Domains for Higher-Order Games[*][†]

Matthew Hague[1], Roland Meyer[‡][2], and Sebastian Muskalla[3]

1 **Royal Holloway University of London, United Kingdom**
matthew.hague@rhul.ac.uk

2 **TU Braunschweig, Germany**
roland.meyer@tu-braunschweig.de

3 **TU Braunschweig, Germany**
s.muskalla@tu-braunschweig.de

Abstract

We study two-player inclusion games played over word-generating higher-order recursion schemes. While inclusion checks are known to capture verification problems, two-player games generalize this relationship to program synthesis. In such games, non-terminals of the grammar are controlled by opposing players. The goal of the existential player is to avoid producing a word that lies outside of a regular language of safe words.

We contribute a new domain that provides a representation of the winning region of such games. Our domain is based on (functions over) potentially infinite Boolean formulas with words as atomic propositions. We develop an abstract interpretation framework that we instantiate to abstract this domain into a domain where the propositions are replaced by states of a finite automaton. This second domain is therefore finite and we obtain, via standard fixed-point techniques, a direct algorithm for the analysis of two-player inclusion games. We show, via a second instantiation of the framework, that our finite domain can be optimized, leading to a $(k+1)\mathsf{EXP}$ algorithm for order-k recursion schemes. We give a matching lower bound, showing that our approach is optimal. Since our approach is based on standard Kleene iteration, existing techniques and tools for fixed-point computations can be applied.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Higher-order recursion schemes, games, semantics, abstract interpretation, fixed points.

Digital Object Identifier 10.4230/LIPIcs.MFCS.2017.59

1 Introduction

Inclusion checking has recently received considerable attention [53, 22, 1, 2, 36]. One of the reasons is a new verification loop, which invokes inclusion as a subroutine in an iterative fashion. The loop has been proposed by Podelski et al. for the safety verification of recursive programs [32], and then been generalized to parallel and parameterized programs [42, 20, 18] and to liveness [19]. The idea of Podelski's loop is to iteratively approximate unsound data flow in the program of interest, and add the approximations to the specification. Consider a program with control-flow language *CF* that is supposed to satisfy a safety specification

[*] The full version is available as technical report [28].
[†] This work was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1]. The work instigated while some of the authors were visiting the Institute for Mathematical Sciences, National University of Singapore in 2016. The visit was partially supported by the Institute.
[‡] A part of the work was carried out when the author was at Aalto University.

given by a regular language R. If the check $CF \subseteq R$ succeeds, then the program is correct as the data flow only restricts the set of computations. If a computation $w \in CF$ is found that lies outside R, then it depends on the data flow whether the program is correct. If data is handled correctly, w is a counterexample to R. Otherwise, w is generalized to a regular language S of infeasible computations. We set $R = R \cup S$ and repeat the procedure.

Podelski's loop has also been generalized to synthesis [35, 44]. In that setting, the program is assumed to have two kinds of non-determinism. Some of the non-deterministic transitions are understood to be controlled by the environment. They provide inputs that the system has to react to, and are also referred to as demonic non-determinism. In contrast, the so-called angelic non-determinism are the alternatives of the system to react to an input. The synthesis problem is to devise a controller that resolves the angelic non-determinism in a way that a given safety specification is met. Technically, the synthesis problem corresponds to a two-player perfect information game, and the controller implements a winning strategy for the system player. When generalizing Podelski's loop to the synthesis problem, the inclusion check thus amounts to solving a strategy-synthesis problem.

Our motivation is to synthesize functional programs with Podelski's loop. We assume the program to be given as a non-deterministic higher-order recursion scheme where the non-terminals are assigned to two players. One player is the system player who tries to enforce the derivation of words that belong to a given regular language. The other player is the environment, trying to derive a word outside the language. The use of the corresponding strategy-synthesis algorithm in Podelski's loop comes with three characteristics: (1) The algorithm is invoked iteratively, (2) the program is large and the specification is small, and (3) the specification is non-deterministic. The first point means that the strategy synthesis should not rely on costly precomputation. Moreover, it should have the chance to terminate early. The second says that the cost of the computation should depend on the size of the specification, not on the size of the program. Computations on the program, in particular iterative ones, should be avoided. Together with the third characteristic, these two consequences rule out reductions to reachability games. The required determinization would mean a costly precomputation, and the reduction to reachability would mean a product with the program. This discussion in particular forbids a reduction of the strategy-synthesis problem to higher-order model checking [45], which indeed can be achieved (see the full version [28] for a comparison to intersection types [41]). Instead, we need a strategy synthesis that can directly deal with non-deterministic specifications.

We show that the winning region of a higher-order inclusion game wrt. a non-deterministic right-hand side can be computed with a standard fixed-point iteration. Our contribution is a domain suitable for this computation. The key idea is to use Boolean formulas whose atomic propositions are the states of the targeted finite automaton. While a formula-based domain has recently been proposed for context-free inclusion games [35] (and generalized to infinite words [44]), the generalization to higher-order is new. Consider a non-terminal that is ground and for which we have computed a formula. The Boolean structure reflects the alternation among the players in the plays that start from this non-terminal. The words generated along the plays are abstracted to sets of states from which these words can be accepted. Determining the winner of the game is done by evaluating the formula when sets of states containing the initial state are assigned the value true. To our surprise, the above domain did not give the optimal complexity. Instead, it was possible to further optimize it by resolving the determinization information. Intuitively, the existential player can also resolve the non-determinism captured by a set. Crucially, our approach handles the non-determinism of the specification inside the analysis, without preprocessing.

Besides offering the characteristics that are needed for Podelski's loop, our development also contributes to the research program of *effective denotational semantics*, as recently proposed by Salvati and Walukiewicz [51] as well as Grellois and Melliès [24, 24], with [5, 48] being early works in this field. The idea is to solve verification problems by computing the semantics of a program in a suitable domain. Salvati and Walukiewicz studied the expressiveness of greatest fixed-point semantics and their correspondence to automata [51], and constructions of enriched Scott models for parity conditions [50, 49]. A similar line of investigation has been followed in recent work by Grellois and Melliès [25, 26]. Hofmann and Chen considered the verification of more restricted ω-path properties with a focus on the domain [33]. They show that explicit automata constructions can be avoided and give a domain that directly captures subsets (so-called patches) of the ω-language. The work has been generalized to higher order [34]. Our contribution is related in that we focus on the domain (suitable for capturing plays).

Besides the domain, the correctness proof may be of interest. We employ an exact fixed-point transfer result as known from abstract interpretation. First, we give a semantic characterization showing that the winning region can be captured by an infinite model (a greatest fixed point). This domain has as elements (potentially infinite) sets of (finite) Boolean formulas. The formulas capture plays (up to a certain depth) and the atomic propositions are terminal words. The infinite set structure is to avoid infinite syntax. Then we employ the exact fixed-point transfer result to replace the terminals by states and get rid of the sets. The final step is another exact fixed-point transfer that justifies the optimization. We give a matching lower bound. The problem is $(k+1)$EXP-complete for order-k schemes.

Related Work. The relationship between recursion schemes and extensions of pushdown automata has been well studied [16, 17, 37, 29]. This means algorithms for recursion schemes can be transferred to extensions of pushdown automata and vice versa. In the sequel, we will use *pushdown automata* to refer to pushdown automata and their family of extensions.

The decidability of Monadic Second Order Logic (MSO) over trees generated by recursion schemes was first settled in the restricted case of *safe* schemes by Knapik *et al.* [37] and independently by Caucal [14]. This result was generalized to all schemes by Ong [45]. Both of these results consider *deterministic* schemes only.

Related results have also been obtained in the consideration of games played over the configuration graphs of pushdown automata [52, 13, 38, 29]. Of particular interest are *saturation* methods for pushdown games [7, 21, 12, 8, 30, 31, 9]. In these works, automata representing sets of winning configurations are constructed using fixed-point computations.

A related approach pioneered by Kobayashi et al. operating directly on schemes is that of *intersection types* [40, 41], where types embedding a property automaton are assigned to terms of a scheme. Recently, saturation techniques were transferred to intersection types by Broadbent and Kobayashi [10]. The typing algorithm is then a least fixed-point computation analogous to an optimized version of our Kleene iteration, restricted to deterministic schemes. This has led to one of the most competitive model-checking tools for schemes [39].

One may reduce our language inclusion problems to many of the above works. E.g. from an inclusion game for schemes, we may build a game over an equivalent kind of pushdown automaton and take the product with a determinization of the NFA. This obtains a reachability game over a pushdown automaton that can be solved by any of the above methods. However, such constructions are undesirable for iterative invocations as in Podelski's loop.

We already discussed the relationship to model-theoretic verification algorithms. Abstract interpretation has also been used by Ramsay [47], Salvati and Walukiewicz [50, 49], and

Grellois and Melliès [24, 23] for verification. The former used a Galois connection between safety properties (concrete) and equivalence classes of intersection types (abstract) to recreate decidability results known in the literature. The latter two strands gives a semantics capable of computing properties expressed in MSO. Indeed, abstract interpretation has long been used for static analysis of higher-order programs [4].

2 Preliminaries

Complete Partial Orders. Let $(D, \leq)$ be a *partial order* with set D and (partial) ordering $\leq$ on D. We call $(D, \leq)$ *pointed* if there is a greatest element, called the *top element* and denoted by $\top \in D$. A *descending chain* in D is a sequence $(d_i)_{i \in \mathbb{N}}$ of elements in D with $d_i \geq d_{i+1}$. We call $(D, \leq)$ *ω-complete* if every descending chain has a greatest lower bound, called the *meet* or the *infimum*, and denoted by $\bigsqcap_{i \in \mathbb{N}} d_i$. If $(D, \leq)$ is pointed and ω-complete, we call it a *pointed ω-complete partial order (cppo)*. In the following, we will only consider partial orders that are cppos. Note, cppo is usually used to refer to the dual concept, i.e. partial orders with a least element and least upper bounds for ascending chains.

A function $f : D \to D$ is *$\sqcap$-continuous* if for all descending chains $(d_i)_{i \in \mathbb{N}}$ we have $f(\bigsqcap_{i \in \mathbb{N}} d_i) = \bigsqcap_{i \in \mathbb{N}} f(d_i)$. We call a function $f : D \to D$ *monotonic* if for all $d, d' \in D$, $d \leq d'$ implies $f(d) \leq f(d')$. Any function that is $\sqcap$-continuous is also monotonic. For a monotonic function, $\top \geq f(\top) \geq f^2(\top) = f(f(\top)) \geq f^3(\top) \geq \ldots$ is a descending chain.

If the function is $\sqcap$-continuous, then $\bigsqcap_{i \in \mathbb{N}} f^i(\top)$ is by Kleene's theorem the greatest fixed point of f, i.e. $f(\bigsqcap_{i \in \mathbb{N}} f^i(\top)) = \bigsqcap_{i \in \mathbb{N}} f^i(\top)$ and $\bigsqcap_{i \in \mathbb{N}} f^i(\top)$ is larger than any other element d with $f(d) = d$. We also say $\bigsqcap_{i \in \mathbb{N}} f^i(\top)$ is the greatest solution to the equation $x = f(x)$.

A lattice satisfies the *descending chain condition (DCC)* if every descending chain has to be stationary at some point. In this case $\bigsqcap_{i \in \mathbb{N}} f^i(\top) = \bigsqcap_{i=0}^{i_0} f^i(\top)$ for some index i_0 in $\mathbb{N}$. With this, we can compute the greatest fixed point: Starting with $\top$, we iteratively apply f until the result does not change. This process is called *Kleene iteration*. Note that finite cppos, i.e. with finitely many elements in D, trivially satisfy the descending chain condition.

Finite Automata. A *non-deterministic finite automaton (NFA)* is a tuple $A = (Q_{NFA}, \Gamma, \delta, q_0, Q_f)$ where Q_{NFA} is a finite set of states, Γ is a finite alphabet, $\delta \subseteq Q_{NFA} \times \Gamma \times Q_{NFA}$ is a (non-deterministic) transition relation, $q_0 \in Q_{NFA}$ is the initial state, and $Q_f \subseteq Q_{NFA}$ is a set of final states. We write $q \xrightarrow{a} q'$ to denote $(q, a, q') \in \delta$. Moreover, given a word $w = a_1 \cdots a_\ell$, we write $q \xrightarrow{w} q'$ whenever there is a sequence of transitions, also called *run*, $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots \xrightarrow{a_\ell} q_{\ell+1}$ with $q_1 = q$ and $q_{\ell+1} = q'$. The run is accepting if $q = q_0$ and $q' \in Q_f$. The language of A is $\mathcal{L}(A) = \{w \mid q_0 \xrightarrow{w} q \in Q_f\}$.

3 Higher-Order Recursion Schemes

We introduce higher-order recursion schemes, *schemes* for short, following the presentation in [27]. Schemes can be understood as grammars generating the computation trees of programs in a functional language. As is common in functional languages, we need a typing discipline. To avoid confusion with type-based approaches to higher-order model checking [40, 46, 41], we refer to types as *kinds*. Kinds define the functionality of terms, without specifying the data domain. Technically, the only data domain is the ground kind o, from which (potentially higher-order) function kinds are derived by composition:

$$\kappa \ ::= \ o \ \mid \ (\kappa_1 \to \kappa_2) \ .$$

We usually omit the brackets and assume that the arrow associates to the right. The number of arguments to a kind is called the *arity*. The *order* defines the functionality of the arguments: A first-order kind defines functions that act on values, a second-order kind functions that expect functions as parameters. Formally, we have

$$\begin{aligned} \mathsf{arity}(o) &= 0, & \mathsf{order}(o) &= 0, \\ \mathsf{arity}(\kappa_1 \to \kappa_2) &= \mathsf{arity}(\kappa_2) + 1, & \mathsf{order}(\kappa_1 \to \kappa_2) &= \max(\mathsf{order}(\kappa_1) + 1, \mathsf{order}(\kappa_2)) \ . \end{aligned}$$

Let K be the set of all kinds. Higher-order recursion schemes assign kinds to symbols from different alphabets, namely non-terminals, terminals, and variables. Let Γ be a set of such *kinded symbols*. For each kind κ, we denote by Γ^κ the restriction of Γ to the symbols with kind κ. The *terms* $\mathcal{T}^\kappa(\Gamma)$ of kind κ over Γ are defined by simultaneous induction over all kinds. They form the smallest set satisfying

1. $\Gamma^\kappa \subseteq \mathcal{T}^\kappa(\Gamma)$,
2. $\bigcup_{\kappa_1}\{t\ v \mid t \in \mathcal{T}^{\kappa_1 \to \kappa_2}(\Gamma), v \in \mathcal{T}^{\kappa_1}(\Gamma)\} \subseteq \mathcal{T}^{\kappa_2}(\Gamma)$, and
3. $\{\lambda x.t \mid x \in \mathcal{T}^{\kappa_1}(\Gamma), t \in \mathcal{T}^{\kappa_2}(\Gamma)\} \subseteq \mathcal{T}^{\kappa_1 \to \kappa_2}(\Gamma)$.

If term t is of kind κ, we also write $t\colon \kappa$. We use $\mathcal{T}(\Gamma)$ for the set of all terms over Γ. We say a term is λ-*free* if it contains no sub-term of the form $\lambda x.t$. A term is *variable-closed* if all occurring variables are bound by a preceding λ-expression.

▶ **Definition 1.** A *higher-order recursion scheme*, (*scheme* for short), is a tuple $G = (V, N, T, R, S)$, where V is a finite set of kinded symbols called *variables*, T is a finite set of kinded symbols called *terminals*, and N is a finite set of kinded symbols called *non-terminals* with $S \in N$ the *initial symbol*. The sets V, T, and N are pairwise disjoint. The finite set R consists of *rewriting rules* of the form $F = \lambda x_1 \dots \lambda x_n.e$, where $F \in N$ is a non-terminal of kind $\kappa_1 \to \dots \kappa_n \to o$, $x_1, \dots, x_n \in V$ are variables of the required kinds, and e is a λ-free, variable-closed term of ground kind from $\mathcal{T}^o(T \uplus N \uplus \{x_1\colon \kappa_1, \dots, x_n\colon \kappa_n\})$.

The semantics of G is defined by rewriting subterms according to the rules in R. A *context* is a term $C[\bullet] \in \mathcal{T}(\Gamma \uplus \{\bullet\colon o\})$ in which $\bullet$ occurs exactly once. Given a context $C[\bullet]$ and a term $t : o$, we obtain $C[t]$ by replacing the unique occurrence of $\bullet$ in $C[\bullet]$ by t. With this, $t \Rightarrow_G t'$ if there is a context $C[\bullet]$, a rule $F = \lambda x_1 \dots \lambda x_n.e$, and a term $F\ t_1\ \dots\ t_n : o$ such that $t = C[F\ t_1\ \dots\ t_n]$ and $t' = C[e[x_1 \mapsto t_1, \dots, x_n \mapsto t_n]]$. In other words, we replace one occurrence of F in t by a right-hand side of a rewriting rule, while properly instantiating the variables. We call such a replaceable $F\ t_1\ \dots\ t_n$ a *reducible expression (redex)*. The rewriting step is *outermost to innermost (OI)* if there is no redex that contains the rewritten one as a proper subterm. The OI-language $\mathcal{L}(G)$ of G is the set of all (finite, ranked, labeled) trees T over the terminal symbols that can be created from the initial symbol S via OI-rewriting steps. We will restrict the rewriting relation to OI-rewritings in the rest of this paper. Note, all words derivable by IO-rewriting are also derivable with OI-rewriting.

Word-Generating Schemes. We consider *word-generating schemes*, i.e. schemes with terminals $T \uplus \{\$: o\}$ where exactly one terminal symbol $\$$ has kind o and all others are of kind $o \to o$. The generated trees have the shape $a_1\ (a_2\ (\cdots\ (a_k\ \$)))$, which we understand as the finite word $a_1 a_2 \dots a_k \in T^*$. We also see $\mathcal{L}(G)$ as a language of finite words.

Determinism. The above schemes are non-deterministic in that several rules may rewrite a non-terminal. We associate with a non-deterministic scheme $G = (V, N, T, R, S)$ a deterministic scheme G^{det} with exactly one rule per non-terminal. Intuitively, G^{det} makes the non-determinism explicit with new terminal symbols.

Formally, let $F : \kappa$ be a non-terminal with rules $F = t_1$ to $F = t_\ell$. We may assume each $t_i = \lambda x_1 \dots \lambda x_k.e_i$, where e_i is λ-free. We introduce a new terminal symbol $op_F : o \to o \to \dots \to o$ of arity ℓ. Let the set of all these terminals be $T^{det} = \{op_F \mid F \in N\}$. The set of rules R^{det} now consists of a single rule for each non-terminal, namely $F = \lambda x_1 \dots \lambda x_k.op_F\ e_1\ \cdots\ e_\ell$. The original rules in R are removed. This yields $G^{det} = (V, N, T \uplus T^{det}, R^{det}, S)$. The advantage of resolving the non-determinism explicitly is that we can give a semantics to non-deterministic choices that depends on the non-terminal instead of having to treat non-determinism uniformly.

Semantics. Let $G = (V, N, T, R, S)$ be a deterministic scheme. A *model* of G is a pair $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, where $\mathcal{D}$ is a family of domains $(\mathcal{D}(\kappa))_{\kappa \in K}$ that satisfies the following: $\mathcal{D}(o)$ is a cppo and $\mathcal{D}(\kappa_1 \to \kappa_2) = \mathit{Cont}(\mathcal{D}(\kappa_1), \mathcal{D}(\kappa_2))$. Here, $\mathit{Cont}(A, B)$ is the set of all $\sqcap$-continuous functions from domain A to B. We comment on this cppo in a moment. The interpretation $\mathcal{I} : T \to \mathcal{D}$ assigns to each terminal $s : \kappa$ an element $\mathcal{I}(s) \in \mathcal{D}(\kappa)$.

The ordering on functions is defined component-wise, $f \leq_{\kappa_1 \to \kappa_2} g$ if $(f\ x) \leq_{\kappa_2} (g\ x)$ for all $x \in \mathcal{D}(\kappa_1)$. For each κ, we denote the top element of $\mathcal{D}(\kappa)$ by $\top_\kappa$. For the ground kind, $\top_o$ exists since $\mathcal{D}(\kappa)$ is a cppo, and $\top_{\kappa_1 \to \kappa_2}$ is the function that maps every argument to $\top_{\kappa_2}$. The meet of a descending chain of functions $(f_i)_{i \in \mathbb{N}}$ is the function defined by $(\bigsqcap_{\kappa_1 \to \kappa_2} (f_i)_{i \in \mathbb{N}})\ x = \bigsqcap_{\kappa_2} (f_i\ x)_{i \in \mathbb{N}}$. Note that the sequence on the right-hand side is a descending chain.

The *semantics of terms* defined by a model is a function

$$\mathcal{M}[\![-]\!] : \mathcal{T} \to (N \uplus V \nrightarrow \mathcal{D}) \to \mathcal{D}\ .$$

that assigns to each term built over the non-terminals and terminals again a function. This function expects a valuation $\nu : N \uplus V \nrightarrow \mathcal{D}$ and returns an element from the domain. A valuation is a partial function that is defined on all non-terminals and the free variables. We lift $\sqcap$ to descending chains of valuations with $(\bigsqcap_{i \in \mathbb{N}} \nu_i)(y) = \bigsqcap_{i \in \mathbb{N}} (\nu_i(y))$ for $y \in N \uplus V$. We obtain that the set of such valuations is a cppo where the greatest elements are those valuations which assign the greatest elements of the appropriate domain to all arguments.

Since the right-hand sides of the rules in the scheme are variable-closed, we do not need a variable valuation for them. We need the variable valuation, however, whenever we proceed by induction on the structure of terms. The semantics is defined by such an induction:

$$\begin{aligned}
&\mathcal{M}[\![s]\!]\ \nu = \mathcal{I}(s) \quad \mathcal{M}[\![F]\!]\ \nu = \nu(F) \qquad && \mathcal{M}[\![t_1\ t_2]\!]\ \nu = (\mathcal{M}[\![t_1]\!]\ \nu)\ (\mathcal{M}[\![t_2]\!]\ \nu) \\
&\mathcal{M}[\![x]\!]\ \nu = \nu(x) && \mathcal{M}[\![\lambda x : \kappa.t_1]\!]\ \nu = d \in \mathcal{D}(\kappa) \mapsto \mathcal{M}[\![t_1]\!]\ \nu[x \mapsto d]\ .
\end{aligned}$$

We show that $\mathcal{M}[\![t]\!]$ is $\sqcap$-continuous for all terms t. This follows from continuity of the functions in the domain, but requires some care when handling application.

▶ **Proposition 2.** *For all t, $\mathcal{M}[\![t]\!]$ is $\sqcap$-continuous (in ν) over the respective lattice.*

Given $\mathcal{M}$, the rules $F_1 = t_1, \dots, F_k = t_k$ of the (deterministic) scheme give a function

$$\mathit{rhs}_\mathcal{M} : (N \to \mathcal{D}) \to (N \to \mathcal{D})\ , \qquad \text{where} \quad \mathit{rhs}_\mathcal{M}(\nu)(F_j) = \mathcal{M}[\![t_j]\!]\ \nu\ .$$

Since the right-hand sides are variable-closed, the $\mathcal{M}[\![t_j]\!]$ are functions in the non-terminals. Provided $\mathcal{M}[\![t_1]\!]$ to $\mathcal{M}[\![t_k]\!]$ are $\sqcap$-continuous (in the valuation of the non-terminals), the function $\mathit{rhs}_\mathcal{M}$ will be $\sqcap$-continuous. This allows us to apply Kleene iteration as follows. The initial value is the greatest element $\sigma^0_\mathcal{M}$ where $\sigma^0_\mathcal{M}(F_j) = \top_j$ with $\top_j$ the top element of $\mathcal{D}(\kappa_j)$. The $(i+1)$th approximant is computed by evaluating the right-hand side at the ith solution, $\sigma^{i+1}_{\mathcal{M}} = rhs_{\mathcal{M}}(\sigma^{i}_{\mathcal{M}})$. The greatest fixed point is the tuple $\sigma_{\mathcal{M}}$ defined below. It can be understood as the greatest solution to the equation $\nu = rhs_{\mathcal{M}}(\nu)$. We call this greatest solution $\sigma_{\mathcal{M}}$ the *semantics of the scheme* in the model.

$$\sigma_{\mathcal{M}} = \bigsqcap_{i\in\mathbb{N}} \sigma^{i}_{\mathcal{M}} = \bigsqcap_{i\in\mathbb{N}} rhs^{i}_{\mathcal{M}}(\sigma^{0}_{\mathcal{M}})$$

4 Higher-Order Inclusion Games

Our goal is to solve higher-order games, whose arena is defined by a scheme. We assume that the derivation process is controlled by two players. To this end, we divide the non-terminals of a word-generating scheme into those owned by the existential player $\Diamond$ and those owned by the universal player $\Box$. Whenever a non-terminal is to be replaced during the derivation, it is the owner who chooses which rule to apply. The winning condition is given by an automaton A, Player $\Diamond$ attempts to produce a word that is in $\mathcal{L}(A)$, while Player $\Box$ attempts to produce a word outside of $\mathcal{L}(A)$.

▶ **Definition 3.** A *higher-order game* is a triple $\mathcal{G} = (G, A, O)$ where G is a word-generating scheme, A is an NFA, $O : N \to \{\Diamond, \Box\}$ is a partitioning of the non-terminals of G.

A play of the game is a sequence of OI-rewriting steps. Since terms generate words, it is unambiguous which term forms the next redex to be rewritten. In particular, all terms are of the form $a_1(a_2(\cdots(a_k(t))))$, where t is either \$ or a redex $F\ t_1\ \cdots\ t_m$. If $O(F) = \Diamond$ then Player $\Diamond$ chooses a rule $F = \lambda x_1 \ldots \lambda x_m.e$ to apply, else Player $\Box$ chooses the rule. This moves the play to $a_1\ (a_2\ (\cdots\ (a_k\ e[x_1 \mapsto t_1, \ldots, x_m \mapsto t_m])))$.

Each play begins at the initial non-terminal S, and continues either ad infinitum or until a term $a_1\ (a_2\ (\cdots\ (a_k\ \$)))$, understood as the word $w = a_1 \ldots a_k$, is produced. Infinite plays do not produce a word and are won by Player $\Diamond$. Finite maximal plays produce such a word w. Player $\Diamond$ wins whenever $w \in \mathcal{L}(A)$, Player $\Box$ wins if $w \in \overline{\mathcal{L}(A)}$. Since the winning condition is Borel, either Player $\Diamond$ or Player $\Box$ has a winning strategy [43].

The Winner of a Higher-Order Game (HOG)	
Input:	A higher-order game $\mathcal{G}$.
Question:	Does Player $\Diamond$ win $\mathcal{G}$? If so, effectively represent Player $\Diamond$'s strategy.

Our contribution is a fixed-point algorithm to decide HOG. We derive it in three steps. First, we develop a concrete model for higher-order games whose semantics captures the above winning condition. Second, we introduce a framework that for two models and a mapping between them guarantees that the mapping of the greatest fixed point with respect to the one model is the greatest fixed point with respect to the other model. Finally, we introduce an abstract model that uses a finite ground domain. The solution of HOG can be read off from the semantics in the abstract model, which in turn can be computed via Kleene iteration. Moreover, this semantics can be used to define Player $\Diamond$'s winning strategy. We instantiate the framework for the concrete and abstract model to prove the soundness of the algorithm.

Concrete Semantics

Consider a HOG instance $\mathcal{G} = (G, A, O)$. Let G^{det} be the determinized version of G. Our goal is to define a model $\mathcal{M}^C = (\mathcal{D}^C, \mathcal{I}^C)$ such that the semantics of G^{det} in this model allows us to decide HOG. Recall that we only have to define the ground domain. For composed kinds, we use the functional lifting discussed in Section 3.

Our idea is to associate to kind o the set of positive Boolean formulas where the atomic propositions are words in T^*. To be able to reuse the definition, we define formula domains in more generality as follows.

Domains of Boolean Formulas. Given a (potentially infinite) set P of atomic propositions, the *positive Boolean formulas* $\mathsf{PBool}(P)$ over P are defined to contain true, every p from P, and compositions of formulas via conjunction and disjunction. We work up to logical equivalence, which means we treat ϕ_1 and ϕ_2 as equal as long as they are logically equivalent.

Unfortunately, if the set P is infinite, $\mathsf{PBool}(P)$ is not a cppo, because the meet of a descending chain of formulas might not be a finite formula. The idea of our domain is to have conjunctions of infinitely many formulas. As is common in logic, we represent them as infinite sets. Therefore, we consider the set of all sets of (finite) positive Boolean formulas $\mathcal{P}(\mathsf{PBool}(T^*)) \setminus \{\emptyset\}$ factorized modulo logical equivalence, denoted $(\mathcal{P}(\mathsf{PBool}(T^*)) \setminus \{\emptyset\})/_{\Leftrightarrow}$. To be precise, the sets may be finite or infinite, but they must be non-empty.

To define the factorization, let an assignment to the atomic propositions be given by a subset of $P' \subseteq P$. The atomic proposition p is true if $p \in P'$. An assignment satisfies a Boolean formula, if the formula evaluates to true in that assignment. It satisfies a set of Boolean formulas, if it satisfies all elements. Given two sets of formulas Φ_1 and Φ_2, we write $\Phi_1 \Rightarrow \Phi_2$, if every assignment that satisfies Φ_1 also satisfies Φ_2. Two sets of formulas are equivalent, denoted $\Phi_1 \Leftrightarrow \Phi_2$, if $\Phi_1 \Rightarrow \Phi_2$ and $\Phi_2 \Rightarrow \Phi_1$ holds.

The ordering on these factorized sets is implication (which by transitivity is independent of the representative). The top element is the set $\{\mathsf{true}\}$, which is implied by every set. The conjunction of two sets is union. Note that it forms the meet in the partial order, and moreover note that meets over arbitrary sets exist, in particular the domain is a cppo. We will also need an operation of disjunction, which is defined by $\Phi_1 \vee \Phi_2 = \{\phi_1 \vee \phi_2 \mid \phi_1 \in \Phi_1, \phi_2 \in \Phi_2\}$. We will also use disjunctions of higher (but finite) arity where convenient. Note that the disjunction on finite formulas is guaranteed to result in a finite formula. Therefore, the above is well-defined.

In our case, the assignment $P' \subseteq T^*$ of interest is the language of the automaton A. Player $\Diamond$ will win the game iff the concrete semantics assigns a set of formulas to S that is satisfied by $\mathcal{L}(A)$.

The Concrete Domains and Interpretation of Terminals. From a ground domain, higher-order domains are defined as continuous functions as in Section 3. Thus we only need

$$\mathcal{D}^C(o) = (\mathcal{P}(\mathsf{PBool}(T^*)) \setminus \{\emptyset\})/_{\Leftrightarrow} \ .$$

The endmarker \$ yields the set of formulas $\{\varepsilon\}$, i.e. $\mathcal{I}^C(\$) = \{\varepsilon\}$. A terminal $a : o \to o$ prepends a to a given word w. That is $\mathcal{I}^C(a) = \mathsf{prepend}_a$, where $\mathsf{prepend}_a$ distributes over conjunction and disjunction:

$$\mathsf{prepend}_a(\phi) = \begin{cases} aw & \phi = w \ , \\ \mathsf{prepend}_a(\phi_1)\ op\ \mathsf{prepend}_a(\phi_2) & \phi = \phi_1\ op\ \phi_2 \text{ and } op \in \{\wedge, \vee\} \ , \\ \phi & \phi = \mathsf{true} \ . \end{cases}$$

We apply $\mathsf{prepend}_a$ to sets of formulas by applying it to every element. Finally, $\mathcal{I}^C(op_F)$ where op_F has arity ℓ is an ℓ-ary conjunction (resp. disjunction) if Player $\Box$ (resp. $\Diamond$) owns F.

For $\mathcal{M}^C = (\mathcal{D}^C, \mathcal{I}^C)$ to be a model, we need our interpretation of terminals to be $\sqcap$-continuous. This follows largely by the distributivity of our definitions.

▶ **Lemma 4.** *For all non-ground terminals s, $\mathcal{I}^C(s)$ is $\sqcap$-continuous.*

▶ **Example 5.** Consider the higher-order game defined by the scheme $S = H\ a\ \$ \mid b\ \$$ and $H = \lambda f.\lambda x.f\ (f\ x) \mid \lambda f.\lambda x.H\ (H\ f)\ x$. Assume S is owned by Player $\Diamond$ and H is owned by Player $\Box$. Let the automaton accept the language $\{b\}$. Player $\Diamond$ can choose to rewrite S to $b\ \$$ and therefore has a strategy to produce a word in the language. To derive this information from the concrete semantics, we compute $\sigma_{\mathcal{M}^C}(H)$. It is the function mapping $f \in \mathit{Cont}(\mathcal{D}^C(o), \mathcal{D}^C(o))$ and $d \in \mathcal{D}^C(o)$ to $\bigcup_{k>0} f^{2k}(d)$. Note that the union is the conjunction of sets of formulas, which is the interpretation of op_H for the universal player. Moreover, note that due to non-determinism we obtain all even numbers of applications of f, not only the powers of 2. With this, the semantics of the initial symbol is

$$\sigma_{\mathcal{M}^C}(S) = \bigcup_{k>0} \mathsf{prepend}_a^{2k}(\{\varepsilon\}) \vee \mathsf{prepend}_b(\{\varepsilon\}) = \{a^{2k} \vee b \mid k > 0\}.$$

The assignment $\{b\}$ given by the language of the NFA satisfies $\{a^{2k} \vee b \mid k > 0\}$. Indeed, since b evaluates to true, every formula in the set evaluates to true.

Correctness of Semantics and Winning Strategies. We need to show that the concrete semantics matches the original semantics of the game.

▶ **Theorem 6.** *$\sigma_{\mathcal{M}^C}(S)$ is satisfied by $\mathcal{L}(A)$ iff there is a winning strategy for Player $\Diamond$.*

When $\sigma_{\mathcal{M}^C}(S)$ is satisfied by $\mathcal{L}(A)$ the concrete semantics gives a winning strategy for $\Diamond$: From a term t such that $\mathcal{M}^C[\![t]\!]\ \sigma_{\mathcal{M}^C}$ is satisfied by $\mathcal{L}(A)$, Player $\Diamond$, when able to choose, picks a rewrite rule that transforms t to t', where $\mathcal{M}^C[\![t']\!]\ \sigma_{\mathcal{M}^C}$ remains satisfied. The proof of Theorem 6 shows this is always possible, and, moreover, Player $\Box$ is unable to reach a term for which satisfaction does not hold. This does not yet give an effective strategy since we cannot compute $\mathcal{M}^C[\![t]\!]\ \sigma_{\mathcal{M}^C}$. However, the abstract semantics will be computable, and can be used in place of the concrete semantics by Player $\Diamond$ to implement the winning strategy.

The proof that $\sigma_{\mathcal{M}^C}(S)$ being unsatisfied implies a winning strategy for Player $\Box$ is more involved and requires the definition of a correctness relation between semantics and terms that is lifted to the level of functions, and shown to hold inductively.

5 Framework for Exact Fixed-Point Transfer

The concrete model $\mathcal{M}^C$ does not lead to an algorithm for solving HOG since its domains are infinite. Here, we consider an abstract model $\mathcal{M}^A$ with finite domains. The soundness of the resulting Kleene iteration relies on the two semantics being related by a precise abstraction α. Since both semantics are defined by fixed points, this requires us to prove $\alpha(\sigma_{\mathcal{M}^C}) = \sigma_{\mathcal{M}^A}$. In this section, we provide a general framework to this end.

Consider the deterministic scheme G together with two models (left and right) $\mathcal{M}_l = (\mathcal{D}_l, \mathcal{I}_l)$ and $\mathcal{M}_r = (\mathcal{D}_r, \mathcal{I}_r)$. Our goal is to relate the semantics in these models in the sense that $\sigma_{\mathcal{M}_r} = \alpha(\sigma_{\mathcal{M}_l})$. Such exact fixed-point transfer results are well-known in abstract interpretation. To generalize them to higher-order we give easy to instantiate conditions on α, $\mathcal{M}_l$, and $\mathcal{M}_r$ that yield the above equality. Interestingly, exact fixed-point transfer results seem to be rare for higher-order (e.g. [46]). Our development is inspired by Abramsky's lifting of abstraction functions to logical relations [3], which generalizes [11, 4]. These works focus on approximation and the compatibility we need for exactness is missing. Our framework is easier to apply than [15, 6], which are again concerned with approximation and do not offer (but may lead to) exact fixed-point transfer results.

For the terminology, an *abstraction* is a function $\alpha : \mathcal{D}_l(o) \to \mathcal{D}_r(o)$. To lift the abstraction to function domains, we define the notion of *being compatible with* α. Compatibility intuitively states that the function on the concrete domain is not more precise than what the abstraction function distinguishes. This allows us to define the abstraction of a function by applying the function and abstracting the result, $\alpha(f)\ \alpha(v_l) = \alpha(f\ v_l)$. Compatibility ensures the independence of the choice of v_l.

By definition, all ground elements $v_l \in \mathcal{D}_l(o)$ are compatible with α. For function domains, compatibility and the abstraction are defined as follows.

▶ **Definition 7.** Assume α and the notion of compatibility are defined on $\mathcal{D}_l(\kappa_1)$ and $\mathcal{D}_l(\kappa_2)$. Let $\top^l_\kappa$ (resp. $\top^r_\kappa$) be the greatest element of $\mathcal{D}_l(\kappa)$ (resp. $\mathcal{D}_r(\kappa)$) for each κ.

1. Function $f \in \mathcal{D}_l(\kappa_1 \to \kappa_2)$ is compatible with α, if
   a. for all compatible $v_l, v'_l \in \mathcal{D}_l(\kappa_1)$ with $\alpha(v_l) = \alpha(v'_l)$ we have $\alpha(f\ v_l) = \alpha(f\ v'_l)$, and
   b. for all compatible $v_l \in \mathcal{D}_l(\kappa_1)$ we have that $f\ v_l$ is compatible.
2. We define $\alpha(f) \in \mathcal{D}_r(\kappa_1 \to \kappa_2)$ as follows.
   a. If f is compatible, we set $\alpha(f)\ v_r = \alpha(f\ v_l)$, provided there is a compatible $v_l \in \mathcal{D}_l(\kappa_1)$ with $v_r = \alpha(v_l)$, and $\alpha(f)\ v_r = \top^r_{\kappa_2}$ otherwise.
   b. If f is not compatible, $\alpha(f) = \top^r_{\kappa_1 \to \kappa_2}$.

We lift α to valuations $\nu : N \uplus V \nrightarrow \mathcal{D}_l$ by $\alpha(\nu)(F) = \alpha(\nu(F))$ and similar for x. We also lift compatibility to valuations $\nu : N \uplus V \nrightarrow \mathcal{D}_l$ by requiring $\nu(F)$ to be compatible for all $F \in N$ and similar for $x \in V$.

The conditions needed for the exact fixed-point transfer are the following.

▶ **Definition 8.** Function α is *precise* for $\mathcal{M}_l$ and $\mathcal{M}_r$, if

(P1) $\alpha(\mathcal{D}_l(o)) = \mathcal{D}_r(o)$,

(P2) $\alpha : \mathcal{D}_l(o) \to \mathcal{D}_r(o)$ is $\sqcap$-continuous,

(P3) $\alpha(\top^l_o) = \top^r_o$,

(P4) $\alpha(\mathcal{I}_l(s)) = \mathcal{I}_r(s)$ for all terminals $s\colon o$, and similarly $\alpha(\mathcal{I}_l(s)\ v_l) = \mathcal{I}_r(s)\ \alpha(v_l)$ for all terminals $s : \kappa_1 \to \kappa_2$ and all compatible $v_l \in \mathcal{D}_l(\kappa_1)$,

(P5) $\mathcal{I}_l(s)\ v_l$ is compatible for all terminals $s : \kappa_1 \to \kappa_2$, and all compatible $v_l \in \mathcal{D}_l(\kappa_1)$.

(P1) is surjectivity of α. (P2) states that α is well-behaved wrt. $\sqcap$. (P3) says that the greatest element is mapped as expected. Note that (P1)-(P3) are only posed for the ground domain. One can prove that they generalize to function domains by the definition of function abstraction. (P4) is that the interpretations of terminals in $\mathcal{M}^C$ and $\mathcal{M}^A$ are suitably related. Finally (P5) is compatibility. (P4) and (P5) are generalized to terms in Lemma 9.

To prove $\alpha(\sigma_{\mathcal{M}_l}) = \sigma_{\mathcal{M}_r}$, we need that $rhs_{\mathcal{M}_r}$ is an exact abstract transformer of $rhs_{\mathcal{M}_l}$. The following lemma states this for all terms t, in particular those that occur in the equations. The generalization to product domains is immediate. Note that the result is limited to compatible valuations, but this will be sufficient for our purposes. The proof proceeds by induction on the structure of terms, while simultaneously proving $\mathcal{M}_l[\![t]\!]$ compatible with α. With this result, we obtain the required exact fixed-point transfer for precise abstractions.

▶ **Lemma 9.** *Assume* (P1)*,* (P4)*, and* (P5) *hold. For all terms t and all compatible ν, we have $\mathcal{M}_l[\![t]\!]\ \nu$ compatible and $\alpha(\mathcal{M}_l[\![t]\!]\ \nu) = \mathcal{M}_r[\![t]\!]\ \alpha(\nu)$.*

▶ **Theorem 10** (Exact Fixed-Point Transfer). *Let G be a scheme with models $\mathcal{M}_l$ and $\mathcal{M}_r$. Let σ_l and σ_r be the corresponding semantics. If $\alpha : \mathcal{D}_l \to \mathcal{D}_r$ is precise, we have $\sigma_r = \alpha(\sigma_l)$.*

6 Domains for Higher-Order Games

We propose two domains, *abstract* and *optimized*, that allow us to solve HOG. The computation is a standard fixed-point iteration, and, in the optimized domain, this iteration has optimal complexity. Correctness follows by instantiating the previous framework.

Abstract Semantics. Our goal is to define an abstract model for games that (1) suitably relates to the concrete model from Section 4 and (2) is computable. By a suitable relation, we mean the two models should relate via an abstraction function. Provided the conditions on precision hold, correctness of the abstraction then follows from Theorem 10. Combined with Theorem 6, this will allow us to solve HOG. Computable in particular means the domain should be finite and the operations should be efficiently computable.

We define the $\mathcal{M}^A = (\mathcal{D}^A, \mathcal{I}^A)$ as follows. Again, we resolve the non-determinism into Boolean formulas. But rather than tracking the precise words generated by the scheme, we only track the current set of states of the automaton. To achieve the surjectivity required by precision, we restrict the powerset to those sets of states from which a word is accepted. Let $\mathsf{acc}(w) = \{q \mid q \xrightarrow{w} q_f \in Q_f\}$. For a language L we have $\mathsf{acc}(L) = \{\mathsf{acc}(w) \mid w \in L\}$. The abstract domain for terms of ground kind is $\mathcal{D}^A(o) = \mathsf{PBool}(\mathsf{acc}(T^*))$. The lifting to functions is as explained in Section 3. Satisfaction is now defined relative to a set Ω of elements of $\mathcal{P}(Q_{NFA})$ (cf. Section 4). With finitely many atomic propositions, there are only finitely many formulas (up to logical equivalence). This means we no longer need sets of formulas to represent infinite conjunctions, but can work with plain formulas. The ordering is thus the ordinary implication with the meet being conjunction and top being true.

The interpretation of ground terms is $\mathcal{I}^A(\$) = Q_f$ and $\mathcal{I}^A(a) = \mathsf{pre}_a$. Here pre_a is the predecessor computation under label a, $\mathsf{pre}_a(Q) = \{q' \in Q_{NFA} \mid q' \xrightarrow{a} q \in Q\}$. It is lifted to formulas by distributing it over conjunction and disjunction. The composition operators are again interpreted as conjunctions and disjunctions, depending on the owner of the non-terminal. Since we restrict the atomic propositions to $\mathsf{acc}(T^*)$, we have to show that the interpretations use only this restricted set. Proving $\mathcal{I}^A(s)$ is $\sqcap$-continuous is standard.

▶ **Lemma 11.** *The interpretations are defined on the abstract domain.*

▶ **Lemma 12.** *For all terminals s, $\mathcal{I}^A(s)$ is $\sqcap$-continuous over the respective lattices.*

Recall our concrete model is $\mathcal{M}^C = (\mathcal{D}^C, \mathcal{I}^C)$, where $\mathcal{D}^C = \mathcal{P}(\mathsf{PBool}(T^*))$. To relate this model to $\mathcal{M}^A$, we define the abstraction function $\alpha : \mathcal{D}^C(o) \to \mathcal{D}^A(o)$. It leaves the Boolean structure of a formula unchanged but maps every word (which is an atomic proposition) to the set of states from which this word is accepted. For a set of formulas, we take the conjunction of the abstraction of the elements. This conjunction is finite as we work over a finite domain, so there is no need to worry about infinite syntax. Technically, we define α on $\mathsf{PBool}(T^*)$ by $\alpha(\Phi) = \bigwedge_{\phi \in \Phi} \alpha(\phi)$ for a set of formulas $\Phi \in \mathcal{P}(\mathsf{PBool}(T^*))$, and

$$\alpha(\phi) = \begin{cases} \mathsf{acc}(w) & \text{if } \phi = w, \\ \alpha(\phi_1)\, op\, \alpha(\phi_2) & \text{if } \phi = \phi_1\, op\, \phi_2 \text{ and } op \in \{\wedge, \vee\}, \\ \phi & \text{if } \phi = \mathsf{true}\ . \end{cases}$$

This definition is suitable in that $\alpha(\sigma_{\mathcal{M}^C}) = \sigma_{\mathcal{M}^A}$ entails the following.

▶ **Theorem 13.** *$\sigma_{\mathcal{M}^A}(S)$ is satisfied by $\{Q \in \mathsf{acc}(T^*) \mid q_0 \in Q\}$ iff Player $\Diamond$ wins $\mathcal{G}$.*

To see that the theorem is a consequence of the exact fixed-point transfer, observe that $\{Q \in \mathsf{acc}(T^*) \mid q_0 \in Q\} = \mathsf{acc}(\mathcal{L}(A))$. Then, by $\sigma_{\mathcal{M}^A} = \alpha(\sigma_{\mathcal{M}^C})$ we have $\mathsf{acc}(\mathcal{L}(A))$ satisfies

$\sigma_{\mathcal{M}^A}(S)$ iff it also satisfies $\alpha(\sigma_{\mathcal{M}^C}(S))$. This holds iff $\mathcal{L}(A)$ satisfies $\sigma_{\mathcal{M}^C}(S)$ (a simple induction over formulas). By Theorem 6, this occurs iff Player $\Diamond$ wins the game.

It remains to establish $\alpha(\sigma_{\mathcal{M}^C}) = \sigma_{\mathcal{M}^A}$. With the framework, the exact fixed-point transfer follows from precision, Theorem 10. The proof of the following is routine.

▶ **Proposition 14.** *α is precise. Hence, $\alpha(\sigma_{\mathcal{M}^C}) = \sigma_{\mathcal{M}^A}$.*

Optimized Semantics. The above model yields a decision procedure for HOG via Kleene iteration. Unfortunately, the complexity is one exponential too high: The height of the domain for a symbol of order k in the abstract model is $(k+2)$-times exponential, where the height is the length of the longest strictly descending chain in the domain. This gives the maximum number of steps of Kleene iteration needed to reach the fixed point.

We present an optimized version of our model that is able to close the gap: In this model, the domain for an order-k symbol is only $(k+1)$-times exponentially high. The idea is to resolve the atomic propositions in $\mathcal{M}^A$, which are sets of states, into disjunctions among the states. The reader familiar with inclusion algorithms will find this decomposition surprising.

We first define $\alpha : \mathsf{PBool}(\mathsf{acc}(T^*)) \to \mathsf{PBool}(Q_{NFA})$. The optimized domain will then be based on the image of α. This guarantees surjectivity. For a set of states Q, we define $\alpha(Q) = \bigvee Q = \bigvee_{q\in Q} q$. For a formula, the abstraction function is defined to distribute over conjunction and disjunction. The optimized model is $\mathcal{M}^O = (\mathcal{D}^O, \mathcal{I}^O)$ with ground domain $\alpha(\mathsf{PBool}(\mathsf{acc}(T^*)))$. The interpretation is $\mathcal{I}^O(\$) = \bigvee Q_f$. For a, we resolve the set of predecessors into a disjunction, $\mathcal{I}^O(a)\ q = \bigvee \mathsf{pre}_a(\{q\})$. The function distributes over conjunction and disjunction. Finally, $\mathcal{I}^O(op_F)$ is conjunction or disjunction of formulas, depending on the owner of the non-terminal. Since we use a restricted domain, we have to argue that the operations do not leave the domain. It is also straightforward to prove our interpretation is $\sqcap$-continuous as required.

▶ **Lemma 15.** *The interpretations are defined on the optimized domain.*

▶ **Lemma 16.** *For all terminals s, $\mathcal{I}^O(s)$ is $\sqcap$-continuous over the respective lattices.*

We again show precision, enabling the required exact fixed-point transfer.

▶ **Proposition 17.** *α is precise. Hence, $\alpha(\sigma_{\mathcal{M}^A}) = \sigma_{\mathcal{M}^O}$.*

▶ **Theorem 18.** *$\sigma_{\mathcal{M}^O}(S)$ is satisfied by $\{q_0\}$ iff Player $\Diamond$ wins $\mathcal{G}$.*

It is sufficient to show $\sigma_{\mathcal{M}^A}(S)$ is satisfied by $\{Q \in \mathsf{acc}(T^*) \mid q_0 \in Q\}$ iff $\sigma_{\mathcal{M}^O}(S)$ is satisfied by $\{q_0\}$. Theorem 13 then yields the statement. Propositions Q in $\sigma_{\mathcal{M}^A}(S)$ are resolved into disjunctions $\bigvee Q$ in $\sigma_{\mathcal{M}^O}(S)$. For such a proposition, we have $Q \in \{Q \in \mathsf{acc}(T^*) \mid q_0 \in Q\}$ iff $\bigvee Q$ is satisfied by $\{q_0\}$. This equivalence propagates to the formulas $\sigma_{\mathcal{M}^A}(S)$ and $\sigma_{\mathcal{M}^O}(S)$ as the Boolean structure coincides. The latter follows from $\alpha(\sigma_{\mathcal{M}^A}(S)) = \sigma_{\mathcal{M}^O}(S)$.

Complexity. To solve HOG, we compute the semantics $\sigma_{\mathcal{M}^O}$ and then evaluate $\sigma_{\mathcal{M}^O}(S)$ at the assignment $\{q_0\}$. For the complexity, assume that the highest order of any non-terminal in $\mathcal{G}$ is k. We show the number of iterations needed to compute the greatest fixed point is at most $(k+1)$-times exponential. We do this via a suitable upper bound on the length of strictly descending chains in the domains assigned by $\mathcal{D}^O$.

▶ **Proposition 19.** *The semantics $\sigma_{\mathcal{M}^O}$ can be computed in $(k+1)$*EXP*, where k is the highest order of any non-terminal in the input scheme.*

The lower bound is via a reduction from the word membership problem for alternating k-iterated pushdown automata with polynomially-bounded auxiliary work-tape. This problem was shown by Engelfriet to be $(k+1)$EXP-hard. We can reduce this problem to HOG via well-known translations between iterated stack automata and recursion schemes, using the regular language specifying the winning condition to help simulate the work-tape.

▶ **Proposition 20.** *Determining whether Player* $\Diamond$ *wins* $\mathcal{G}$ *is* $(k+1)$EXP*-hard for* $k > 0$*.*

Together, these results show the following corollary and final result.

▶ **Corollary 21.** HOG *is* $(k+1)$EXP*-complete for order-*k *schemes and* $k > 0$*.*

References

1 P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In *CAV*, volume 6174 of *LNCS*, pages 132–147. Springer, 2010.

2 P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. Advanced Ramsey-based Büchi automata inclusion testing. In *CONCUR*, volume 6901 of *LNCS*, pages 187–202. Springer, 2011.

3 S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *J. Log. Comp.*, 1(1):5–40, 1990.

4 S. Abramsky and C. Hankin. An introduction to abstract interpretation. In *Abstract Interpretation of declarative languages*, volume 1, pages 63–102. Ellis Horwood, 1987.

5 K. Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. *LMCS*, 3(3):1–23, 2007.

6 K. Backhouse and R. C. Backhouse. Safety of abstract interpretations for free, via logical relations and Galois connections. *Sci. Comp. Prog.*, 51(1-2):153–196, 2004.

7 A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997.

8 A. Bouajjani and A. Meyer. Symbolic reachability analysis of higher-order context-free processes. In *FSTTCS*, volume 3328 of *LNCS*, pages 135–147. Springer, 2004.

9 C. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *ICALP*, volume 7392 of *LNCS*, pages 165–176. Springer, 2012.

10 C. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *CSL*, volume 23 of *LIPIcs*, pages 129–148. Dagstuhl, 2013.

11 G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Sci. Comp. Prog.*, 7(3):249–278, 1986.

12 T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *ICALP*, volume 2380 of *LNCS*, pages 704–715. Springer, 2002.

13 T. Cachat. Higher order pushdown automata, the Caucal hierarchy of graphs and parity games. In *ICALP*, volume 2719 of *LNCS*, pages 556–569. Springer, 2003.

14 D. Caucal. On infinite terms having a decidable monadic theory. In *MFCS*, volume 2420 of *LNCS*, pages 165–176. Springer, 2002.

15 P. Cousot and R. Cousot. Higher order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection, and PER analysis. In *ICCL*, pages 95–112. IEEE, 1994.

16 W. Damm. The IO- and OI-hierarchies. *Theor. Comp. Sci.*, 20:95–207, 1982.

17 W. Damm and A. Goerdt. An automata-theoretical characterization of the OI-hierarchy. *Inf. Comp.*, 71:1–32, 1986.

18 A. Farzan, Z. Kincaid, and A. Podelski. Proof spaces for unbounded parallelism. In *POPL*, pages 407–420. ACM, 2015.

19 A. Farzan, Z. Kincaid, and A. Podelski. Proving liveness of parameterized programs. In *LICS*, pages 185–196. IEEE, 2016.

20 Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proofs that count. In *POPL*, pages 151–164. ACM, 2014.

21 A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *ENTCS*, 9:27–37, 1997.

22 S. Fogarty and M. Y. Vardi. Efficient Büchi universality checking. In *TACAS*, volume 6015 of *LNCS*, pages 205–220. Springer, 2010.

23 C. Grellois. *Semantics of linear logic and higher-order model-checking.* PhD thesis, Université Paris Diderot (Paris 7), 2016.

24 C. Grellois and P.-A. Melliès. Finitary semantics of linear logic and higher-order model-checking. In *MFCS*, volume 9234 of *LNCS*, pages 256–268. Springer, 2015.

25 C. Grellois and P.-A. Melliès. An infinitary model of linear logic. In *FoSSaCS*, volume 9034 of *LNCS*, pages 41–55. Springer, 2015.

26 C. Grellois and P.-A. Melliès. Relational semantics of linear logic and higher-order model checking. In *CSL*, volume 41 of *LIPIcs*, pages 260–276. Dagstuhl, 2015.

27 A. Haddad. IO vs OI in higher-order recursion schemes. In *FICS*, volume 77 of *EPTCS*, pages 23–30, 2012.

28 M. Hague, R. Meyer, and S. Muskalla. Domains for higher-order games. *CoRR*, abs/1705.00355, 2017. URL: `http://arxiv.org/abs/1705.00355`.

29 M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, pages 452–461. IEEE, 2008.

30 M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. In *FoSSaCS*, volume 4423 of *LNCS*, pages 213–227. Springer, 2007.

31 M. Hague and C.-H. L. Ong. Winning regions of pushdown parity games: A saturation method. In *CONCUR*, volume 5710 of *LNCS*, pages 384–398. Springer, 2009.

32 M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, pages 471–482. ACM, 2010.

33 M. Hofmann and W. Chen. Abstract interpretation from Büchi automata. In *CSL-LICS*, pages 51:1–51:10, 2014.

34 M. Hofmann and J. Ledent. A cartesian-closed category for higher-order model checking. In *LICS*. IEEE, 2017. To appear.

35 L. Holík, R. Meyer, and S. Muskalla. Summaries for context-free games. In *FSTTCS*, volume 65 of *LIPIcs*, pages 41:1–41:16. Dagstuhl, 2016.

36 Lukás Holík and Roland Meyer. Antichains for the verification of recursive programs. In *NETYS*, volume 9466 of *LNCS*, pages 322–336. Springer, 2015.

37 T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS*, volume 2303 of *LNCS*, pages 205–222. Springer, 2002.

38 T. Knapik, D. Niwiński, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP*, volume 3580 of *LNCS*, pages 1450–1461. Springer, 2005.

39 N. Kobayashi. HorSat2: A model checker for HORS based on SATuration. A tool available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/horsat2/`.

40 N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, pages 416–428. ACM, 2009.

41 N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188. IEEE, 2009.

42 Z. Long, G. Calin, R. Majumdar, and R. Meyer. Language-theoretic abstraction refinement. In *FASE*, volume 7212 of *LNCS*, pages 362–376. Springer, 2012.

43 D. A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975. URL: `http://www.jstor.org/stable/1971035`.

44 R. Meyer, S. Muskalla, and E. Neumann. Liveness verification and synthesis: New algorithms for recursive programs. `https://arxiv.org/abs/1701.02947`.

45 C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90. IEEE, 2006.

46 S. J. Ramsay. *Intersection-Types and Higher-Order Model Checking*. PhD thesis, Oxford University, 2013.

47 S. J. Ramsay. Exact intersection type abstractions for safety checking of recursion schemes. In *PPDP*, pages 175–186. ACM, 2014.

48 S. Salvati. Recognizability in the simply typed lambda-calculus. In *WoLLIC*, volume 5514 of *LNCS*, pages 48–60. Springer, 2009.

49 S. Salvati and I. Walukiewicz. A model for behavioural properties of higher-order programs. In *CSL*, volume 41 of *LIPIcs*, pages 229–243. Dagstuhl, 2015.

50 S. Salvati and I. Walukiewicz. Typing weak MSOL properties. In *FoSSaCS*, volume 9034 of *LNCS*, pages 343–357. Springer, 2015.

51 S. Salvati and I. Walukiewicz. Using models to model-check recursive schemes. *LMCS*, 11(2):1–23, 2015.

52 I. Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comp.*, 164(2):234–263, 2001.

53 M. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.