CTL, CTL*, and μ -calculus, cannot express all assume-guarantee specifications built from any formulas in the respective logic.

³ This special instance of the more general definition is applicable to fixed points whose approximations converge within ordinal ω

However, assume-guarantee properties can be expressed naturally using a recursively defined function on predicates of states. Writing $\neg X$ to denote the complement $S - X$ for a set $X \in 2^S$, and $X \wedge Y$ to denote the intersection of sets X, Y , consider the function $AssGuar^{A,G}$ (where $A, G: 2^S \rightarrow 2^S$ monotonic) which takes two arguments $x, y \in 2^S$ and returns an element of 2^S , which is the greatest solution satisfying the following recursive definition:

$$AssGuar^{A,G}(x, y) = (\neg x \vee y) \wedge AssGuar^{A,G}(A(x), G(y))$$

The function $AssGuar^{A,G}(S_1, S_2)$ returns a set of states S_3 such that $s \in S_3$ iff $\forall k \geq 0. s \in A^k(S_1) \Rightarrow s \in G^k(S_2)$. By using S for S_1 and $G(S)$ for S_2 , we can get the assume-guarantee property as $\nu X. A(X) \triangleright \nu X. G(X) = AssGuar^{A,G}(\mathbf{tt}, G(\mathbf{tt}))$, where we write \mathbf{tt} to denote the set S .

The property $\nu X. A(X) \triangleright \nu X. G(X)$ can thus be written using a function on subsets of sets ($AssGuar^{A,G}$) that is not monotonic in one of its arguments (x) and which is recursively defined using a body that applies negation to one of its parameters (x). Because of the use of negation, it is not even clear that the recursive solution that we require for $AssGuar^{A,G}$ is semantically well-defined. This motivates the formulation of a logic in which such functions can be defined and their semantic validity can be established.

3 The Logic HFL

Similar to the μ -calculus, our logic will be interpreted over labelled transition systems. Let $\mathcal{P} = \{p, q, \dots\}$ be a set of propositional constants, and $Act = \{a, b, \dots\}$ be a set of action names. A labelled transition system is a structure $\mathcal{T} = (S, \{\overset{a}{\rightarrow} \mid a \in Act\}, L, s_0)$, where S is a set of states, $\overset{a}{\rightarrow}$ for each $a \in Act$ is a binary relation on states, $L: S \rightarrow 2^{\mathcal{P}}$ with $L(s)$ for any $s \in S$ being the set of propositional constants that are true in state s , and $s_0 \in S$ is the state designated as the initial state. We use the infix notation $s \overset{a}{\rightarrow} t$ to denote that $(s, t) \in \overset{a}{\rightarrow}$. The transition system is finite if the set of states S is finite.

The types of the logic are given by the following grammar:

$$\begin{array}{ll} \text{(Variances)} & v ::= + \mid - \mid 0 \\ \text{(Types)} & A ::= \mathbf{Prop} \mid A^v \rightarrow A \end{array}$$

We use letters A, B, \dots to range over types. Each type will be interpreted as a partially ordered set. The base type \mathbf{Prop} intuitively represents the type of “properties” — its elements are subsets of states ordered by set inclusion. The elements of the type $A^v \rightarrow B$ are functions from A to B that respect the ordering on the type A in a manner given by the variance v — the type $A^+ \rightarrow B$ consists of functions that are monotonic with respect to the ordering on A , the type $A^- \rightarrow B$ consists of functions that are antimonotonic with respect to the ordering on A , and the type $A^0 \rightarrow B$ consists of arbitrary functions that are not required to be monotonic or antimonotonic. These intuitions are formalized in Definition 2 below, where for partial orders $\mathcal{A} = (A, \leq_A)$, $\mathcal{B} = (B, \leq_B)$, we use

$\mathcal{A} \rightarrow \mathcal{B}$ to denote the partial order of monotone functions ordered pointwise, *i.e.*, the underlying set of $\mathcal{A} \rightarrow \mathcal{B}$ is $\{f: \mathcal{A} \rightarrow \mathcal{B} \mid \forall x, y \in \mathcal{A}. x \leq_{\mathcal{A}} y \Rightarrow f(x) \leq_{\mathcal{B}} f(y)\}$ and the ordering relation given by $f \leq_{\mathcal{A} \rightarrow \mathcal{B}} g$ iff $\forall x \in \mathcal{A}. f(x) \leq_{\mathcal{B}} g(x)$.

Definition 2 (Semantics of Types).

1. For any binary relation $R \subseteq A \times A$ on a set A , define the binary relations $R^+, R^-, R^0 \subseteq A \times A$ as follows: $R^+ = R$, $R^- = \{(a, b) \mid (b, a) \in R\}$, and $R^0 = R^+ \cap R^-$. For any partial order $\mathcal{A} = (A, \leq_{\mathcal{A}})$, define the partial order \mathcal{A}^v as $(A, \leq_{\mathcal{A}}^v)$, where $v \in \{+, -, 0\}$.
2. Let $\mathcal{T} = (S, \{\overset{a}{\rightarrow} \mid a \in \text{Act}\}, L, s_0)$ be a labelled transition system. The semantics $\mathcal{T}[[A]]$ of any type A is a partial order defined by induction on the type A as:

$$\begin{aligned} \mathcal{T}[[\text{Prop}]] &= (2^S, \subseteq) \\ \mathcal{T}[[A^v \rightarrow B]] &= (\mathcal{T}[[A]])^v \rightarrow \mathcal{T}[[B]] \end{aligned}$$

It is easily verified that for any partial order $\mathcal{A} = (A, \leq_{\mathcal{A}})$, the structure \mathcal{A}^v is a partial order (*i.e.*, the relation $\leq_{\mathcal{A}}^v$ is also reflexive, transitive, and antisymmetric) and since $\mathcal{A} \rightarrow \mathcal{B}$ is a partial order for any partial orders \mathcal{A}, \mathcal{B} , it follows that $\mathcal{T}[[A]]$ is a well-defined partial order for any type A . Furthermore, the partial order $(2^S, \subseteq)$ is a complete lattice (with set unions and intersections giving joins and meets respectively) and for any partial order \mathcal{A} and complete lattice \mathcal{B} , the partial order $\mathcal{A} \rightarrow \mathcal{B}$ is a complete lattice (with joins and meets computed pointwise); it therefore follows that $\mathcal{T}[[A]]$, for any type A , is a complete lattice. For a partial order \mathcal{A} that is a complete lattice, we use $\sqcup_{\mathcal{A}}$ and $\sqcap_{\mathcal{A}}$ to denote its join and meet operations, and $\perp_{\mathcal{A}}$ and $\top_{\mathcal{A}}$ to denote its least and greatest elements.

Let $\mathcal{V} = \{x, y, x_1, \dots\}$ be a set of variable names. The terms of the logic are generated by the following grammar, where p ranges over the set of propositional constants \mathcal{P} , x ranges over the set of variable names \mathcal{V} , and a ranges over the set of action names Act .

$$\varphi ::= \text{ff} \mid p \mid x \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid \lambda(x^v: A)\varphi \mid (\varphi \varphi) \mid \mu(x: A)\varphi$$

We use $\varphi, \psi, \varphi_1, \dots$ to range over terms. In comparing with the propositional μ -calculus, the new term form $\lambda(x^v: A)\varphi$ corresponds to function definitions and $(\varphi \psi)$ denotes the value of the function φ on the argument ψ . Additionally, least fixed points $\mu(x: A)\varphi$ are now available at all types, with the type annotation A on the μ -bound variable indicating the type at which the fixed point is being taken. In the term $\lambda(x^v: A)\varphi$, the type annotation A corresponds to the expected type of its argument x and the variance annotation v corresponds to the expected variance of the function in its argument x . We identify terms upto renaming of bound variables (α -equivalence) with λ and μ being the variable-binding constructs, and use $\varphi[x \mapsto \psi]$ for the substitution of term ψ for variable x in the term φ (with suitable renaming of bound variables to avoid capturing free variables). Terms for propositional conjunction, $[a]$ modalities, and greatest fixed points can be derived and therefore not included in the syntax of primitive terms. Following

Table 1. Syntax of Higher Order Fixed Point Logic

<i>(false)</i>	$\Gamma \vdash \text{ff}: \text{Prop}$	<i>(prop)</i>	$\Gamma \vdash p: \text{Prop}$
<i>(var)</i>	$\Gamma', x^v: A, \Gamma'' \vdash x: A \quad \text{if } v \in \{0, +\}$	<i>(not)</i>	$\frac{\Gamma^- \vdash \varphi: \text{Prop}}{\Gamma \vdash \neg \varphi: \text{Prop}}$
<i>(or)</i>	$\frac{\Gamma \vdash \varphi_1: \text{Prop} \quad \Gamma \vdash \varphi_2: \text{Prop}}{\Gamma \vdash \varphi_1 \vee \varphi_2: \text{Prop}}$	<i>(\langle \rangle)</i>	$\frac{\Gamma \vdash \varphi: \text{Prop}}{\Gamma \vdash \langle a \rangle \varphi: \text{Prop}}$
<i>(abs)</i>	$\frac{\Gamma, x^v: A \vdash \varphi: B}{\Gamma \vdash \lambda(x^v: A)\varphi: A^v \rightarrow B}$	<i>(\mu)</i>	$\frac{\Gamma, x^{\dagger}: A \vdash \varphi: A}{\Gamma \vdash \mu(x: A)\varphi: A}$
<i>(app⁺)</i>	$\frac{\Gamma \vdash \varphi: A^+ \rightarrow B \quad \Gamma \vdash \psi: A}{\Gamma \vdash (\varphi \psi): B}$	<i>(app⁻)</i>	$\frac{\Gamma \vdash \varphi: A^- \rightarrow B \quad \Gamma^- \vdash \psi: A}{\Gamma \vdash (\varphi \psi): B}$
<i>(app⁰)</i>	$\frac{\Gamma \vdash \varphi: A^0 \rightarrow B \quad \Gamma \vdash \psi: A \quad \Gamma^- \vdash \psi: A}{\Gamma \vdash (\varphi \psi): B}$		

standard λ -calculus conventions, we write $A_1^{v_1} \rightarrow \dots \rightarrow A_n^{v_n} \rightarrow B$ to mean the type $A_1^{v_1} \rightarrow (\dots \rightarrow (A_n^{v_n} \rightarrow B))$ and $\varphi \psi_1 \dots \psi_n$ as shorthand for $(\dots ((\varphi \psi_1) \psi_2) \dots \psi_n)$.

We use a type system to identify a subset of the terms generated by the above grammar as being well-formed. Besides restricting the application of functional terms to arguments of the right type, the main purpose of the typing rules is to ensure that in a term $\mu(x: A)\varphi$, the term φ is monotonic in its μ -bound variable x to assure the existence of the least fixed point. The type system consists of proof rules for deriving judgements of the form $\Gamma \vdash \varphi: A$, where the context Γ is a sequence of the form $x_1^{v_1}: A_1, \dots, x_n^{v_n}: A_n$ with variables x_1, \dots, x_n all distinct and each variance annotation $v_i \in \{+, -, 0\}$. A derivable judgement $x_1^{v_1}: A_1, \dots, x_n^{v_n}: A_n \vdash \varphi: A$ is read as consisting of two assertions: (1) if variables x_1, \dots, x_n have types A_1, \dots, A_n respectively then φ is a well-formed term of type A , and (2) the variance of the term φ in the variable x_i is given by the annotation v_i : if $v_i = +$ then φ is monotonic in x_i , if $v_i = -$ then φ is antimonotonic in x_i , and if $v_i = 0$ then nothing about the variance in x_i is asserted.

In defining the typing rules, we use the following notation. For a variance v , we define its negation v^- as: $+^- = -$, $-^- = +$, and $0^- = 0$. This definition is extended pointwise to contexts, so that for a context $\Gamma = x_1^{v_1}: A_1, \dots, x_n^{v_n}: A_n$, the context Γ^- is defined to be $x_1^{v_1^-}: A_1, \dots, x_n^{v_n^-}: A_n$. The type system is given in Table 1 and consists of axioms for ff , propositional constants, and variables, and inference rules for the remaining term constructs. As is to be expected, the proof rule (μ) requires the μ -bound variable to appear monotonically in the body. The most interesting typing rule is that for application which splits into three cases depending on the variance of the function being applied in its argument. It is most easily understood on the basis of the semantic requirement of derivable typing judgements given by the second part of Lemma 1 below. The typing rules are simple but account faithfully for some of the subtleties in the interaction of negation with variables of higher-order type. As a simple example, consider the term $\varphi \equiv (f(\neg x)) \vee (\neg z)$ which at first glance seems to have x appearing negatively. Following the typing rules (var) and (not) , we can

Table 2. Semantics of Higher Order Fixed Point Logic

For $\mathcal{T} = (S, \{\overset{a}{\rightarrow} \mid a \in \text{Act}\}, L, s_0)$

$$\begin{aligned}
\mathcal{T}[\Gamma \vdash \text{ff}: \text{Prop}] \eta &= \emptyset \\
\mathcal{T}[\Gamma \vdash p: \text{Prop}] \eta &= \{s \in S \mid p \in L(s)\} \\
\mathcal{T}[\Gamma \vdash x: A] \eta &= \eta(x) \\
\mathcal{T}[\Gamma \vdash \neg \varphi: \text{Prop}] \eta &= S - \mathcal{T}[\Gamma \vdash \varphi: \text{Prop}] \eta \\
\mathcal{T}[\Gamma \vdash \varphi \vee \psi: \text{Prop}] \eta &= \mathcal{T}[\Gamma \vdash \varphi: \text{Prop}] \eta \cup \mathcal{T}[\Gamma \vdash \psi: \text{Prop}] \eta \\
\mathcal{T}[\Gamma \vdash \langle a \rangle \varphi: \text{Prop}] \eta &= \{s \in S \mid s \overset{a}{\rightarrow} t \text{ for some } t \in \mathcal{T}[\Gamma \vdash \varphi: \text{Prop}] \eta\} \\
\mathcal{T}[\Gamma \vdash \lambda(x^v: A) \varphi: A^v \rightarrow B] \eta &= F \in \mathcal{T}[A^v \rightarrow B] \text{ s.t.} \\
&\quad \forall d \in \mathcal{T}[A]. F(d) = \mathcal{T}[\Gamma, x^v: A \vdash \varphi: B] \eta[x \mapsto d] \\
\mathcal{T}[\Gamma \vdash \varphi \psi: B] \eta &= \mathcal{T}[\Gamma \vdash \varphi: A^v \rightarrow B] \eta(\mathcal{T}[\Gamma' \vdash \psi: A] \eta) \\
\mathcal{T}[\Gamma \vdash \mu(x: A) \varphi: A] \eta &= \sqcap_{\mathcal{T}[A]} \{d \in A \mid \mathcal{T}[\Gamma, x^+: A \vdash \varphi: A] \eta[x \mapsto d] \leq_{\mathcal{T}[A]} d\}
\end{aligned}$$

see that f appears positively and z appears negatively, but the variance in x depends on the variance of the variable f in its argument type; it would in fact be a positive occurrence if f is antimonotonic. Indeed, using the typing rules, we can derive $f^+: \text{Prop}^- \rightarrow \text{Prop}, z^-: \text{Prop}, x^+: \text{Prop} \vdash \varphi: \text{Prop}$, from which it follows that $\mu(f: \text{Prop}^- \rightarrow \text{Prop}) \lambda(z^-: \text{Prop}) \mu(x: \text{Prop}) \varphi$ is a well-typed term of type $\text{Prop}^- \rightarrow \text{Prop}$ — a recursive definition of an antimonotonic function.

The following proposition states a technically useful property of the type system, where we use \equiv to denote syntactic identity.

Proposition 2 (Unique Types). *If $\Gamma \vdash \varphi: A$ and $\Gamma \vdash \varphi: A'$ are derivable, then $A \equiv A'$.*

In particular, every closed term has a unique type. From Proposition 2 and the form of the typing rules (every construct has a unique rule for its introduction except for application but whose proof rule is uniquely determined by the type of the subterm), it follows that the derivation tree is unique as well (upto renaming of any bound variables). The proof of Proposition 2 is a straightforward induction on the length of the derivation, but it holds only because of the inclusion of the type and variance annotations for bound variables in the syntax of terms. For example, without type annotations, the term $\mu(x)x$ can be given any type A , and without variance annotations, the term $\lambda(x: \text{Prop})x$ would have both the types $\text{Prop}^+ \rightarrow \text{Prop}$ and $\text{Prop}^0 \rightarrow \text{Prop}$.

We are now ready to define the semantics of terms. Let \mathcal{T} be a transition system. An environment η is a possibly partial map on the variable set \mathcal{V} . For a context $\Gamma = x_1^{v_1}: A_1, \dots, x_n^{v_n}: A_n$, we say that η is Γ -respecting, written $\eta \models \Gamma$ if $\eta(x_i) \in \mathcal{T}[A_i]$ for $i = 1, \dots, n$. We write $\eta[x \mapsto a]$ for the environment that maps x to a and is the same as η on all other variables: if $\eta \models \Gamma$ and $a \in \mathcal{T}[A]$ for some type A then $\eta[x \mapsto a] \models \Gamma, x: A$, where x is a variable that does not appear in Γ . For any well-typed term $\Gamma \vdash \varphi: A$ and environment $\eta \models \Gamma$, Table 2 defines its semantics $\mathcal{T}[\Gamma \vdash \varphi: A] \eta$ to be an element of $\mathcal{T}[A]$. Referring to Table 2, in the case of the application term $(\varphi \psi)$, the type $A^v \rightarrow B$ is the unique type (as

given by Proposition 2) of the term φ in the context Γ , the context Γ' is Γ if $v \in +, 0$, and is Γ^- if $v = -$.

For a context $\Gamma = x_1^{v_1}:A_1, \dots, x_n^{v_n}:A_n$, we define the preorder relation \preceq_Γ on Γ -respecting environments as $\eta \preceq_\Gamma \chi$ iff $\eta(x_i) \leq_{\mathcal{T}[[A_i]]}^{v_i} \chi(x_i)$ for $i = 1, \dots, n$ (the relation \leq^v as given by Definition 2). We then show by induction on the typing derivation that the semantics of terms given in Table 2 is well-defined as an element of the appropriate type and is monotonic with respect to the preordering on the context.

Lemma 1 (Semantics of Terms). *Let \mathcal{T} be a transition system. For any derivable $\Gamma \vdash \varphi: A$ and environments $\eta, \chi \models \Gamma$, we have the following:*

1. (Well-Definedness): $\mathcal{T}[[\Gamma \vdash \varphi: A]]\eta \in \mathcal{T}[[A]]$ and is uniquely defined.
2. (Variance): If $\eta \preceq_\Gamma \chi$ then $\mathcal{T}[[\Gamma \vdash \varphi: A]]\eta \leq_{\mathcal{T}[[A]]} \mathcal{T}[[\Gamma \vdash \varphi: A]]\chi$.

Since for any closed term φ , there is a unique type A_φ such that $\emptyset \vdash \varphi: A_\varphi$ is derivable, we use $\mathcal{T}[[\varphi]]$ to denote $\mathcal{T}[[\emptyset \vdash \varphi: A_\varphi]]\emptyset$ where the environment \emptyset is undefined on all variables. Formulas are closed terms of type **Prop**, i.e., terms φ such that $\emptyset \vdash \varphi: \mathbf{Prop}$ is derivable. As is standard, a transition system satisfies a formula, $\mathcal{T} \models \varphi$, iff the initial state $s_0 \in \mathcal{T}[[\varphi]]$. A property of a class \mathcal{C} of transition systems is simply a subset $\mathcal{P} \subseteq \mathcal{C}$. A property \mathcal{P} of a class of transition systems \mathcal{C} is *expressible* if there is a characteristic formula $\varphi_{\mathcal{P}}$ such that for any $\mathcal{T} \in \mathcal{C}$, we have that $\mathcal{T} \models \varphi_{\mathcal{P}}$ iff $\mathcal{T} \in \mathcal{P}$. For a closed term φ , we write $\varphi: A$ for the derivability of $\emptyset \vdash \varphi: A$.

3.1 Invariance Under Bisimilarity

Satisfaction of any HFL formula by a transition system is invariant under bisimilarity of transition systems. This property cannot be established directly by induction because HFL formulas (closed terms of type **Prop**) can have subterms of higher-order type — we therefore need to suitably relate the semantics of higher-order terms in different transition systems. For transition systems $\mathcal{T} = (S, \{\overset{a}{\rightarrow} \mid a \in \mathit{Act}\}, L, s_0)$ and $\mathcal{T}' = (S', \{\overset{a}{\rightarrow}' \mid a \in \mathit{Act}\}, L', s'_0)$ and states $s \in S, s' \in S'$, we write $s \sim_{\mathcal{T}, \mathcal{T}'} s'$ to denote that s (with respect to \mathcal{T}) is (label-respecting) bisimilar to s' (with respect to \mathcal{T}'), and $\mathcal{T} \sim \mathcal{T}'$ iff $s_0 \sim_{\mathcal{T}, \mathcal{T}'} s'_0$. For any type A , we define a binary relation $\mathcal{R}_{\mathcal{T}, \mathcal{T}'}^A \subseteq \mathcal{T}[[A]] \times \mathcal{T}'[[A]]$ by induction on the type A as follows (where we use the infix notation $a \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^A a'$ to denote that $(a, a') \in \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^A$):

$$\begin{aligned} P \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^{\mathbf{Prop}} P' &\text{ iff } \forall s \in S, s' \in S'. s \sim_{\mathcal{T}, \mathcal{T}'} s' \Rightarrow (s \in P \text{ iff } s' \in P') \\ F \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^{A \overset{v}{\rightarrow} B} F' &\text{ iff } \forall a \in \mathcal{T}[[A]], a' \in \mathcal{T}'[[A]]. a \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^A a' \Rightarrow F(a) \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^B F'(a') \end{aligned}$$

For a context $\Gamma = x_1^{v_1}:A_1, \dots, x_n^{v_n}:A_n$, define the relation $\mathcal{R}_{\mathcal{T}, \mathcal{T}'}^\Gamma$ between \mathcal{T} - and \mathcal{T}' -environments as $\eta \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^\Gamma \eta'$ iff $\eta(x_i) \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^{A_i} \eta'(x_i)$ for $i = 1, \dots, n$. The following lemma establishes the connection between the semantics of higher-order terms in different models and is proved by induction on the structure of terms. As an immediate corollary, bisimilar transition systems satisfy the same set of HFL formulas.

Lemma 2. *Let $\Gamma \vdash \varphi: A$ be any derivable term. For any transition systems \mathcal{T} , \mathcal{T}' and respective environments η, η' with $\eta \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^{\Gamma}$, η' , we have that*

$$\mathcal{T}[\Gamma \vdash \varphi: A]\eta \mathcal{R}_{\mathcal{T}, \mathcal{T}'}^A \mathcal{T}'[\Gamma \vdash \varphi: A]\eta'$$

Corollary 1 (Bisimilarity Invariance). *If $\mathcal{T} \sim \mathcal{T}'$ then for any formula φ , $\mathcal{T} \models \varphi$ iff $\mathcal{T}' \models \varphi$.*

3.2 Model Checking

The model checking problem for HFL is decidable over finite state transition systems. This is an immediate consequence of the fact that for any finite transition system \mathcal{T} , the underlying set of $\mathcal{T}[[B]]$ is finite for every type B . It therefore follows that $\mathcal{T}[\Gamma \vdash \varphi: A]\eta$ can be computed inductively on the term φ (following the definition in Table 2) and using the standard iterative approximations to compute the semantics of fixed point terms. These iterative approximations for a fixed point term of type B converge after at most h_B iterations where h_B is the length of the longest strictly increasing chain in the partial order $\mathcal{T}[[B]]$ (which is a finite number for any type B). This model-checking procedure is effective but not the most efficient; we leave exploration of other model-checking methods such as those based on tableaux to future work.

4 Expressiveness of HFL

Section 4.1 describes some basic definable operations in HFL that are used in developing the expressivity results, and Section 4.2 shows that HFL can express the assume-guarantee semantics of [8]. In Section 4.3, we show that the fixed point logic with chop (FLC)[1] can be translated into HFL so that any property expressible in FLC can also be expressed in HFL. In Section 4.4, we describe a representation of FLC formulas as transition systems over which we can diagonalize to construct properties inexpressible in FLC. In Section 4.5, we show that such a diagonalized property can be expressed in HFL, thereby establishing that HFL is strictly more expressive than FLC.

4.1 Definable Operations

Using standard dualities, we can define terms tt (the set of all states), $\varphi \wedge \psi$ (set intersection), and $[a]\varphi$; each of these terms is of type **Prop** and require φ, ψ to be of type **Prop**. Greatest fixed points, written $\nu(x: A)\varphi$, can be defined at arbitrary types A and require φ to be of type A with x appearing positively. For any type A , we can define a closed term $\text{bot}_A: A$ denoting the least element at the type A . Call a transition system *finitely strongly connected* if it is strongly connected under transitions that have labels belonging to some finite set. Over a finitely strongly connected transition system, we can define functions on the type **Prop** by case-analysis. Let $\varphi_1, \dots, \varphi_n$ be terms of type **Prop**,

and $\psi_1, \dots, \psi_n, \psi$ be terms of type $\text{Prop}^0 \rightarrow A$ for some type A . We can define a term $\text{case}^A(\varphi_1 \Rightarrow \psi_1, \dots, \varphi_n \Rightarrow \psi_n, \text{else} \Rightarrow \psi)$ of type $\text{Prop}^0 \rightarrow A$ denoting a function that when applied to a singleton set $\{s\}$ (s is a state), returns $\llbracket \psi_i \rrbracket(\{s\})$ if $s \in \llbracket \varphi_i \rrbracket - \llbracket \varphi_{i-1} \rrbracket - \dots - \llbracket \varphi_1 \rrbracket$ and returns $\llbracket \psi \rrbracket(\{s\})$ if $s \in S - \llbracket \varphi_n \rrbracket - \dots - \llbracket \varphi_1 \rrbracket$. We use $\text{case}^A(\varphi_1 \Rightarrow \psi_1, \dots, \varphi_n \Rightarrow \psi_n)$ as syntactic sugar for the missing else clause returning $\text{bot}_{\text{Prop}^0 \rightarrow A}$. Note that by its very definition, the case-defined function cannot be monotonic or antimonotonic in its argument (of type Prop).

4.2 Assume Guarantee Properties

In this section, we show how assume-guarantee properties can be expressed in HFL. The encoding directly follows the informal recursive definition presented in Section 2; the main interest here is illustrating its well-typedness and its type. We define the closed term AssGuar as

$$\begin{aligned} & \lambda(f^- : \text{Prop}^+ \rightarrow \text{Prop}) \lambda(g^+ : \text{Prop}^+ \rightarrow \text{Prop}) \\ & (\nu(z : \text{Prop}^- \rightarrow \text{Prop}^+ \rightarrow \text{Prop}) \lambda(x^- : \text{Prop}) \lambda(y^+ : \text{Prop}) (\neg x \vee y) \wedge z(fx)(gy) \\ &) \text{tt}(g \text{tt}) \end{aligned}$$

which is typable as $\text{AssGuar} : (\text{Prop}^+ \rightarrow \text{Prop})^- \rightarrow (\text{Prop}^+ \rightarrow \text{Prop})^+ \rightarrow \text{Prop}$. This typing judgement can be read as asserting that the assumption property (its first argument) and the guarantee property (its second argument) are required to be monotonic and that the assume-guarantee property itself varies antimonotonically in its assumption and monotonically in its guarantee. Constructing the type derivation for AssGuar is instructive in showing how these natural properties of AssGuar follow directly from the constraints imposed by the type system of Table 1.

The following proposition shows that the term AssGuar encodes assume-guarantee properties and establishes that HFL is closed under assume-guarantee specifications.

Proposition 3 (Expressibility of Assume-Guarantee). *Consider any transition system \mathcal{T} with state set S and any monotonic functions $A, G : 2^S \rightarrow 2^S$. Then we have that $\mathcal{T}[\llbracket \text{AssGuar} \rrbracket](A)(G) = \nu X.A(X) \triangleright \nu X.G(X)$.*

4.3 Translating FLC into HFL

Let $\mathcal{T} = (S, \{\overset{a}{\rightarrow} \mid a \in \text{Act}\}, L, s_0)$, \mathcal{P} , Act , and \mathcal{V} be as described in Section 3. The following grammar describes the syntax of FLC formulas, where $p \in \mathcal{P}$, $a \in \text{Act}$, $x \in \mathcal{V}$.

$$\varphi ::= \text{ff} \mid \text{tt} \mid p \mid \bar{p} \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle a \rangle \mid [a] \mid x \mid \mu x.\varphi \mid \nu x.\varphi \mid \text{term} \mid \varphi; \varphi$$

The formula \bar{p} is the negation of p ; thus, negation in FLC is only applicable to propositional constants. Formulas are interpreted in FLC as predicate transformers, *i.e.*, functions $f : 2^S \rightarrow 2^S$ that are monotonic with respect to the subset ordering. The formula term denotes the identity function, and the chop operator

Table 3. Translation of FLC into HFL

$\widehat{\text{ff}} = \lambda(z^+ : \text{Prop})\text{ff}$	$\widehat{\text{tt}} = \lambda(z^+ : \text{Prop})\text{tt}$
$\widehat{p} = \lambda(z^+ : \text{Prop})p$	$\widehat{\bar{p}} = \lambda(z^+ : \text{Prop})\neg p$
$\widehat{x} = x$	
$\widehat{\langle a \rangle} = \lambda(z^+ : \text{Prop})\langle a \rangle z$	$\widehat{[a]} = \lambda(z^+ : \text{Prop})[a]z$
$\widehat{\varphi_1 \vee \varphi_2} = \lambda(z^+ : \text{Prop})(\widehat{\varphi_1} z) \vee (\widehat{\varphi_2} z)$	$\widehat{\varphi_1 \wedge \varphi_2} = \lambda(z^+ : \text{Prop})(\widehat{\varphi_1} z) \wedge (\widehat{\varphi_2} z)$
$\widehat{\text{term}} = \lambda(z^+ : \text{Prop})z$	$\widehat{\varphi_1 ; \varphi_2} = \lambda(z^+ : \text{Prop})\widehat{\varphi_1}(\widehat{\varphi_2} z)$
$\widehat{\mu x. \varphi} = \mu(x : \text{Prop}^+ \rightarrow \text{Prop})\widehat{\varphi}$	$\widehat{\nu x. \varphi} = \nu(x : \text{Prop}^+ \rightarrow \text{Prop})\widehat{\varphi}$

; denotes function composition. An environment η for a formula φ is a map from variables to monotonic functions from 2^S to 2^S that is defined on all the free variables of φ . For such an environment η , the FLC-semantics of a formula, written, $\mathcal{T}[\varphi]^C \eta$ yields a monotonic function from 2^S to 2^S . The reader is referred to [1] for the details of this definition, though it should also be clear from the translation into HFL that we next describe. A transition system satisfies a closed FLC formula, written $\mathcal{T} \models_C \varphi$ iff $s_0 \in \mathcal{T}[\varphi]^C \emptyset(S)$, *i.e.*, the initial state is in the set obtained by applying the semantics to the full state set S . The superscript or subscript C refers to the semantics or satisfaction relation in the FLC logic.

Every FLC formula can be interpreted naturally as an HFL term of type $\text{Prop}^+ \rightarrow \text{Prop}$. Table 3 details the straightforward inductive translation of any FLC formula φ into an HFL term $\widehat{\varphi}$; it follows almost directly the semantics of FLC defined in [1]. The HFL term forms tt , $\varphi_1 \wedge \varphi_2$, $[a]\varphi$ and $\nu(x:A)\varphi$ used in the translation are the definable operations of Section 4.1, and the λ -bound variable z used in the translation of $\vee, \wedge, ;$ is one that does not appear free in the formula being translated. For an FLC formula φ , define the HFL context Γ_φ to be $x_1 : \text{Prop}^+ \rightarrow \text{Prop}, \dots, x_n : \text{Prop}^+ \rightarrow \text{Prop}$ for some enumeration x_1, \dots, x_n of the free variables of φ . The following theorem shows that the translation is well-typed and preserves the semantics.

Theorem 1. *For any FLC formula φ and transition system \mathcal{T} , we have the following properties:*

1. $\Gamma_\varphi \vdash \widehat{\varphi} : \text{Prop}^+ \rightarrow \text{Prop}$ is derivable.
2. For any FLC environment η for φ , we have that $\eta \models \Gamma_\varphi$ and

$$\mathcal{T}[\Gamma_\varphi \vdash \widehat{\varphi} : \text{Prop}^+ \rightarrow \text{Prop}]\eta = \mathcal{T}[\varphi]^C \eta$$

As a straightforward corollary, any property of transition systems expressed by an FLC formula φ can be expressed by the HFL formula $\widehat{\varphi} \text{tt}$. From the results established in [1], it then also follows that satisfiability and validity of HFL formulas is undecidable.

Corollary 2. *For any transition system \mathcal{T} and closed FLC formula φ , we have that $\mathcal{T} \models \widehat{\varphi} \text{tt}$ iff $\mathcal{T} \models_C \varphi$.*

4.4 Properties Inexpressible in FLC

Define the set $SF(\varphi)$ of subformulas of any FLC formula φ in the standard way with $SF(\sigma x.\varphi) = \{\sigma x.\varphi\} \cup SF(\varphi)$, where σ is μ or ν . Call an FLC formula φ well-named if each bound variable in the formula φ is distinct. In this case, there is a well-defined function $FP_\varphi: (\mathcal{V} \cap SF(\varphi)) \rightarrow SF(\varphi)$ that maps each variable $x \in SF(\varphi)$ to a unique formula of the form $\sigma x.\psi \in SF(\varphi)$ where σ is μ or ν . We identify four action names from the set Act which we call lc, rc, ev, and dm and let $A = \{lc, rc, ev, dm\}$. These four names can be read as “left child”, “right child”, “evaluation”, and “dummy”. We also identify propositional constants from \mathcal{P} that we will refer to by $p_l, p_v, p_\vee, p_\wedge, p_\mu, p_\nu, p_{term}, p_;$, and $p_{\langle a \rangle}, p_{[a]}$ for each $a \in A$.

We now give our representation of FLC formulas as labelled transition systems.

Definition 3. For any well-named FLC formula φ whose action names all belong to the set $A = \{lc, rc, ev, dm\}$, the transition system T^φ is defined to be $(SF(\varphi), \{\overset{a}{\rightarrow}\}, L, \varphi)$ where:

- The labelling function is defined according to the form of the formula: $L(\psi) = \{p_l\}$ if ψ is one of $\mathbf{tt}, \mathbf{ff}, p, \bar{p}$ for some $p \in \mathcal{P}$; $L(x) = \{p_v\}$; $L(\psi) = \{p_\psi\}$ if ψ is of the form $\mathbf{term}, \langle a \rangle$ or $[a]$ for $a \in A$; $L(\psi_1 O \psi_2) = \{p_O\}$ where O is one of $\vee, \wedge, ;$, and $L(\sigma x.\psi) = \{p_\sigma\}$ where σ is μ or ν .
- For any action name $a \notin A$, $\overset{a}{\rightarrow} = \emptyset$. The set $\overset{lc}{\rightarrow}$ includes pairs (ψ_1, ψ_2) where ψ_2 is of the form $\psi_1 O \psi'$, for some ψ' , or $\sigma x.\psi_1$, where O is one of $\vee, \wedge, ;$, and σ is one of μ, ν . The set $\overset{rc}{\rightarrow}$ includes pairs (ψ_1, ψ_2) where ψ_2 is of the form $\psi' O \psi_1$, for some ψ' and O one of $\vee, \wedge, ;$. The set $\overset{ev}{\rightarrow}$ includes pairs (ψ_1, ψ_2) which satisfy one of the four conditions: (1) ψ_2 is the formula \mathbf{tt} , (2) ψ_2 is p for some $p \in \mathcal{P}$, and $p \in L(\psi_1)$, (3) ψ_2 is \bar{p} for some $p \in \mathcal{P}$, and $p \notin L(\psi_1)$, (4) ψ_2 is a variable x and ψ_1 is $FP_\varphi(x)$. Finally, the set $\overset{dm}{\rightarrow}$ includes pairs (φ, ψ) (where φ is the formula being represented) and ψ is one of $\mathbf{tt}, \mathbf{ff}, p, \bar{p}, x, \langle a \rangle, [a]$ for $a \in A$.

The transition system T^φ is finite and strongly connected by the transitions from A .

Definition 3 can be intuitively understood as follows. The transition system for a formula φ is essentially its parse tree (with sharing of the trees for common subformulas) with edges directed from child to parent. These parse tree edges are labelled with the lc, rc transitions, and the propositional labeling indicates the outermost construct of the corresponding subformula (with p_l standing for constant literals and p_v for variables). Additionally, we have transitions labeled ev to the constant literals $\mathbf{tt}, \mathbf{ff}, p, \bar{p}$ from all the states in which the literals hold (\mathbf{ff} does not hold anywhere), and to variables x from their defining fixed point formula. Note that because the formula is well-named, there is exactly one transition labeled ev to every node x . Finally, the dummy transition edges dm are added from the root to every leaf node — the only purpose of these edges is

Table 4. Encoding FLC Diagonalization in HFL

$$\begin{aligned}
\text{decode} &\triangleq \mu(d: (\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop}))^+ \rightarrow (\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop}))) \\
&\quad \lambda(e^+: (\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop}))) \\
&\quad \text{case}(p_l \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) \langle \text{ev} \rangle x, \\
&\quad \quad p_v \Rightarrow \lambda(x^0: \text{Prop}) e \langle (\text{ev}) x \rangle, \\
&\quad \quad p_{\langle a \rangle} \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) \langle a \rangle z \quad a \in \{\text{lc}, \text{rc}, \text{ev}, \text{dm}\}, \\
&\quad \quad p_{[a]} \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) [a] z \quad a \in \{\text{lc}, \text{rc}, \text{ev}, \text{dm}\}, \\
&\quad \quad p_{\vee} \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) (d e \langle (\text{lc}) x \rangle z) \vee (d e \langle (\text{rc}) x \rangle z), \\
&\quad \quad p_{\wedge} \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) (d e \langle (\text{lc}) x \rangle z) \wedge (d e \langle (\text{rc}) x \rangle z), \\
&\quad \quad p_{\text{term}} \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) z, \\
&\quad \quad p_i \Rightarrow \lambda(x^0: \text{Prop}) \lambda(z^+: \text{Prop}) (d e \langle (\text{lc}) x \rangle) ((d e \langle (\text{rc}) x \rangle) z), \\
&\quad \quad p_{\mu} \Rightarrow \lambda(x^0: \text{Prop}) \mu(f: \text{Prop}^+ \rightarrow \text{Prop}) \\
&\quad \quad \quad d (\text{case}(x \Rightarrow \lambda(z^0: \text{Prop}) f, \text{else} \Rightarrow e)) \langle (\text{lc}) x \rangle, \\
&\quad \quad p_{\nu} \Rightarrow \lambda(x^0: \text{Prop}) \nu(f: \text{Prop}^+ \rightarrow \text{Prop}) \\
&\quad \quad \quad d (\text{case}(x \Rightarrow \lambda(z^0: \text{Prop}) f, \text{else} \Rightarrow e)) \langle (\text{lc}) x \rangle \\
&\quad) \\
\text{init} &\triangleq \langle \text{dm} \rangle \text{tt} \\
\text{flc-sem} &\triangleq \text{decode bot}_{\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop})} \text{init} \\
\text{flc-diag} &\triangleq \neg(\text{flc-sem tt})
\end{aligned}$$

to make the transition system strongly connected (thus allowing us to use the case-construct over these transition systems). It also allows us to identify the initial state (as the only one that has a dm transition enabled).

By diagonalizing over this representation, we obtain properties of finite transition systems that cannot be expressed in FLC.

Theorem 2. *Let C be any property of finite transition systems such that for any closed well-named formula φ with actions from A , we have that $T^\varphi \in C$ iff $T^\varphi \not\models_C \varphi$. Then C is not expressible in FLC.*

Note that the inexpressible property described by Theorem 2 is unconstrained on transition systems that are not a T^φ .

4.5 HFL is more expressive than FLC

We now show how to construct a formula in HFL that expresses a property of the form prescribed by Theorem 2. Table 4 defines HFL terms whose types are as follows:

$$\begin{aligned}
\text{decode} &: (\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop}))^+ \rightarrow (\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop})) \\
\text{init} &: \text{Prop} \\
\text{flc-sem} &: \text{Prop}^+ \rightarrow \text{Prop} \\
\text{flc-diag} &: \text{Prop}
\end{aligned}$$

with the HFL formula `flc-diag` expressing a property of the form prescribed by Theorem 2. The properties of the terms `decode` and `init` defined in Table 4 are given by the following theorem:

Theorem 3. *Let φ be a closed well-named FLC formula over the action set A .*

1. *Consider any subformula $\psi \in SF(\varphi)$ and FLC environment $\eta: \mathcal{V} \rightarrow (2^S \rightarrow 2^S)$. For any function $F_\eta \in \mathcal{T}^\varphi[\text{Prop}^0 \rightarrow (\text{Prop}^+ \rightarrow \text{Prop})]$ such that $F_\eta(\{FP_\varphi(x)\}) = \eta(x)$ for every x free in ψ , $\mathcal{T}^\varphi[\text{decode}](F_\eta)(\{\psi\}) = \mathcal{T}^\varphi[\psi]^c \eta$*
2. $\mathcal{T}^\varphi[\text{init}] = \{\varphi\}$

The heart of the construction is `decode` that shows how to decode (in HFL) the transition system representing an FLC formula. Its definition given in Table 4 is easiest understood on the basis of its property given in Theorem 3, with the λ -bound variable e read as standing for the function (F_η) representing an environment η , and the λ -bound variable x in each of the cases read as standing for the singleton set $\{\psi\}$. On an argument $\{\psi\}$, the formula ψ is decoded in cases according to its outermost form which in turn is inferred based on which of the propositional constants p_l, p_v, \dots holds in x (standing for $\{\psi\}$). For all constructs other than variables and fixed points, their corresponding cases can be understood by close analogy with the HFL-translation of these constructs given in Table 3 together with the understanding that $\langle \text{lc} \rangle x$ and $\langle \text{rc} \rangle x$ yield singleton sets including the corresponding subformulas of ψ , and that for constant literals term $\langle \text{ev} \rangle x$ yields the set of states in which the literal ψ holds. If ψ is a variable, we evaluate the environment on the set $\{FP_\varphi(\psi)\}$ (as given by the property of F_η) which is yielded by the term $\langle \text{ev} \rangle x$. If ψ is a fixed point formula, we correspondingly bind (using μ or ν) a new variable f and decode the subformula of ψ (given by $\langle \text{lc} \rangle x$) but in an environment that is obtained by modifying the current environment e to map $\{\psi\}$ (given by x) to f (the `case-term` used for the environment argument to d in the fixed point cases yields this updated environment). This ensures that when decoding the subformulas of ψ any use of a variable corresponding to this recursive definition will be decoded as f . The decoding of the fixed-point cases explains the presence of the environment argument in defining `decode`. Finally, it is worth noting that: (1) `decode` is a recursive definition of a higher-order function, and (2) because `decode` is defined by case-analysis, it is not monotonic in the argument x (standing for the formula being decoded). These features of HFL are therefore crucial to its definition.

As an easy corollary of Theorem 3, we get the relevant properties of the terms `flc-sem` and `flc-diag`.

Corollary 3. *For any closed well-named FLC formula φ over the action set A , we have that*

1. $\mathcal{T}^\varphi[\text{flc-sem}] = \mathcal{T}^\varphi[\varphi]^c \emptyset$.
2. $\mathcal{T}^\varphi \models \text{flc-diag}$ iff $\mathcal{T}^\varphi \not\models_c \varphi$.

Combined with Theorem 2 this gives us that the HFL formula `flc-diag` is a characteristic formula for a property of finite transition systems that is inexpressible in FLC, and thus HFL is strictly more expressive than FLC even over finite transition systems.

References

1. Müller-Olm, M.: A Modal Fixpoint Logic with Chop. In: Proceedings of the Symposium on the Theoretical Aspects of Computer Science. Volume 1563 of Lecture Notes in Computer Science., Springer (1999) 510–520
2. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems. NATO ASI Series. Springer-Verlag (1984) 123–144
3. Misra, J., Chandy, K.M.: Proofs of network processes. IEEE Transactions on Software Engineering **SE-7** (1981) 417–426
4. Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems **15** (1993) 73–132
5. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems **17** (1995) 507–534
6. McMillan, K.: Circular compositional reasoning about liveness. In Pierre, L., Kropf, T., eds.: CHARME 99: Correct Hardware Design and Verification. Volume 1703 of Lecture Notes in Computer Science., Springer-Verlag (1999) 342–345
7. Henzinger, T.A., Qadeer, S., Rajamani, S.K., Tasiran, S.: An assume-guarantee rule for checking simulation. In Gopalakrishnan, G., Windley, P., eds.: FMCAD 98: Formal Methods in Computer-aided Design. Volume 1522 of Lecture Notes in Computer Science., Springer-Verlag (1998) 421–432
8. Viswanathan, M., Viswanathan, R.: Foundations of Circular Compositional Reasoning. In: Proceedings of the International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, Springer (2001)
9. Kozen, D.: Results on the propositional μ -calculus. Theoretical Computer Science **27** (1983) 333–354
10. Stirling, C.: Modal and Temporal Logics. In: Handbook of Logic in Computer Science. Volume 2. Clarendon Press, Oxford, UK (1992) 477–563
11. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model checking for the μ -calculus and its fragments. Theoretical Computer Science **258** (2001) 491–522
12. Lange, M., Stirling, C.: Model Checking Fixed Point Logic with Chop. In: Proceedings of the Foundations of Software Science and Computation Structures. Volume 2303 of Lecture Notes in Computer Science., Springer (2002) 250–263
13. Lange, M.: Local model checking games for fixed point logic with chop. In: Proceedings of the Conference on Concurrency Theory, CONCUR’02. Volume 2421 of Lecture Notes in Computer Science., Springer (2002) 240–254
14. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
15. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on Infinite Structures. In: Handbook of Process Algebra. Elsevier Science Publishers (2001) 545–623
16. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial algorithm for deciding bisimilarity of normed context-free processes. Theoretical Computer Science **15** (1996) 143–159
17. Sénizergues, G.: Decidability of bisimulation equivalence for equations graphs of finite out-degree. In: Proceedings of the IEEE Symposium on the Foundations of Computer Science. (1998) 120–129
18. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5** (1955) 285–309