Higher-Order MSL Horn Constraints

JEROME JOCHEMS, University of Bristol, UK
EDDIE JONES, University of Bristol, UK
STEVEN RAMSAY, University of Bristol, UK

The monadic shallow linear (MSL) class is a decidable fragment of first-order Horn clauses that was discovered and rediscovered around the turn of the century, with applications in static analysis and verification. We propose a new class of higher-order Horn constraints which extend MSL to higher-order logic and develop a resolution-based decision procedure. Higher-order MSL Horn constraints can quite naturally capture the complex patterns of call and return that are possible in higher-order programs, which make them well suited to higher-order program verification. In fact, we show that the higher-order MSL satisfiability problem and the HORS model checking problem are interreducible, so that higher-order MSL can be seen as a constraint-based approach to higher-order model checking. Finally, we describe an implementation of our decision procedure and its application to verified socket programming.

CCS Concepts: • **Theory of computation → Functional constructs**; **Program verification**; *Logic and verification.*

Additional Key Words and Phrases: higher-order program verification, constraint-based program analysis

ACM Reference Format:
Jerome Jochems, Eddie Jones, and Steven Ramsay. 2023. Higher-Order MSL Horn Constraints. *Proc. ACM Program. Lang.* 7, POPL, Article 69 (January 2023), 31 pages. https://doi.org/10.1145/3571262

1 INTRODUCTION

Constraints of various kinds form the basis of many program analyses and type inference algorithms. Specifying an analysis as a combination of generating and resolving constraints is very appealing: as Aiken [1999] remarks in his overview paper, *constraints help to separate specification from implementation*, they can *yield natural specifications* and their often rich theory (typically independent from the problem at hand) can *enable sophisticated optimisations* that may not be apparent if stating the analysis algorithm directly.

To realise these advantages, we want classes of constraints that can naturally express important features of the problem domain, that draw upon a well-understood theory and yet have a decidable satisfiability problem.

In this work we propose a new class of constraints that are designed to capture the complex, higher-order behaviours of programs with first-class procedures. We develop a part of the theory of these constraints and situate them in relation to other higher-order program analyses. Finally, we obtain an efficient decision procedure and we describe an application of the constraints to automatic verification of socket programming in a functional programming language.

Authors' addresses: Jerome Jochems, Department of Computer Science, University of Bristol, Bristol, UK, jerome.jochems@bristol.ac.uk; Eddie Jones, Department of Computer Science, University of Bristol, Bristol, UK, ej16147@bristol.ac.uk; Steven Ramsay, Department of Computer Science, University of Bristol, Bristol, UK, steven.ramsay@bristol.ac.uk.

2475-1421/2023/1-ART69
https://doi.org/10.1145/3571262

1.1 MSL Horn Constraints

Our constraints can be framed as an extension of the well-known *Monadic Shallow Linear* (MSL) Horn constraints to higher-order logic. Like many natural ideas, MSL constraints were discovered independently in two different communities. At CADE'99, Weidenbach [1999] proposed MSL Horn constraints as a natural setting in which to "combine the benefits of the finite state analysis and the inductive method". He showed that satisfiability was decidable and described how this class of constraints could be used in an automatic analysis of security protocols. Independently, at SAS'02, Nielson et al. [2002] proposed the $\mathcal{H}1$ class of Horn constraints, and it was later observed by [Goubault-Larrecq 2005] to be an equivalent reformulation of MSL. The $\mathcal{H}1$ class was originally used for the control flow analysis of the Spi language, but has also been applied to e.g. the verification of cryptosystems written in C.

MSL is the fragment of first-order Horn clauses obtained by restricting predicates to be monadic and restricting the subject of positive literals to be shallow and linear – that is, the single argument of a predicate in the head of a clause must be either a variable x or a function symbol applied to distinct variables: $f(x_1, \ldots, x_n)$. In practice, because we can view the function symbol f as constructing a tuple $(x_1, \ldots, x_n)$, it is convenient to also allow non-monadic predicates, so long as they are only applied to variables when used positively: $P(x_1, \ldots, x_n)$. With this concession, all of the following are MSL Horn clauses (we omit the prefix of universal quantifiers in each case):

$$\mathsf{Zero}(x) \Rightarrow \mathsf{Leq}(x, z) \qquad \mathsf{Leq}(\mathsf{s}(x), \mathsf{s}(y)) \Rightarrow \mathsf{Leq}(x, y) \qquad \mathsf{Leq}(x, y) \wedge \mathsf{Leq}(y, z) \Rightarrow \mathsf{Leq}(x, z)$$

$$\mathsf{Black}(\mathsf{leaf}) \qquad \mathsf{Black}(\mathsf{branch}(x, d, z)) \qquad \mathsf{Black}(x) \wedge \mathsf{Black}(z) \Rightarrow \mathsf{Red}(\mathsf{branch}(x, d, z))$$

$$\mathsf{M}(\mathsf{sent}(y, \mathsf{b}, \mathsf{pr}(\mathsf{encr}(\mathsf{tr}(y, x, \mathsf{tb}(z)), \mathsf{bt}), \mathsf{encr}(\mathsf{nb}(z), x)))) \wedge \mathsf{Sb}(\mathsf{pr}(y, z)) \Rightarrow \mathsf{Bk}(\mathsf{key}(x, y))$$

Note: there are no syntactical restrictions on the body of clauses. As can be seen in the last example, which is taken from Weidenbach's security protocol analysis, atoms in the body may contain terms with arbitrary nesting.

Sets of MSL clauses were shown by Weidenbach to have decidable satisfiability. The procedure is an instance of ordered resolution, with a carefully crafted ordering that guarantees terminating saturation. Essentially the same procedure was rediscovered independently by Goubault-Larrecq as he sought to construct a more standard procedure than Nielsen, Nielsen and Seidl's original, which was a bespoke kind of constraint normalisation.

In each case, the authors identify a key, solved form for constraints. Clauses in this solved form have shape: $Q_1(y_1) \wedge \cdots \wedge Q_k(y_k) \Rightarrow P(f(x_1, \ldots, x_m))$ with $\{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_m\}$. A set of clauses of this form can straightforwardly be viewed as an alternating tree automaton and so such clauses are called *automaton clauses*. Given as input a set of MSL Horn constraints C, each of the above decision procedures can be viewed as constructing a set of automaton clauses $\mathcal{A}$ equi-satisfiable with C; and since $\mathcal{A}$ is essentially a tree automaton, its satisfiability can be effectively determined.

1.2 Contributions of this Paper

In this work, we propose three different higher-order extensions of MSL constraints: (i) the class HOMSL(1) obtained by allowing predicates of higher types but maintaining the limitation of first-order function symbols, (ii) the class MSL(ω) obtained by allowing function symbols of higher types but maintaining the limitation of first-order (monadic) predicates and (iii) the class HOMSL(ω) obtained by allowing both predicates and function symbols of higher type.

I. Reduction to Existential-Free MSL(ω). Our first contribution is to show that the satisfiability problem for all of the above classes reduces to the satisfiability problem for a fragment of MSL(ω)

– i.e. first-order predicates with higher-order function symbols – in which formulas contain no existential quantification. We show that existential quantifiers are, in a sense, already definable using higher-order predicates and that higher-order predicates in general can be represented as higher-order functions, whose truth is internalised as a new first-order predicate.

II. Decidability of MSL(ω). Our second contribution is to give a resolution-based algorithm for deciding the satisfiability problem of MSL(ω). A key difficulty in generalising the resolution-based decision procedure for the first-order fragment is handling negative literals whose subject is headed by a variable: $P(x\ s_1\ \cdots s_n)$. Literals of this form simply cannot occur in the first-order case, and allowing (higher-order) resolution on such literals creates clauses that violate one of the cornerstones of the decidability result at first order, namely that clause heads are shallow.

We introduce a novel kind of higher-order resolution that avoids this phenomenon, but it necessitates a significantly different notion of automaton formula (i.e. solved form), which nevertheless specialises to the existing definition at first order. A simple type system for automaton clauses ensures that the level of nesting and the binding structure of variables is in tight correspondence with the type-theoretic order of the function symbols involved. Consequently, as in the first-order case, there can be only finitely many automaton clauses associated with a given problem instance, and this forms the backbone of the decidability proof.

III. Interreducibility of MSL(ω), HORS Model Checking, and Intersection (Refinement) Typability. Although they look superficially complex, it is easy to see that our higher-order automaton clauses, viewed as constraints, are in 1-1 correspondence with a simple kind of intersection types used in higher-order model checking. Since it is known that this class of intersection types define regular tree languages [Broadbent and Kobayashi 2013], the name *automaton clauses* is still justified. Our third contribution is to use this correspondence to moreover give two problem reductions: (i) from MSL(ω) satisfiability to HORS model checking and (ii) from intersection typeability to MSL(ω) satisfiability. The reduction from HORS model checking to intersection typeability is already well known [Kobayashi 2013], and thus completes the cycle. We obtain from these reductions that MSL(ω) satisfiability is $(n-1)$-EXPTIME hard for $n \geq 2$ (here n refers to the type-theoretic order of the function symbols). This is the class of problems that can be solved in time bounded by a tower of exponentials of height-n.

IV. Application to Verified Socket Programming. As proof of concept, we have implemented a prototype of our decision procedure in Haskell and applied it to the higher-order verification problem of socket usage in Haskell programs. Interestingly, our extraction of clauses from a given Haskell program does not analyse the source code directly. Instead, the domain-specific language is represented as a typeclass that can be instantiated with a specific "analysis" instance that extracts a representation of the program to be passed to our decision procedure in addition to the usual IO implementation. The principle advantage of this approach is that the source code need not be present, making way for library functions to appear freely. Furthermore, it is easier to implement and maintain, as source code need not be separately processed. As far as we are aware, this technique is novel and could be fruitfully applied to other domains.

1.3 Wider Significance

In first-order automated program verification there is a consensus around first-order logic as foundation, to the benefit of the field. On the one hand, ideas from first-order logic, such as interpolants, the Horn fragment, abduction, resolution, and so on have found an important place in automated verification. On the other, first-order logic provides a common vocabulary with which to understand the automated verification landscape *conceptually*, and locate different technologies.

However, higher-order automated program verification comprises a disparate collection of formalisms: refinement types [Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009; Vazou et al. 2015, 2013; Zhu and Jagannathan 2013], higher-order grammars and automata [Hague et al. 2008; Kobayashi 2013; Kobayashi and Ong 2009; Ramsay et al. 2014; Salvati and Walukiewicz 2016], higher-order fixpoint logic [Bruse et al. 2021; Kobayashi 2021; Viswanathan and Viswanathan 2004], and many others. Moreover, the procedures involved are often bespoke and difficult to relate to techniques that are standard in first-order verification.

This paper is part of a larger effort to establish an analogous foundation for higher-order automated program verification based on higher-order logic [Cathcart Burn et al. 2017, 2021; Jochems 2020; Ong and Wagner 2019]. We have shown that a standard technique from first-order automated reasoning, namely saturation under resolution, is effective at higher order, and even forms a decision procedure for the higher-order extension of MSL. Furthermore, in combination with our correspondence between intersection types and higher-order automaton clauses, this sets up a pattern for understanding type-based approaches to verification more generally.

Our interreducibility results allow us to situate higher-order model checking (also known as HORS model checking) conceptually, within the HOL landscape. This is beneficial because HORS model checking, although influential, is a set of techniques for solving a somewhat exotic problem. The safety version of the problem asks to decide if the value tree determined by a certain kind of higher-order grammar is accepted by a Büchi tree automaton with a trivial acceptance condition [Kobayashi 2013]. Thanks to the results of this paper, this form of higher-order model checking can be located more simply as "a group of decision procedures for the MSL fragment". We hope this will make this topic more accessible to the wider verification community, since monadic, shallow, and linear restrictions are well understood even by non-experts on higher-order program verification.

Our results also make an interesting connection with work on constructive logic and logic programming. Our higher-order automaton formulas are a special form of higher-order hereditary Harrop clauses (HOHH), lying in the intersection of HOHH goal and definite formulas. The fact that we have shown them to play an essential role in characterising satisfiability for this class of higher-order *Horn* clauses sheds a novel light on the relationship between these two classes of formulas, which have been instrumental in the work of Miller and his collaborators [Miller and Nadathur 2012; Miller et al. 1991]. Furthermore, our intersection type and higher-order automaton clause correspondence suggests an alternative to the "Horn Clauses as Types" interpretation pioneered by Fu, Komendantskaya, and coauthors [Farka 2020; Fu and Komendantskaya 2015, 2017; Fu et al. 2016]. Instead of identifying Horn clauses and types, resolution and proof term construction, our work casts types as monadic predicates, represented as HOHH clauses with a single free variable, and saturation-under-resolution as type inference.

1.4 Outline

The paper is structured as follows. In Section 2 we introduce higher-order MSL Horn constraints and their proof system, and we give an example of their use in verifying lazy IO computations. In Section 3 we reduce the provability problem for HOMSL(ω) to the same problem for existential-free MSL(ω). In Section 4 we introduce higher-order automaton clauses and use them in deciding satisfiability (via goal-formula entailment). In Section 5, we show that the MSL(ω) satisfiability and HORS model checking are interreducible. In Section 6 we describe our implementation and its application. Finally, in Section 7 we conclude with a description of related and future work. All proofs are available in the long version of this paper [Jochems et al. 2022].

$$\text{(Var)}\,\frac{}{\Delta \vdash x : \tau}\, x : \tau \in \Delta \qquad \text{(Cst)}\,\frac{}{\Delta \vdash c : \gamma}\, c : \gamma \in \Sigma \qquad \text{(Pred)}\,\frac{}{\Delta \vdash P : \rho}\, P : \rho \in \Pi$$

$$\text{(True)}\,\frac{}{\Delta \vdash \mathsf{true} : o} \qquad \text{(App)}\,\frac{\Delta \vdash s : \tau_1 \to \tau_2 \quad \Delta \vdash t : \tau_1}{\Delta \vdash s\,t : \tau_2} \qquad \text{(And)}\,\frac{\Delta \vdash G : o \quad \Delta \vdash H : o}{\Delta \vdash G \wedge H : o}$$

$$\text{(Ex)}\,\frac{\Delta,\, x : \sigma \vdash G : o}{\Delta \vdash \exists x{:}\sigma.\, G : o}\, x \notin \mathsf{dom}(\Delta) \qquad \text{(Cl)}\,\frac{\Delta,\, \overline{y{:}\sigma} \vdash G : o \quad \Delta,\, \overline{y{:}\sigma} \vdash A : o}{\Delta \vdash \forall \overline{y{:}\sigma}.\, G \Rightarrow A}$$

Fig. 1. Typing of terms, goals and clauses

2 HIGHER-ORDER MSL HORN FORMULAS

The logics we consider will make a type-level distinction between predicates and the subjects that they classify.

Definition 2.1 (Syntax of types). We consider a subset of simply typed applicative terms. Starting from the atomic *type of individuals* ι and the atomic *type of propositions* o, the types are given by:

$$\begin{array}{lrcl@{\qquad}lrcl} \text{(Constructor Types)} & \gamma & ::= & \iota \mid \kappa \to \gamma & \text{(Predicate Types)} & \rho & ::= & o \mid \sigma \to \rho \\ \text{(Constructor Arg Types)} & \kappa & ::= & \gamma & \text{(Predicate Arg Types)} & \sigma & ::= & \kappa \mid \rho \end{array}$$

We use τ as a metavariable for types in general (i.e. that may belong to any of the above classes). Thus, the constructor types are just the simple types built over a single base type ι, and the predicate types are those simple types with an o in tail position (i.e. that are ultimately constructing a proposition) and whose arguments are either other predicates or constructors. The introduction of the metavariable κ seems unmotivated, but we will later place restrictions on κ that differ from those on γ more generally. We define the *order of a type* as follows:

$$\mathsf{ord}(\iota) = \mathsf{ord}(o) = 0 \qquad \mathsf{ord}(\tau_1 \to \tau_2) = \max(\mathsf{ord}(\tau_1) + 1,\, \mathsf{ord}(\tau_2))$$

After introducing terms below, we will say that the *order of a typed term* is the order of its type.

Definition 2.2 (Terms, Clauses and Formulas). In all that follows, we assume a finite signature Σ of typed function symbols and a finite signature Π of typed predicate symbols. We use a, b, c and other lowercase letters from the beginning of the Roman alphabet to stand for function symbols and P, Q, R and so on to stand for predicates. Function symbols have types of shape γ and predicate symbols have type of shape ρ.

Terms. We assume a countably infinite set of variables. We use lowercase letters x, y, z and so on to stand for variables. A *type context*, typically Δ, is a finite, partial function from variables to types. Then *terms*, typically s, t, u, are given by the following grammar:

$$\text{(Term)} \quad s,\, t,\, u \quad ::= \quad x \mid c \mid P \mid s\,t$$

We will only consider those terms that are well typed according to the system specified in Figure 1, in which (Ex) instantiates one variable at a time for convenience.

Since we work in higher-order logic, the syntactic category of terms includes both objects that we think of as predicates (which have type ρ) and those that we think of as strictly the subjects of predicates (which have type γ). To make the discussion easier, we will typically refer to terms of the former type as predicates and terms of the latter type γ as trees, or tree constructors.

The *depth* of a symbol x, c or P is 0 and the depth of an application $s\,t$ is the maximum of the depth of s and the depth of t with 1 added.

$$\text{(T)}\,\frac{}{D \vdash \mathsf{true}} \qquad \text{(And)}\,\frac{D \vdash G \quad D \vdash H}{D \vdash G \wedge H} \qquad \text{(Ex)}\,\frac{D \vdash G[t/x]}{D \vdash \exists x.\, G} \qquad \text{(Res)}\,\frac{D \vdash G[\overline{s}/\overline{y}]}{D \vdash A[\overline{s}/\overline{y}]}\,(\forall \overline{y}.\, G \Rightarrow A) \in D$$

Fig. 2. Proof system for goal formulas

Formulas. The *atomic formulas*, typically A and B, are just those terms of type o. We define the *goal formulas* and *definite formulas* by mutual induction using the following grammar:

$$\begin{array}{lrcl} \text{(Goal Formula)} & G,\, H & ::= & A \mid G \wedge H \mid \exists x{:}\sigma.\, G \mid \mathsf{true} \\ \text{(Definite Formula)} & C,\, D & ::= & \forall \overline{y{:}\sigma}.\, G \Rightarrow P\,\overline{y} \mid \forall \overline{y{:}\sigma}.\, G \Rightarrow P\,(c\,\overline{y}) \mid C \wedge D \mid \mathsf{true} \end{array}$$

Wherever possible we will omit the explicit type annotation on binders and we write $\mathsf{FV}(X)$ to denote the typed free variables of some term, clause or formula X. We identify formulas up to renaming of bound variables.

The first two alternatives of the syntactic class of definite formulas are the two kinds of *definite clause* that we consider in this work. They differ in the shape of the head of the clause ($P\,\overline{y}$ vs $P\,(c\,\overline{y})$), but in both cases the variables $\overline{y}$ are required to be distinct from each other (i.e. the arguments are *shallow* and *linear*). This is how we express the MSL restriction in our higher-order setting.

We will often think of definite formulas rather as sets of definite *clauses* (the conjuncts of the formula), thus legitimising notation such as $(\forall \overline{y}.\, G \Rightarrow P\,\overline{y}) \in D$, and this is greatly smoothed by adopting the following conventions: we identify formulas up to the commutativity and associativity of conjunction and the use of true as unit.

In the following, we will only consider those goal formulas and clauses that are well typed according to the judgement defined inductively in Figure 1.

2.1 Proof System and Decision Problems

In this paper we will mostly work with respect to a certain proof system for judgements of the form $D \vdash G$, that is: from a given set of definite clauses D a given goal formula G follows.

Definition 2.3 (Proof System). The rules defining the system are given in Figure 2, with $[\overline{s}/\overline{y}]$ denoting the substitution of s_i for each y_i in $\overline{y}$, and $[t/x]$ of t for x. Substitution terms may be higher order and are assumed to be well typed. Note that the substitution in (Res) may be vacuous.

Most of the proof rules are standard but note that (Res) is given its name because it simulates (part of) the role of the resolution rule in clausal presentations of higher-order logic and we will sometimes refer to this rule as performing a "resolution step". We note that the mechanism by which a given definite clause and a corresponding atomic formula interact in the (Res) rule is by matching rather than unification, and the reason that this is possible is because we have an explicit existential introduction rule (Ex).

The above remarks are substantiated by the following result which, leveraging results on higher-order constrained Horn clauses due to Ong and Wagner [2019], shows that this system characterises truth for this fragment of higher-order logic under the standard semantics. The result requires that we assume (a) the predicate signature contains a universal relation Univ_ρ for each type ρ occurring in D or G, which is axiomatised by a corresponding universal clause $(\forall \overline{y}.\, \mathsf{true} \Rightarrow \mathsf{Univ}_\rho\,\overline{y})$ in D, and (b) that every tree constructor type γ that occurs in D or G is inhabited by some closed term. Clearly, both of these can be satisfied by suitable extensions of the signatures if they are not satisfied already. We shall call signatures that satisfy these requirements *adequate*.

Theorem 2.4. *Assuming adequate signatures, the proof system is sound and complete.*

For this reason, we will develop most of our results with respect to the proof system and implicitly adhere to the usual definition of the semantics of formulas.

As a corollary of the correspondence between the systems, we also obtain that higher-type existentials do not add any power to the system, and so can be ignored:

Corollary 2.5. *$D \vdash \exists x{:}\rho.\, G$ if, and only if, $D \vdash G[\mathsf{Univ}_\rho/x]$*

The proof system is very straightforward to use since, apart from choosing which clause to apply in a (Res) step, the rules are syntax directed according to the shape of the goal. An example of the use of the proof system is given following the next subsection, in 2.3.

As usual in higher-order logic, we write $D \models G$ to mean "G holds for every model of D".

Decision Problems. Although we have been discussing satisfiability in the introduction, a conjunction of Horn constraints $D \wedge (G \Rightarrow \mathsf{false})$ is satisfiable iff $D \not\models G$. Hence, we can equivalently consider the problem of deciding the entailment problem $D \models G$, and this is more natural in our setup. Moreover, due to completeness, we can equivalently consider the provability of $D \vdash G$. We state all kinds of problems because we will prefer one or the other in reductions according to the shape of yes-instances and the form of the input.

Definition 2.6 (Entailment, satisfiability, and provability problems). Given a definite formula D and a goal formula G over signatures Π and Σ, the *entailment problem* is to determine $D \models G$, the *satisfiability problem* is to determine $D \not\models G$ and the *provability problem* is to determine $D \vdash G$.

2.2 Stratification by Type-Theoretic Order

In the forgoing subsection we have given a very liberal account of what it means to extend MSL to higher types, but one can imagine at least two other possible definitions which are just as natural. First, we may consider a higher-order extension in which tree constructors are allowed to be of higher type, but predicates are not – in other words, as in first-order MSL, the subject of a predicate is a term of type ι, but now this term may be constructed internally using constants of higher type. Second, we may consider a higher-order extension in which predicates are allowed to be higher order, but tree constructors are not – in other words, as in first-order MSL, the terms of type ι are essentially first order, but now predicates may take other predicates as arguments.

These two fragments and the unrestricted logic of the previous subsection arise naturally from the following stratification according to type-theoretic order.

Definition 2.7 (Higher-order fragments). The family of fragments $\mathsf{HOMSL}(n)$, for n drawn from $\mathbb{N} \cup \{\omega\}$, denotes the restriction of the logic of the previous subsection to tree constructor argument types κ of order at most $n-1$. The family of fragments $\mathsf{MSL}(n)$ additionally restricts predicate argument types σ to ι only (i.e. all predicates are of type $\iota \to o$).

Here we regard $\omega - 1$ as ω and an index of 0 as a prohibition on arguments (i.e. one may not construct function types). Under this stratification, we can recognise the following fragments:

- $\mathsf{MSL}(0)$ is Datalog: predicates are first order and their subjects are (nullary) constants.
- $\mathsf{MSL}(1)$ is the first-order MSL fragment.
- $\mathsf{MSL}(\omega)$ is the second of the two higher-order extensions described above: we have first-order predicates whose subjects are trees constructed from an arbitrary higher-order signature.
- $\mathsf{HOMSL}(0)$ is higher-order Datalog, as studied by, e.g. Charalambidis et al. [2019].
- $\mathsf{HOMSL}(1)$ is the first of the two higher-order extensions described above: we have higher-order predicates over trees defined using only first-order tree constructors.
- $\mathsf{HOMSL}(\omega)$ is the full language described in the previous subsection.

The MSL(0) and MSL(1) fragments live within first-order Horn clauses and follow Miller and Nadathur [2012]'s presentation thereof (as *fohc*), while our larger fragments fall within higher-order Horn clauses (as *hohc*):

$$\begin{array}{ccccccc}
 & \textit{fohc} & & & & & \\
\mathrm{MSL}(0) & \subseteq & \mathrm{MSL}(1) & \subseteq & \cdots & \subseteq & \mathrm{MSL}(\omega) \\
\rotatebox{90}{$\supseteq$} & & \rotatebox{90}{$\supseteq$} & & & & \rotatebox{90}{$\supseteq$} \\
\mathrm{HOMSL}(0) & \subseteq & \mathrm{HOMSL}(1) & \subseteq & \cdots & \subseteq & \mathrm{HOMSL}(\omega) \\
 & & & & \textit{hohc} & &
\end{array}$$

2.3 Example: Constraints for Lazy IO

Our motivation is to use higher-order constraints to specify certain higher-order program verification problems, and especially the verification of safety properties for functional programs. We describe a general approach to using higher-order MSL constraints for verified socket programming in Section 6, but let us here consider a different example: verifying the correctness of a lazy IO computation. Consider the following Haskell expression, which is featured on the Haskell.org wiki as a prototypical example of a mistake due to improper use of lazy IO for any input [Haskell.org 2013]. The expression throws a runtime exception for attempting to read from a closed file handle.

```
1    do  contents ← withFile "test.txt"  ReadMode hGetContents
2        putStrLn  contents
```

This code reads the file named "test.txt" (line 1) and prints the contents to stdout (line 2).

The problem comes from the interaction between the lines. The reading of the file is done using the primitive hGetContents, which returns the list of characters read from a handle lazily[1]. The hGetContents action is wrapped in the withFile combinator, which brackets the execution of hGetContents between a call to open the handle and a call to close it again. Hence, at the point at which the contents of the file are demanded, in line 2, the file handle has already been closed as a result of leaving withFile, and forcing the lazy list of characters results in attempting to read from this closed handle.

An abstraction of the behaviour of this expression, and the primitives and combinators contained therein, can be expressed as a set of higher-order MSL clauses shown in Figure 3. A systematic approach to verifying lazy IO is not a contribution of this work, so it is not essential to understand the way in which these clauses model the situation. However, it is worth looking at the encoding in some detail, since we will use it as a kind of running example. Note that [] and : are the usual Haskell nullary and binary list constructors that denote the empty list and list composition, respectively.

The clauses effectively model a version of the above expression in which both global state (the status of the file handle) and control flow (lazy evaluation) are represented explicitly by threading a state parameter h and passing continuations k respectively. In addition to the various functions that appear in the source code, but which now expect an additional pair of arguments h and k, there are two constants o and c, representing the two possible states (open and closed – recall that we omit semi-closed for simplicity) of the handle, and four predicates V, Ex, Open and Closed.

The idea is that the predicate V (for 'V'iolation) is true of its argument s just if s represents an expression that will attempt to read from a closed file handle.

The goal V(withFile `"test.txt"` hGetContents c (act_2 id)) represents the verification problem: does the given expression crash with a closed file-handle violation? The idea of the representation is

[1]In fact the handle is put into an intermediate *semi-closed* state, but it is not important to this example so, in the interests of simplicity, we will not model it in what follows.

$$
\begin{aligned}
p\,[\,] &\Rightarrow \mathsf{Ex}\,p && (1) \\
\exists y.\ p(y{:}\mathsf{read}) &\Rightarrow \mathsf{Ex}\,p && (2) \\
\mathsf{V}(x) &\Rightarrow \mathsf{V}(\mathsf{id}\,x) && (3) \\
\mathsf{V}(k\,x\,h) &\Rightarrow \mathsf{Pred}\,h\,k\,x && (4) \\
\mathsf{V}(k\,\mathsf{o}) &\Rightarrow \mathsf{V}(\mathsf{open}\,h\,k) && (5) \\
\mathsf{V}(k\,\mathsf{c}) &\Rightarrow \mathsf{V}(\mathsf{close}\,h\,k) && (6)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Closed}(h) &\Rightarrow \mathsf{V}(\mathsf{read}\ h\ k) && (7) \\
\mathsf{Open}(h) \wedge \mathsf{Ex}\,(\mathsf{Pred}\,h\,k) &\Rightarrow \mathsf{V}(\mathsf{read}\ h\ k) && (8) \\
\mathsf{V}(k\ ()) &\Rightarrow \mathsf{V}(\mathsf{putCont}\ k\ y\ h) && (9) \\
\mathsf{V}(x\ h\ (\mathsf{putCont}\ k)) &\Rightarrow \mathsf{V}(\mathsf{putStrLn}\ x\ h\ k) && (10) \\
\mathsf{V}(\mathsf{open}\ h\ (\mathsf{withFile}_1\ f\ k)) &\Rightarrow \mathsf{V}(\mathsf{withFile}\,x\,m\,f\,h\,k) && (11) \\
\mathsf{V}(f\ h_0\ (\mathsf{withFile}_2\ k)) &\Rightarrow \mathsf{V}(\mathsf{withFile}_1\ f\ k\ h_0) && (12) \\
\mathsf{V}(\mathsf{close}\ h_1(\mathsf{withFile}_3\ y\ k)) &\Rightarrow \mathsf{V}(\mathsf{withFile}_2\ k\ h_1\ y) && (13) \\
\mathsf{V}(k\ y\ h_2) &\Rightarrow \mathsf{V}(\mathsf{withFile}_3\ y\ k\ h_2) && (14) \\
\mathsf{Open}(h) \wedge \mathsf{V}(k\ \mathsf{o}\ \mathsf{read}) &\Rightarrow \mathsf{V}(\mathsf{hGetContents}\ h\ k) && (15) \\
\mathsf{V}(\mathsf{putStrLn}\ x\ h_3\ \mathsf{id}) &\Rightarrow \mathsf{V}(\mathsf{act}_2\ x\ h_3) && (16) \\
\mathsf{true} &\Rightarrow \mathsf{Open}(\mathsf{o}) && (17) \\
\mathsf{true} &\Rightarrow \mathsf{Closed}(\mathsf{c}) && (18)
\end{aligned}
$$

Fig. 3. Clauses corresponding to the verification of Example 2.3

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Open}(\mathsf{o})}\,(17) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Closed}(\mathsf{c})}\,(18)}{\mathsf{V}(\mathsf{read}\ \mathsf{c}\ (\mathsf{putCont}\ \mathsf{id}))}\,(7)}{\mathsf{V}(\mathsf{putStrLn}\ \mathsf{read}\ \mathsf{c}\ \mathsf{id})}\,(10)}{\mathsf{V}(\mathsf{act}_2\ \mathsf{read}\ \mathsf{c})}\,(16)}{\mathsf{V}(\mathsf{withFile}_3\ \mathsf{read}\ \mathsf{act}_2\ \mathsf{c})}\,(14)}{\mathsf{V}(\mathsf{close}\ \mathsf{o}\ (\mathsf{withFile}_3\ \mathsf{read}\ \mathsf{act}_2))}\,(6)}{\mathsf{V}(\mathsf{withFile}_2\ \mathsf{act}_2\ \mathsf{o}\ \mathsf{read})}\,(13)}{\mathsf{Open}(\mathsf{o}) \wedge \mathsf{V}(\mathsf{withFile}_2\ \mathsf{act}_2\ \mathsf{o}\ \mathsf{read})}}{\mathsf{V}(\mathsf{hGetContents}\ \mathsf{o}\ (\mathsf{withFile}_2\ \mathsf{act}_2))}\,(15)}{\mathsf{V}(\mathsf{withFile}_1\ \mathsf{hGetContents}\ \mathsf{act}_2\ \mathsf{o})}\,(12)}{\mathsf{V}(\mathsf{open}\ \mathsf{c}\ (\mathsf{withFile}_1\ \mathsf{hGetContents}))}\,(5)}{\mathsf{V}(\mathsf{withFile}\ \text{“}\texttt{test.txt}\text{”}\ \mathsf{ReadMode}\ \mathsf{hGetContents}\ \mathsf{c}\ \mathsf{act}_2)}\,(11)
$$

Fig. 4. Proof in the environment given by Figure 3

as follows. Given that we think of every function as taking a file handle and a continuation, we can rephrase the expression as: withFile `"test.txt"` hGetContents c $(\lambda x\, h_3.\ \mathsf{putStrLn}\ x\ h_3\ (\lambda y.\, y))$ This captures via continuation passing style that evaluation must proceed by executing the expression withFile `"test.txt"` hGetContents in the initially closed handle state and with continuation $\lambda x\, h_3.\ \mathsf{putStrLn}\ x\ h_3\ k$. This continuation takes the suspended lazy stream x that is output by hGetContents and the state of the handle h_3 on exit from withFile, and attempts to print it to stdout before continuing with the remainder of the program, which just returns whichever value is output by putStrLn (which is just unit). However, since we don't allow for λ-abstractions in our constraints, we give a λ-lifted version of the above, with the innermost abstraction given instead by id and the outer one given by act_2.

$$\begin{aligned}
\mathrm{P}\ \mathrm{Q}\ x &\Rightarrow \mathrm{S}\ x \\
\mathrm{R}\ y \wedge x\ y &\Rightarrow \mathrm{P}\ x\ y \\
\mathrm{R}\ x &\Rightarrow \mathrm{Q}\ (\mathrm{a}\ x) \\
\mathrm{true} &\Rightarrow \mathrm{R}\ (\mathrm{a}\ x)
\end{aligned}
\qquad
\begin{aligned}
\mathrm{T}(\mathrm{p}\ \mathrm{q}\ x) &\Rightarrow \mathrm{T}(\mathrm{s}\ x) \\
\mathrm{T}(\mathrm{r}\ y) \wedge \mathrm{T}(x\ y) &\Rightarrow \mathrm{T}(\mathrm{p}\ x\ y) \\
\mathrm{Q}(z) &\Rightarrow \mathrm{T}(\mathrm{q}\ z) \\
\mathrm{R}(z) &\Rightarrow \mathrm{T}(\mathrm{r}\ z) \\
\mathrm{T}(\mathrm{r}\ x) &\Rightarrow \mathrm{Q}(\mathrm{a}\ x) \\
\mathrm{true} &\Rightarrow \mathrm{R}(\mathrm{a}\ x)
\end{aligned}
\qquad
\dfrac{\dfrac{\dfrac{\overline{\mathrm{R}(\mathrm{a}\ (\mathrm{a}\ \mathrm{c}))} \quad \dfrac{\dfrac{\overline{\mathrm{R}(\mathrm{a}\ \mathrm{c})}}{\mathrm{T}(\mathrm{r}\ (\mathrm{a}\ \mathrm{c}))}}{\mathrm{Q}(\mathrm{a}\ (\mathrm{a}\ \mathrm{c}))}}{\mathrm{T}(\mathrm{r}\ (\mathrm{a}\ (\mathrm{a}\ \mathrm{c}))) \wedge \mathrm{T}(\mathrm{q}\ (\mathrm{a}\ (\mathrm{a}\ \mathrm{c})))}}{\mathrm{T}(\mathrm{p}\ \mathrm{q}\ (\mathrm{a}\ (\mathrm{a}\ \mathrm{c})))}}{\mathrm{T}(\mathrm{s}\ (\mathrm{a}\ (\mathrm{a}\ \mathrm{c})))}$$

Fig. 5. Example of clauses (left) and their transform (center) and a proof (right)

Similarly, clauses (11)–(14) model the bracketing behaviour of withFile described above. An application of withFile to a filename x and an action on handle f will cause a violation (when started in a state in which the handle is h and the remaining program to compute is k), whenever open h $(\lambda h_0.\ f\ h_0\ (\lambda y\, h_1.\ \mathrm{close}\ h_1\ (\lambda h_2.\ k\ y\ h_2)))$ does. That is, calling open (with the same state h) to open the (implicit) file, then continuing by running the action f, then continuing by calling close and then finally continuing by executing the remainder of the program k (supplying the output y of the action f). The abstractions are lifted to, from left to right, $\mathrm{withFile}_1$, $\mathrm{withFile}_2$, and $\mathrm{withFile}_3$.

Clauses (1) and (2) constrain Ex to act like a specialised kind of (higher-order) existential quantifier. Ex takes a predicate p as input and holds whenever there is some list, of a certain form, that satisfies p. The form of the list models the thunking behaviour of the lazy stream resulting from hGetContents – in particular the fact that the tail of the list comprises another call to read.

A proof of V(withFile "test.txt" hGetContents c $(\mathrm{act}_2\ \mathrm{id})$), witnessing the fact that the expression does cause a violation, can be seen in Figure 4. Here, each use of (Res) is annotated by the number of the clause as given in Figure 3.

3 FROM HOMSL(ω) TO EXISTENTIAL-FREE MSL(ω)

The full HOMSL(ω) fragment is a remarkably expressive language with higher-order constructors, predicates, and existentials, allowing a wide range of higher-order verification problems to be expressed in a language that closely matches a functional source program.

In this section, we show that some of that power is illusory: existential quantification is definable using higher-order predicates (Theorem 3.2) and higher-order predicates are, in a sense, definable already using higher-order function symbols (Theorem 3.1). Hence, we are able to reduce the solvability problem from HOMSL(ω) to the solvability problem in existential-free MSL(ω). These reductions are extremely helpful for developing the rest of the results in the paper.

3.1 Elimination of Higher-Order Predicates

The idea of the first reduction is to simulate higher-order predicates using higher-order function symbols and a new, first-order "truth" predicate $\mathrm{T} : \iota \to o$. Consider the set of HOMSL(ω) clauses over predicates $\mathrm{P} : (\iota \to o) \to \iota \to o$, $\mathrm{Q} : \iota \to o$, $\mathrm{R} : \iota \to o$, and $\mathrm{S} : \iota \to o$, and function symbols $\mathrm{a} : \iota \to \iota$ and $\mathrm{c} : \iota$ that is shown on the left of Figure 5. The goal $\mathrm{S}(\mathrm{a}\ (\mathrm{a}\ \mathrm{c}))$ is provable from these clauses.

We will represent each of the predicates P, Q, R, and S by new function symbols $\mathrm{p} : (\iota \to \iota) \to \iota \to \iota$, $\mathrm{q} : \iota \to \iota$, $\mathrm{r} : \iota \to \iota$, and $\mathrm{s} : \iota \to \iota$ respectively. Since we have exchanged o everywhere in these types for ι, combinations that were possible involving P, Q, and R are still possible using their representatives, just with a different type. For example, P Q x in the body of the first clause can be represented as $\mathsf{p}\ \mathsf{q}\ x$, but note that this is a term of type ι so, in a sense, we have lost the notion of when the proposition is true.

To recover truth, we install a new predicate T and formulate the set of $\mathsf{MSL}(\omega)$ clauses shown in the center of Figure 5. Thus $\mathsf{T}(t)$ is true just if the proposition represented by the tree t is true (according to the representation scheme above). When t is a first-order predicate application, then its truth may depend on pattern matching in clause heads, and so truth is deferred to the original predicate (e.g. in the third and fourth clauses). For example, we can derive the goal $\mathsf{T}(\mathsf{s}\ (\mathsf{a}\ (\mathsf{a}\ \mathsf{c})))$, which encodes the higher-order goal $\mathsf{S}(\mathsf{a}\ (\mathsf{a}\ \mathsf{c}))$, as shown on the right of Figure 5.

Theorem 3.1. *Provability in HOMSL(ω) reduces to provability in MSL(ω).*

Thanks to Corollary 2.5, we assume WLOG that D and G contain no existentials of type ρ.

First, from a given $\mathsf{HOMSL}(\omega)$ constructor signature Σ and predicate signature Π, we construct $\mathsf{MSL}(\omega)$ signatures $\Sigma^\#$ and $\Pi^\#$. Let us write $D \vdash_\omega G$ to distinguish proof in the former and $D \vdash_1 G$ in the latter. Then we construct a section $(\!|-|\!)$ that maps formulas of the former into formulas of the latter in such a way that $D \vdash_\omega G$ iff $(\!|D|\!) \vdash_1 (\!|G|\!)$.

The MSL(ω) Signature. Let Π_1 be the subsignature consisting only of the first-order monadic predicates from Π. We start by transforming $\mathsf{HOMSL}(\omega)$ types τ to $\mathsf{MSL}(\omega)$ types $(\!|\tau|\!)$.

$$(\!|\iota|\!) := \iota \qquad (\!|o|\!) := \iota \qquad (\!|\sigma \to \rho|\!) := (\!|\sigma|\!) \to (\!|\rho|\!)$$

We build $\mathsf{MSL}(\omega)$ signatures $\Sigma^\#$ and $\Pi^\#$ by introducing one additional first-order monadic predicate symbol T to denote “truth”, and a new tree constructor $p^\#$ for each predicate symbol $P \in \Pi$:

$$\Sigma^\# := \{p^\# : (\!|\rho|\!) \mid P : \rho \in \Pi\} \cup \Sigma \qquad \Pi^\# := \{\mathsf{T} : \iota \to o\} \cup \Pi_1$$

The Term Transformation. Then define $(\!|t|\!)$ by:

$$(\!|x|\!) := x \qquad (\!|c|\!) := c \qquad (\!|P|\!) := p^\# \qquad (\!|s\,t|\!) := (\!|s|\!)\,(\!|t|\!)$$

By some abuse we write $(\!|\overline{s}|\!)$ to denote the pointwise transformation of a vector of terms $\overline{s}$. We extend this to goal formulas $(\!|G|\!)$ by:

$$(\!|\mathsf{true}|\!) := \mathsf{true} \qquad (\!|A|\!) := \mathsf{T}\,((\!|A|\!)) \qquad (\!|G \wedge H|\!) := (\!|G|\!) \wedge (\!|H|\!) \qquad (\!|\exists x.\,G|\!) := \exists x.\,(\!|G|\!)$$

where, by some abuse, we refer to the term-level transformation on the right-hand side of the second equation. We extend to definite formulas $(\!|C|\!)$ by:

$$\begin{aligned}
(\!|\mathsf{true}|\!) &:= \mathsf{true}\\
(\!|C \wedge D|\!) &:= (\!|C|\!) \wedge (\!|D|\!)\\
(\!|\forall \overline{y}.\,G \Rightarrow P\,\overline{y}|\!) &:= \forall \overline{y}.\,(\!|G|\!) \Rightarrow \mathsf{T}\,(p^\#\,\overline{y})\\
(\!|\forall \overline{y}.\,G \Rightarrow P\,(c\,\overline{y})|\!) &:= (\forall \overline{y}.\,(\!|G|\!) \Rightarrow P\,(c\,\overline{y})) \wedge (\forall \overline{z}.\,P\,z \Rightarrow \mathsf{T}\,(p^\#\,z))
\end{aligned}$$

We call the second conjunct of the last case of this definition the *reflection clause*. Note that the form of head in this case implies that $P : \iota \to o$ in Π. There is some obvious redundancy in that the image of the transformation will typically contain many copies of the same reflection clause, but this could be easily avoided if considered undesirable.

Finally, $D \vdash G$ iff $(\!|D|\!) \vdash (\!|G|\!)$ completes the reduction from Theorem 3.1.

3.2 Elimination of Existentials

The completeness of our proof system shows (as is usual for Horn logics) that our fragment has the existential witness property, that is: $D \models \exists x.\,G$ iff $D \models G[t/x]$ for some term t. Consequently, we can simulate existential quantifiers of tree constructor types using higher-order predicates. We introduce a family of new predicate symbols $\exists_\gamma$ indexed by γ and constrain them so that they hold of a given predicate p on γ whenever p holds for *some* term t.

For example, an existential quantifier $\exists_\iota : (\iota \to o) \to o$ on natural numbers, constructed using successor $\mathsf{s} : \iota \to \iota$ and zero $\mathsf{z} : \iota$, can be defined by the two clauses:

$$\forall p{:}\iota \to o.\, p\ \mathsf{z} \Rightarrow \exists_\iota\ p \qquad \forall p{:}\iota \to o.\, \exists_\iota(\lambda x.\, p\ (\mathsf{s}\ x)) \Rightarrow \exists_\iota\ p$$

However, since we do without λ-abstraction in our setting, we develop a version of the above with a kind of built-in lambda-lifting in which a predicate $\Lambda_{\gamma,G}$ models the λ-abstraction $\lambda x : \gamma.\, G$. Thus, an atom $\exists_\iota\ \Lambda_{\iota,G}$ will represent the goal formula $\exists x : \iota.\, G$.

Theorem 3.2. *Provability in MSL(ω) reduces to provability in existential-free MSL(ω).*

Because the elimination of higher-order predicates does not introduce existentials (Theorem 3.1), it suffices to reduce provability in MSL(ω) to provability in existential-free HOMSL(ω); given MSL(ω) definite formula D and goal formula G over constructor signature Σ and predicate signature Π, we construct a HOMSL(ω) definite formula $D_{\nexists}$ and goal formula $G_{\nexists}$ over constructor signature $\Sigma_{\nexists}$ and predicate signature $\Pi_{\nexists}$ such that $D \vdash G$ if, and only if, $D_{\nexists} \vdash G_{\nexists}$. Thanks to Corollary 2.5, we assume WLOG that D and G contain no existentials of type ρ.

The Existential-Free HOMSL(ω) Signature. From a given MSL(ω) constructor signature Σ and predicate signature Π, we construct HOMSL(ω) signatures $\Sigma_{\nexists} := \Sigma$ and $\Pi_{\nexists}$.

Let $\mathsf{sorts}_\exists(D \wedge G)$ contain the sorts of existential variables appearing in $D \wedge G$ and the arguments of any constructor from Σ. Furthermore, we define $\mathsf{goals}_\exists(D \wedge G)$ as all goal formulas G' such that $\exists x.\, G'$ appears in D or G. For the purpose of the definition, we assume that there is some fixed ordering on variables. We then define an extended predicate signature $\Pi_{\nexists}$ as follows:

$$\begin{aligned}\Pi_{\nexists} := \Pi &\cup \{\exists_\gamma : (\gamma \to o) \to o \mid \gamma \in \mathsf{sorts}_\exists(D \wedge G)\}\\ &\cup \{\mathsf{Comp}^{i,n}_f : (\gamma \to o) \to \gamma_1 \to \cdots \to \gamma_i \to o \mid f : \gamma_1 \to \cdots \to \gamma_n \to \gamma \in \Sigma,\ 0 \le i \le n\}\\ &\cup \{\Lambda_{\gamma,H} : \gamma_1 \to \cdots \to \gamma_k \to \gamma \to o \mid \gamma \in \Gamma,\ H \in \mathsf{goals}_\exists(D \wedge G),\ \mathsf{FV}(H) \setminus \{x\} = \{x_1, \dots, x_k\}\}\end{aligned}$$

where $\mathsf{Comp}^{0,n}_f : (\gamma \to o) \to o$, and, in the final summand, we require each $x_i : \gamma_i$. Intuitively, $\mathsf{Comp}^{i,n}_f$ denotes an eventual application of constructor f to n arguments, n being at most the arity of f, with $i \le n$ arguments already provided.

Existential-Free Goals. We map an MSL(ω) goal formula G to an existential-free HOMSL(ω) counterpart $\wr G \int$:

$$\wr \mathsf{true} \int := \mathsf{true} \qquad \wr A \int := A \qquad \wr G \wedge H \int := \wr G \int \wedge \wr H \int \qquad \wr \exists x : \gamma.\, G \int := \exists_\gamma\ (\Lambda_{\gamma,G}\, x_1\ \cdots\ x_k)$$

where, in the last clause, $\mathsf{FV}(G) \setminus \{x\} = \{x_1, \dots, x_k\}$, as sequenced by the assumed order.

Existential-Free Definite Formulas. We map an MSL(ω) definite formula C to an existential-free HOMSL(ω) definite formula $\wr C \int$ over $(\Sigma_{\nexists}, \Pi_{\nexists})$:

$$\wr \mathsf{true} \int := \mathsf{true} \qquad \wr C \wedge D \int := \wr C \int \wedge \wr D \int \qquad \wr \forall \overline{y}.\, G \Rightarrow P\,(c\,\overline{y}) \int := (\forall \overline{y}.\, \wr G \int \Rightarrow P\,(c\,\overline{y}))$$

The Equi-Provable Existential-Free HOMSL(ω) Instance. For any MSL(ω) definite formula D and goal formula G, we define an equi-provable existential-free HOMSL(ω) instance with definite formula $D_{\nexists}$ and goal formula $G_{\nexists} := \wr G \int$ [2].

[2]the clause head is $\mathsf{Comp}^{0,n}_f\ v$ when $i = 0$

$$
\begin{aligned}
D_{\nexists} := {} & \left\{ \mathsf{Comp}^{0,n}_{f}\, v \Rightarrow \exists_{\gamma}\, v \;\middle|\; f : \gamma_1 \to \cdots \to \gamma_n \to \gamma \in \Sigma, \exists_{\gamma} \in \Pi_{\nexists} \right\} \\
\cup & \left\{ v\ (f\ x_1\ \cdots\ x_n) \Rightarrow \mathsf{Comp}^{n,n}_{f}\ v\ x_1\ \cdots\ x_n \;\middle|\; f : \gamma_1 \to \cdots \to \gamma_n \to \gamma \in \Sigma \right\} \\
\cup & \left\{ \exists_{\gamma_{i+1}}\ (\mathsf{Comp}^{i+1,n}_{f}\ v\, x_1\ \cdots\ x_i) \Rightarrow \mathsf{Comp}^{i,n}_{f}\ v\, x_1\ \cdots\ x_i \;\middle|\; \begin{array}{l} f : \gamma_1 \to \cdots \to \gamma_n \to \gamma \in \Sigma, \\ 0 \le i < n \end{array} \right\} \\
\cup & \left\{ \lbag H \rbag \Rightarrow \Lambda_{\gamma, H}\ x_1 \cdots x_k\ x \;\middle|\; \gamma \in \mathsf{sorts}_{\exists}(D \wedge G),\ H \in \mathsf{goals}_{\exists}(D \wedge G) \right\} \cup \lbag D \rbag
\end{aligned}
$$

Partial instantiation of existential variables is key to eliminating them. After all, there may be countably many instantiations but only finitely many clauses. To this end, predicates $\mathsf{Comp}^{i,n}_{f}$ act as delayed applications of constructor f that come into effect when all n expected arguments to f are fully instantiated (when $i = n$ and the clause headed by $\mathsf{Comp}^{n,n}_{f}$ applies).

For example, since $g : \iota \to (\iota \to \iota) \to \iota \to \iota$ has tail type $\iota \to \iota$ when given two arguments, we may instantiate $x : \iota \to \iota$ in $\exists x.\, P\,(x\,b) \wedge Q\,(h\,x)$ with $g\,a\,f$ (for $a : \iota$ and $f : \iota \to \iota$), resulting in the existential-free proof in Figure 6.

It follows that, for goal formulas G that are subexpressions of the given instance, $D \vdash G$ iff $D_{\nexists} \vdash G_{\nexists}$. Provability of the latter reduces to provability of an existential-free MSL(ω) instance via the elimination of higher-order predicates (Theorem 3.1), which completes the reduction from Theorem 3.2.

$$
\begin{array}{c}
P\,(g\,a\,f\,b) \wedge Q\,(h\,(g\,a\,f)) \\ \hline
\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)}\ (g\,a\,f) \\ \hline
\mathsf{Comp}^{2,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})\ a\,f \\ \hline
\mathsf{Comp}^{0,0}_{f}\ (\mathsf{Comp}^{2,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})\ a) \\ \hline
\exists_{\iota\to\iota}\ (\mathsf{Comp}^{2,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})\ a) \\ \hline
\mathsf{Comp}^{1,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})\ a \\ \hline
\mathsf{Comp}^{0}_{a}\ (\mathsf{Comp}^{1,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})) \\ \hline
\exists_{\iota}\ (\mathsf{Comp}^{1,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})) \\ \hline
\mathsf{Comp}^{0,2}_{g}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)}) \\ \hline
\exists_{\iota\to\iota}\ (\Lambda_{\iota\to\iota,P\,(x\,b)\wedge Q\,(h\,x)})
\end{array}
$$

Fig. 6. Example existential-free proof

4 HIGHER-ORDER AUTOMATON CLAUSES AND THE DECISION PROCEDURE

Consider MSL(ω) $\Delta \vdash D$ and $\Delta \vdash G$ over constructor signature Σ and predicate signature Π. We aim to decide $D \vDash G$ by rewriting clauses to a solved form we call *(higher-order) automaton formulas*, after their first-order counterparts in Goubault-Larrecq [2002].

4.1 Higher-Order Resolution

In Weidenbach's original work on MSL and Goubault-Larrecq's later work on $\mathcal{H}1$ [Goubault-Larrecq 2005; Weidenbach 1999], satisfiability is decided by a form of ordered resolution: the given set of MSL clauses is saturated under the ordered resolution rule and satisfiability is determined according to the presence or absence of the empty clause.

There is a higher-order analogue of the resolution rule which also forms the core of a refutationally complete calculus for higher-order (constrained) Horn clauses [Ong and Wagner 2019]:

$$
\frac{G \wedge R\,\overline{s} \Rightarrow A \quad G' \Rightarrow R\,\overline{y}}{G \wedge G'[\overline{s}/\overline{y}] \Rightarrow A}\ \text{(HO-Resolution)}
$$

This higher-order rule has exactly the same structure as the standard first-order rule (for Horn clauses). However, as we shall describe below, this form of resolution on its own does not lend itself to a decision procedure for MSL(ω).

In the first-order case, the key to ensuring termination of saturation under resolution is to identify a certain kind of solved form of constraints, which are called *automaton*. Such clauses have shape: $Q_1(x_{\pi(1)}) \wedge \cdots \wedge Q_k(x_{\pi(k)}) \Rightarrow P(f(x_1, \ldots, x_m))$ with $1 \leq \pi(i) \leq m$ for all i. As remarked in the introduction, such formulas are nothing but a clausal representation of alternating tree automata, but for our purposes, there are two features to take note of: (a) they have a depth-1 head and each atom in the body of the clause is depth 0, and (b) there are no existentially quantified variables (variables that occur in the body but not in the head).

The ordering of the first-order resolution calculus is carefully crafted to ensure that the side premise of each resolution inference is automaton. It is easy to see that a resolution inference between an arbitrary MSL clause and an automaton clause will produce an MSL clause that is *strictly closer to automaton form*, whenever the selected negative literal has depth at least 1[3]:

$$\frac{G \wedge P(f(t_1, \ldots, t_n)) \Rightarrow A \quad Q_1(x_{\pi(1)}) \wedge \cdots \wedge Q_k(x_{\pi(k)}) \Rightarrow P(f(x_1, \ldots, x_m))}{G \wedge Q_1(t_{\pi(1)}) \wedge \cdots \wedge Q_k(t_{\pi(k)}) \Rightarrow A}$$

Since the body of an automaton clauses is required to contain only atoms with depth 0, we can think of the clause in the conclusion as closer to automaton form than the main premise (on the left-hand side) since the new atoms $Q_i(t_{\pi(i)})$ in the body replace an atom $P(f(t_1, \ldots, t_n))$ of strictly greater depth.

This is also the case for higher-order clauses in $\mathsf{MSL}(\omega)$ *whenever the selected negative literal is headed by a function symbol.* However, in higher-order clauses, the selected negative literal may be headed by a variable, and this spells trouble. Consider, for example, the following resolution inference. Recall that function application is conventionally left associative, so $h\, y_1\, y_2 = (h\, y_1)\, y_2$.

$$\frac{\mathsf{P}(x_1\ (\mathsf{f}\ \mathsf{a}\ x_2)) \Rightarrow \mathsf{Q}(\mathsf{g}\ x_1\ x_2) \quad \mathsf{R}(y_2) \wedge \mathsf{S}(y_2) \Rightarrow \mathsf{P}(\mathsf{h}\ y_1\ y_2)}{\mathsf{R}(\mathsf{f}\ \mathsf{a}\ x_2) \wedge \mathsf{S}(\mathsf{f}\ \mathsf{a}\ x_2) \Rightarrow \mathsf{Q}(\mathsf{g}\ (\mathsf{h}\ y_1)\ x_2)}$$

As before, the body of the clause in the conclusion can be viewed as closer to our automaton solved form, but the head of the clause is further away. In fact, the clause has departed the MSL fragment completely since it no longer has a shallow head! This is a significant problem because, by inspection, further resolution inferences with this non-MSL clause as the main premise can only produce clauses with a head of the same or even greater depth.

However, resolving on clauses where the selected negative literal is headed by a variable appears inescapable if we insist one of the premises of each resolution inference to be automaton:

$$\mathsf{true} \Rightarrow \mathsf{P}(\mathsf{h}\ y_1\ y_2) \qquad \mathsf{P}(x_1\ (\mathsf{f}\ \mathsf{a}\ x_2)) \Rightarrow \mathsf{Q}(\mathsf{g}\ x_1\ x_2) \qquad \mathsf{Q}(\mathsf{g}\ (\mathsf{h}\ z)\ \mathsf{a}) \Rightarrow \mathsf{false}$$

In this example, we can obtain a contradiction by resolution, but the only automaton clause is the first one, so there is no choice but to resolve the first and second, which leads to a deep head as above.

Our solution to this problem is to radically rethink the form of automaton clauses in the higher-order setting. We observe that a clause with a deep head $\mathsf{R}(\mathsf{f}\ \mathsf{a}\ x_2) \wedge \mathsf{S}(\mathsf{f}\ \mathsf{a}\ x_2) \Rightarrow \mathsf{Q}(\mathsf{g}\ (\mathsf{h}\ y_1)\ x_2)$ can be thought of as a clause with a shallow head $\mathsf{R}(\mathsf{f}\ \mathsf{a}\ x_2) \wedge \mathsf{S}(\mathsf{f}\ \mathsf{a}\ x_2) \wedge x_1 = h\, y_1 \Rightarrow \mathsf{Q}(\mathsf{g}\ x_1\ x_2)$ that contains an additional constraint $x_1 = h\, y_1$ in the body.

Of course, allowing arbitrary equational constraints (and especially at higher type) in the body will lead us immediately outside of a decidable fragment, so we cannot state such constraints directly. Rather, we ask only that the higher-order variable x_1 "behave like" $h\, y_1$. Since x_1 and $h\, y_1$ are both functions, the most obvious route to making this precise is to ask that they behave

[3]Negative literals with depth 0 are essentially already solved.

$$(\mathsf{Fact}_1)\ \frac{}{\vdash P\,c : o}\ c{:}\iota \in \Sigma \qquad (\mathsf{Fact}_2)\ \frac{}{x{:}\iota \vdash P\,x : o}$$

$$(\mathsf{ACl}_1)\ \frac{\overline{y{:}\gamma} \vdash U : o}{\vdash \forall \overline{y{:}\gamma}.\, U \Rightarrow P\,(f\,\overline{y}) : o}\ f{:}\overline{\gamma} \to \iota \in \Sigma \qquad (\mathsf{ACl}_2)\ \frac{\overline{y{:}\gamma} \vdash U : o}{x{:}\overline{\gamma} \to \iota \vdash \forall \overline{y{:}\gamma}.\, U \Rightarrow P\,(x\,\overline{y}) : o}$$

$$(\mathsf{True})\ \frac{}{\Delta \vdash \mathsf{true} : o} \qquad (\mathsf{And})\ \frac{\Delta_1 \vdash U : o \quad \Delta_2 \vdash V : o}{\Delta_1 \cup \Delta_2 \vdash U \wedge V : o}\ \Delta_1 = \emptyset \text{ iff } \Delta_2 = \emptyset$$

Fig. 7. Typing for automaton clauses and formulas

similarly on similar inputs. Moreover, there is a clear way to define similar, because we are only able to observe the behaviour of terms through the lens of our stock of predicate symbols[4].

Hence, to ask that x_1 behaves as $h\ y_1$ is to ask that x_1 satisfies $(\forall y_2.\ R(y_2) \wedge S(y_2) \Rightarrow P(x_1\ y_2))$. Clearly, $h\ y_1$ is an instance of x_1 that satisfies this constraint and, we claim, $\mathsf{MSL}(\omega)$ cannot distinguish between $h\ y_1$ and any other x_1 that also satisfies it.

Incorporating this leads to a kind of abductive inference, in which we infer an additional premise:

$$\frac{\mathsf{P}(x_1\ (\mathsf{f\ a}\ x_2)) \Rightarrow \mathsf{Q}(\mathsf{g}\ x_1\ x_2) \quad \mathsf{R}(y_1) \wedge \mathsf{S}(y_2) \Rightarrow \mathsf{P}(\mathsf{h}\ y_1\ y_2)}{\mathsf{R}(\mathsf{f\ a}\ x_2) \wedge \mathsf{S}(\mathsf{f\ a}\ x_2) \wedge (\forall y_2.\ \mathsf{R}(y_2) \wedge \mathsf{S}(y_2) \Rightarrow \mathsf{P}(x_1\ y_2)) \Rightarrow \mathsf{Q}(\mathsf{g}\ x_1\ x_2)}$$

Of course, we have still ended up outside the MSL fragment, but the additional power required to state this form of constraint on x_1 seems much less dangerous. This nested implication is none other than an MSL clause itself – the head is shallow and linear – and, moreover, its body is already in the correct form to be automaton. In fact we show that this form of nested clause is exactly the generalisation of automaton clause that we need in the higher-order setting.

4.2 Higher-Order Automaton Formulas

Higher-order automaton formulas allow for the nesting of clauses inside the body of other clauses. This makes them more properly a fragment of higher-order hereditary Harrop formulas (HOHH). In fact it is easy to see that they are HOHH formulas of a special kind, since they live in the intersection of HOHH goal and definite formulas [for HOHH see e.g. Miller et al. 1991].

Definition 4.1 (Automaton Formulas). Define the *automaton formulas*, typically U and V, by the following grammar:

$$(\mathsf{Automaton\ Fm}) \quad U, V \quad ::= \quad \mathsf{true} \mid U \wedge V \mid P\,x \mid P\,c \mid \forall \overline{y{:}\gamma}.\, U \Rightarrow P\,(f\,\overline{y}) \mid \forall \overline{y{:}\gamma}.\, U \Rightarrow P\,(x\,\overline{y})$$

Note: the form $P\,x$ concerns such a particular free x (i.e. it is *not* $\forall x.\, P\,x$). We identify formulas and clauses up to renaming of bound variables and we identify up to the commutativity, associativity, and idempotence of conjunction, with true as a unit, so that a formula U will be thought of equally well as a set of conjuncts. We only consider formulas $\Delta \vdash U : o$ that are well typed according to the judgement of Figure 7.

Each clause in U is essentially monadic, and by this we mean that it concerns either a single free variable x or a single constant c or f from the signature. This can be seen in the two pairs of rules (Fact_1), (ACl_1) and (Fact_2), (ACl_2), which have an empty and singleton typing context respectively.

In automaton formulas, clauses can be nested inside the body of another, and one function of the type system is to ensure some stratification to the nesting. In particular, in a closed automaton

[4]For example, if we only had a single predicate P then all terms s for which $P(s)$ is true are alike, we have no mechanism to write a constraint that distinguishes them.

formula (i.e. without free variables), top-level clauses can only concern constants from the signature and strictly nested clauses can only concern variables introduced by the clause that immediately contains them. To this end, note that the type context on the single premise of (ACI$_1$) and (ACI$_2$) contains exactly the variables $\overline{y{:}\gamma}$ introduced by the universal quantifier prefixing the immediately enclosing clause. The side condition of (And) guarantees that each conjunction of automaton clauses contains subjects from the same nesting level, rejecting ill-formed clauses like (5)-(8) below.

Since there is no weakening in general in this type system, the context $\overline{y{:}\gamma}$ introduced on the premise of (ACI$_1$) and (ACI$_2$) implies that the body U must contain a constraint concerning each of the variables in $\overline{y{:}\gamma}$. These variables can be distributed to the appropriate conjuncts of U that contain the corresponding constraint by the (And) rule. However, note that, despite the lack of weakening, it is possible to leave a subset of the variables, say $\overline{y'{:}\gamma'}$ unconstrained, but formally we must do that by introducing a true conjunct and discharge it by the judgement $\overline{y{:}\gamma'} \vdash \mathsf{true} : o$. Since, in practice, we will typically omit true conjuncts we will consider, for example, the clause $\forall xy.\, P\,x \Rightarrow R\,(a\,x\,y)$ to be well typed with $P : \iota \to o$ and $a : \iota \to \iota \to \iota$ by regarding the body $P\,x$ as sugar for $P\,x \wedge \mathsf{true}$.

For example, the following are closed automaton clauses over predicate symbols $P, Q, R : \iota \to o$ and tree constructor symbols $a : \iota$, $b : \iota \to \iota \to \iota$, $c : (\iota \to \iota) \to \iota \to \iota$, and $d : ((\iota \to \iota) \to \iota) \to \iota$.

$$P\,a \tag{1}$$
$$\forall xy.\, P\,x \wedge Q\,x \Rightarrow R\,(b\,x\,y) \tag{2}$$
$$\forall xy.\, Q\,y \wedge (\forall z.\, Q\,z \Rightarrow R\,(x\,z)) \Rightarrow P\,(c\,x\,y) \tag{3}$$
$$\forall x.\, (\forall y.\, (\forall z.\, P\,z \wedge Q\,z \Rightarrow R\,(y\,z)) \Rightarrow R\,(x\,y)) \Rightarrow P\,(d\,x) \tag{4}$$

Note: strictly speaking clause (2) must be constructed as e.g. $\forall xy.\, P\,x \wedge Q\,x \wedge \mathsf{true} \Rightarrow R\,(b\,x\,y)$ (with true representing the constraint on b's argument y). However, the following are *not* well formed as *closed* automaton clauses:

$$P\,x \tag{5}$$
$$\forall xy.\, P\,x \wedge Q\,a \Rightarrow R\,(b\,x\,y) \tag{6}$$
$$\forall xy.\, Q\,y \wedge (\forall z.\, Q\,z \wedge R\,y \Rightarrow R\,(x\,z)) \Rightarrow P\,(c\,x\,y) \tag{7}$$
$$\forall x.\, (\forall y.\, (\forall z.\, P\,z \wedge (\forall z'.\, Q\,z' \Rightarrow R\,(y\,z')) \Rightarrow R\,(y\,z)) \Rightarrow R\,(x\,y)) \Rightarrow P\,(d\,x) \tag{8}$$

In (5) we have a free variable x, yet the clause is supposed to be closed. In (6) we have an atom $Q\,a$ concerning a constant that appears in a strictly nested position and in (7) we have an atom $R\,y$ that concerns a variable from an outer scope – predicates that appear in nested clauses can only concern variables, and only variables that are introduced by the clause that immediately contains them. This is also the problem in clause (8), where the nested clause $(\forall z'.\, Q\,z' \Rightarrow R\,(y\,z'))$ concerns y, but y is not a variable introduced by the immediately enclosing clause $(\forall z.\, P\,z \ldots \Rightarrow R\,(y\,z))$, which introduces only z.

Notation. If U *is automaton wrt* $\overline{y}\,\overline{z}$ (i.e. $\overline{y}$ and $\overline{z}$ are the free variables in U), we write $U_{|\overline{y}}$ and $U_{|\overline{z}}$ for the partition $U = U_{|\overline{y}} \wedge U_{|\overline{z}}$ according to whether the automaton clauses in U contain a (free) variable from either $\overline{y}$ or $\overline{z}$.

Say that an automaton clause T of shape $\forall \overline{y}.\, U \Rightarrow P\,(\xi\,\overline{y})$ (for ξ either a variable or a constant) is *order n* just if the type of ξ is order n. Note that the type system ensures that clauses that are nested in the body of an order-n clause are of strictly smaller order. In the following, $\exp_n(m)$ denotes a tower of exponentials of height $n + 1$, with $\exp_1(m) = 2^m$, $\exp_2(m) = 2^{2^m}$, etc.

Theorem 4.2. *Let k denote the largest arity of any function symbol in Σ and $|\Pi|$ denote the number of predicate symbols in Π. There are $O(\mathrm{exp}_n(k|\Pi|))$ clauses T such that $x : \gamma \vdash T : o$, for $\mathsf{ord}(\gamma) = n$.*

Proof. Fix an order-1 variable x and consider an order-1 clause $\forall \overline{y}.\, U \Rightarrow P\ (x\ \overline{y})$. Since every variable of $\overline{y}$ is of type ι, the body U cannot contain any nested clauses, so every conjunct is of the form $P'y$. Hence, order-1 clauses coincide with the automaton clauses of first-order MSL, and it is easy to see that there are at most $|\Pi| \cdot 2^{|\Pi|^k} = O(2^{k|\Pi|})$ different such, where k is the maximum arity of any function symbol. Now consider an order-$(n+1)$ clause $\forall \overline{y}.\, U \Rightarrow P\ (x\ \overline{y})$. In the worst case, each variable y of $\overline{y}$ is order n, and we may assume there are $O(\mathrm{exp}_n(k|\Pi|))$ clauses that concern y. Hence, choosing this automaton clause amounts to choosing the predicate P in the head and then, for each variable $y \in \overline{y}$, choosing some subset of the $O(\mathrm{exp}_n(k|\Pi|))$ different clauses that can be nested inside U. Hence, we have $O(\mathrm{exp}_{n+1}(k|\Pi|))$ many clauses at most. □

Automaton clauses T that concern a function symbol $f : \gamma$, i.e. for which $\vdash T : o$ holds, are just automaton clauses of the above form in which we have replaced the variable x by f. Hence:

Corollary 4.3. *There are finitely many automaton clauses of a given order.*

4.3 Decision Procedure

Our decision procedure takes a set of $\mathsf{MSL}(\omega)$ definite clauses D as input and iteratively rewrites them into automaton form V. By construction, the new set of automaton clauses V is sufficiently strong, though generally weaker than D, to entail any goal G that follows from D.

Although we have so far been discussing resolution on clauses, the rewriting will be defined only for goals. This is because rewriting will introduce nested clauses, and it seems easier to reason with nested clauses compositionally *a la* Miller et al. [1991].

Definition 4.4 (Rewriting). Given an automaton formula V, variables $\overline{y}$, and two goal clauses G and H, we introduce the rewrite relation $V, \overline{y} \vdash G \triangleright H$ defined by the rules in Figure 8. Note that, in (Assm), we implicitly assume that the length of $\overline{z}$ is the same as the length of $\overline{s}$.

The rules are mostly straightforward, and consist of simulating certain standard logical inference steps directly on the syntax of the formula. The heavy lifting is done by (Step) and (Assm) which simulate resolution steps on a goal. The first, (Step), applies when the head symbol of the goal is a function symbol and the second, (Assm), applies when it is a variable. Thus the latter provides the abductive method of adding an assumption described above.

The automaton formula V is the set of automaton clauses that we are allowed to use when performing resolution steps and the variables $\overline{y}$ that appear before the turnstile are the set of variables that we are willing to make additional assumptions about, via (Assm). The idea is that, although we are only rewriting goal formulas $V, \overline{y} \vdash G \triangleright^* H$, we can think of the goal formula G as the body of an $\mathsf{MSL}(\omega)$ clause $\forall \overline{y}.\, G \Rightarrow A$. Then the outcome of rewriting, namely H, will give us a new clause $\forall \overline{y}.\, H \Rightarrow A$, but we need to be sure we have only made additional assumptions about the top-level universally quantified variables $\overline{y}$. This is justified by the following theorem and the remarks that follow.

Theorem 4.5 (Soundness). *If $V, \overline{y} \vdash G \triangleright^* H$ then $V \models \forall \overline{y}.\, H \Rightarrow G$.*

It follows that, if $V, \overline{y} \vdash G \triangleright^* H$, then $V \wedge (\forall \overline{y}.\, G \Rightarrow A) \models (\forall \overline{y}.\, H \Rightarrow A)$. In practice, we are only interested in rewriting sequences that start with the body of a non-solved clause and end in an automaton formula, giving us new automaton clauses. Termination of branches that have successfully reached automaton form is ensured by e.g. the side conditions of (Imp) and (Scope). The new automaton clauses arising this way unlock additional avenues to rewrite the remaining

$$P\,x \in V \quad \dfrac{}{V,\,\overline{y} \vdash P\,x \rhd \mathsf{true}}\,(\text{Refl}) \qquad (\forall \overline{x}.\, U \Rightarrow P\,(f\,\overline{x})) \in V \quad \dfrac{}{V,\,\overline{y} \vdash P\,(f\,\overline{s}) \rhd U[\overline{s}/\overline{x}]}\,(\text{Step})$$

$$(\forall \overline{xz}.\, U \Rightarrow P\,(f\,\overline{xz})) \in V \quad \dfrac{}{V,\,\overline{y} \vdash P\,(y\,\overline{s}) \rhd U_{|\overline{z}}[\overline{s}/\overline{z}] \wedge (\forall \overline{z}.\, U_{|\overline{z}} \Rightarrow P\,(y\,\overline{z}))}\,(\text{Assm})$$

$$\dfrac{V,\,\overline{y} \vdash G \rhd G'}{V,\,\overline{y} \vdash G \wedge H \rhd G' \wedge H}\,(\text{AndL}) \qquad \begin{array}{c} G \neq P\,(y\,\overline{z}) \\ \text{for any } y \in \overline{y} \end{array} \quad \dfrac{U \wedge V,\,\overline{y} \vdash G \rhd G'}{V,\,\overline{y} \vdash (\forall \overline{z}.\, U \Rightarrow G) \rhd (\forall \overline{z}.\, U \Rightarrow G')}\,(\text{Imp})$$

$$\dfrac{V,\,\overline{y} \vdash H \rhd H'}{V,\,\overline{y} \vdash G \wedge H \rhd G \wedge H'}\,(\text{AndR}) \qquad \overline{z} \cap \mathsf{FV}(G) = \emptyset \quad \dfrac{}{V,\,\overline{y} \vdash \forall \overline{z}.\, U \Rightarrow G \rhd G}\,(\text{Scope})$$

$$\dfrac{}{V,\,\overline{y} \vdash (\forall \overline{z}.\, G_1 \Rightarrow G_2 \wedge G_3) \rhd (\forall \overline{z}.\, G_1 \Rightarrow G_2) \wedge (\forall \overline{z}.\, G_1 \Rightarrow G_3)}\,(\text{ImpAnd})$$

Fig. 8. Rewrite system

MSL(ω) clauses (via (Step) and (Assm)) and this, in turn, generates more automaton clauses and so on. It follows from Corollary 4.3 that, for a given set of definite clauses D, the limit, $\mathcal{V}(D)$, is a finite object.

Definition 4.6 (Canonical solved form). Define the *canonical solved form*, written $\mathcal{V}(D)$, of a set of definite clauses D inductively by the following rule. The base case – when V is empty – occurs for definite clauses that are already automaton, like $\forall y_1 y_2.\, P\,y_1 \wedge R\,y_1 \Rightarrow S\,(f\,y_1\,y_2)$.

$$\begin{array}{r|c} (\forall \overline{y}.\, G \Rightarrow A) \in D & \dfrac{V \subseteq \mathcal{V}(D)}{(\forall \overline{y}.\, U \Rightarrow A) \in \mathcal{V}(D)} \\ V,\,\overline{y} \vdash G \rhd^* U & \end{array}$$

Since every clause that we add to $\mathcal{V}(D)$ is weaker than a clause in D, we have $\mathcal{V}(D) \models G$ implies $D \models G$. It remains to show the converse, from which we deduce that satisfiability can be decided by computing $\mathcal{V}(D)$ and then checking whether or not the goal follows.

Theorem 4.7 (Completeness). *If $D \vdash G$ then $\mathcal{V}(D) \models G$.*

It follows from the fact that there are only finitely many automaton clauses that the canonical solved form is finite. Furthermore, as the rewrite system is well founded and terminates, we can effectively decide whether a given automaton clause is in the canonical solved form. Hence, we can compute $\mathcal{V}(D)$ by enumerating the possible automaton clauses and checking if they are the solved form of any clause in D.

Theorem 4.8 (Decidability). *Let V be an automaton formula, $\overline{y}$ variables, G an goal formula and U an automaton formula. Then $V,\,\overline{y} \vdash G \rhd^* U$ is decidable. Hence, $\mathcal{V}(D)$ is computable.*

Remark. In principle, all fragments identified in Section 2.2 can be decided by our decision procedure. Strictly speaking, the procedure takes as input an existential-free formula of MSL(ω) (and thus of any of the syntactic subfragments MSL(n)). However, our translations from Section 3 allow for transforming any formula of HOMSL(ω) or a HOMSL(n) subfragment thereof into an equi-satisfiable existential-free formula of MSL(ω). Existentials have been eliminated for simplicity; we could have introduced new rules to the calculus to handle them natively.

4.4 Rewriting Example

By way of an example, we show how to use rewriting to obtain automaton clauses (i.e. a subset of $\mathcal{V}(D)$) from the given formulas in Figure 3 that allow for a replay of the proof in Figure 4. One can view this as a concrete example of the completeness proof in action. The strategy of the completeness argument is to start from the leaves of the proof tree, where resolution steps already involve automaton clauses[5], and work back towards the root. We start at the leaf labelled with (18).

Since clauses (18) and (7) are already automaton, we can use (7) to rewrite the body of clause (10) using (Assm):

$$\mathsf{V}(x\ h\ (\mathsf{putCont}\ k)) \triangleright \mathsf{Closed}(h) \wedge (\forall h\, k.\ \mathsf{Closed}(h) \Rightarrow \mathsf{V}\ (x\ h\ k))$$

From this, we obtain (10'): $\mathsf{Closed}(h) \wedge (\forall h\, k.\ \mathsf{Closed}(h) \Rightarrow \mathsf{V}\ (x\ h\ k)) \Rightarrow \mathsf{V}(\mathsf{putStrLn}\ x\ h\ k)$ is an automaton clause in $\mathcal{V}(D)$. Continuing down the tree, we can use (10') to rewrite the body of (16) using (Step):

$$\mathsf{V}(\mathsf{putStrLn}\ x\ h_3\ \mathsf{id}) \triangleright \mathsf{Closed}(h_3) \wedge (\forall h\, k.\ \mathsf{Closed}(h) \Rightarrow \mathsf{V}\ (x\ h\ k))$$

thus obtaining (16'): $\mathsf{Closed}(h_3) \wedge (\forall h\, k.\ \mathsf{Closed}(h) \Rightarrow \mathsf{V}\ (x\ h\ k)) \Rightarrow \mathsf{V}(\mathsf{act}_2\ x\ h)$, another automaton clause. Following the proof branch down, we can use (16') to rewrite the body of (14) using (Assm):

$$\mathsf{V}(k\ y\ h_2) \ \triangleright \ \begin{array}{l} \mathsf{Closed}(h_2) \wedge (\forall h\, k.\ \mathsf{Closed}(h) \Rightarrow \mathsf{V}\ (y\ h\ k)) \\ \quad \wedge (\forall x\, h.\ \mathsf{Closed}(h) \wedge (\forall h\, k.\ \mathsf{Closed}(h) \Rightarrow \mathsf{V}\ (x\ h\ k)) \Rightarrow \mathsf{V}(k\ x\ h)) \end{array}$$

and, naming that formula U for brevity, we obtain (14'): $\forall y\, k\, h_2.\ U \Rightarrow \mathsf{V}(\mathsf{withFile}_3\ y\ k\ h_2)$. At any point we can stop and the automaton clauses we have obtained are sufficient to entail the goal at the same place in the tree. For example, it is easy to check that any model of the automaton clauses we have collected so far satisfies $\mathsf{V}(\mathsf{withFile}_3\ \mathsf{read}\ \mathsf{act}_2\ \mathsf{c})$.

5 AUTOMATON CLAUSES: CONNECTING LOGIC, TYPES, AND AUTOMATA

Recall that our ‘automaton clauses’ are named after their first-order counterparts in Goubault-Larrecq [2002]. We argue that this name is equally justified for our higher-order automaton clauses.

While it is easy to see that a first-order automaton clause $\forall x\, y.\, P\, x \wedge Q\, y \wedge R\, y \Rightarrow S\,(f\, x\, y)$ fits the shape of a transition in a finite tree automaton (if t_1 is accepted from state P and t_2 from states Q and R, then $f\ t_1\ t_2$ is accepted from state S), this relation is more complicated for higher-order automaton clauses with variable-headed atoms and nested clauses.

Nonetheless, our automaton clauses share a vital trait with automaton transitions: the monadic, shallow, linear heads ‘peel’ a single constructor off a tree and separate its children without interdependencies. Furthermore, the fact that the number of automaton clauses is n-exponential in the order n of the program (Theorem 4.2) suggests automaton clauses could be a *defunctionalisation* of MSL(ω) and, thus, ‘first order’ in a sense – maybe even regular, given they are essentially monadic.

This intuition turns out to be true: automaton clauses correspond to intersection types, which are known to give rise to *alternating tree automata*, see Rehof and Urzyczyn [2011] and Broadbent and Kobayashi [2013], that are equivalent to ordinary non-deterministic finite tree automata.

This correspondence between automaton clauses and intersection types allows us to reduce MSL(ω) provability to intersection untypeability and vice versa, giving us a new algorithm for solving higher-order recursion scheme (HORS) model checking along the way.

[5] A resolution step whose conclusion is a leaf must use a clause of shape true $\Rightarrow A$ as premise, which is already automaton.

5.1 Correspondence with Intersection Types

In the remainder of this section, we reserve the word *term* for terms of constructor types and τ for strict types. We follow the practice from HORS model checking in stratifying the definitions of intersection types for convenience.

Fix $(q \in)\ Q$ as a set of atomic *base types*. We define the *intersection types* and *strict types* over Q:

$$\text{(Strict Types)}\ \ \tau \ ::=\ q \mid \sigma_{\mathrm{t}} \to \tau \qquad \text{(Intersection Types)}\ \ \sigma_{\mathrm{t}} \ ::=\ \textstyle\bigwedge_{i=1}^{n} \tau_i$$

Arrows associate to the right; intersections bind tightest. An *intersection type environment*, written Γ, is a finite, partial function from variables to intersection types. We denote an arbitrary strict or intersection type by θ and refer to it WLOG as an intersection type (since strict types are intersection types with $n = 1$). We assume types are well typed, where base types q have type ι.

An *intersection (refinement) type system* is a triple $(\Sigma, Q, \mathsf{type})$ in which Σ is a signature of constants, Q a set of base types equipped with a preorder Θ, and type assigns an intersection type of type σ to each $a : \sigma \in \Sigma$. The *subtype relation* over Q is the least relation on types, written $\leq$, that validates standard subtyping rules (see the long version of this work for details).

Terms can be typed in the following way:

$$\text{(T-Con)}\ \frac{}{\Gamma \vdash c :: \tau_i}\ \mathsf{type}(c) = \textstyle\bigwedge_{i=1}^{n} \tau_i \qquad \text{(T-Var)}\ \frac{}{\Gamma, x :: \bigwedge_{i=1}^{n} \tau_i \vdash x :: \tau_i}$$

$$\text{(T-App)}\ \frac{\Gamma \vdash s :: \sigma_{\mathrm{t}} \to \tau \quad \Gamma \vdash t :: \tau_i\ (\forall i \in [1..n])}{\Gamma \vdash s\, t :: \tau}\ \textstyle\bigwedge_{i=1}^{n} \tau_i \leq \sigma_{\mathrm{t}}$$

We may write $\Gamma \vdash t :: \sigma_{\mathrm{t}}$ as a shorthand for $\bigwedge_{\tau \in \sigma_{\mathrm{t}}} (\Gamma \vdash t :: \tau)$.

We consider intersection type systems $(\Sigma, Q, \mathsf{type})$ with a trivial preorder, i.e. $\Theta = \{(q, q) \mid q \in Q\}$. This is not a restriction, because non-trivial preorders can be simulated with fresh base types. For example, $\mathsf{nat} \leq \mathsf{int}$ can be enforced with a fresh base type pos by replacing nat by $\mathsf{int} \wedge \mathsf{pos}$.

5.1.1 Type-Clause Correspondence. Recall that automaton clauses are essentially monadic; each automaton clause contains precisely one symbol of type γ that is not locally bound, either a free variable or a constructor from signature Σ.

Given an automaton clause T with top-level variable x, an instantiation $T[t/x]$ rewrites to true just if t satisfies the constraints imposed by T. Clearly, $T[t/x]$ rewrites to true iff $T[y/x][t/y]$ does; after all, x and y are the only free variables in T and $T[y/x]$, resp. This means automaton clauses T and $T[y/x]$ are indistinguishable wrt the constraints they impose on their respective variables.

In this section, we discuss the constraints imposed by automaton clauses and formulas. It will be helpful to think of an automaton clause as *forgetful* with regards to its top-level symbol. As we shall see below, forgetful automaton clauses (hereafter just 'automaton clauses') correspond to strict intersection types over base types Π; an instantiated automaton clause $T[t/x]$ rewrites to true precisely when closed term t has the strict type corresponding to T and vice versa.

Automaton formulas and intersection types are simply two (equivalent) ways of imposing constraints on t.

Definition 5.1 (Correspondence between automaton clauses and intersection types). The following typing rules define a one-to-one correspondence between intersection types and $\mathsf{MSL}(\omega)$ automaton formulas. We assign an intersection type θ of type $\sigma_1 \to \cdots \to \sigma_m \to \iota$ to automaton formula $U = U_{|x_1} \wedge \cdots \wedge U_{|x_m}$ with $x_1 : \sigma_1, \ldots, x_m : \sigma_m$, written $U :: \theta$:

$$\frac{}{\mathsf{true} :: \top} \qquad \frac{}{P\,\xi :: q_P} \qquad \frac{U_{|x_1} :: \theta_1 \quad \ldots \quad U_{|x_m} :: \theta_m}{(\forall \overline{x}.\, U \Rightarrow P\,(\xi\,\overline{x})) :: \theta_1 \to \cdots \to \theta_m \to q_P} \qquad \frac{U_{1\,|x} :: \theta_1 \quad U_{2\,|x} :: \theta_2}{U_{1\,|x} \wedge U_{2\,|x} :: \theta_1 \wedge \theta_2}$$

where $q_P \in Q$ is a basetype and ξ a variable or constructor from Σ.

We write $\mathsf{fromCl}(U) = \theta$ for $U :: \theta$, and $\mathsf{toCl}(\theta)(\overline{f}) = U$ for $U :: \theta$ with $\overline{f}$ as top-level constructors. We write $U \leq_{\mathrm{a}} U'$ for U, U' over $x_1 \ldots x_m$ just if $\forall i \in [1..m].\, \mathsf{fromCl}(U_{|x_i}) \leq \mathsf{fromCl}(U'_{|x_i})$.

Example 5.2. Let $U = \forall v\, x\, y.\, Q\, y \wedge (\forall z.\, Q\, z \wedge P\, z \Rightarrow R\,(x\, z)) \Rightarrow P\,(c\, v\, x\, y)$. The following hold:

$$\mathsf{fromCl}(U) = \top \rightarrow (q_Q \wedge q_P \rightarrow q_R) \rightarrow q_Q \rightarrow q_P$$
$$U = \mathsf{toCl}(\top \rightarrow (q_Q \wedge q_P \rightarrow q_R) \rightarrow q_Q \rightarrow q_P)(c)$$

This is witnessed by:

$$\dfrac{\dfrac{}{\mathsf{true} :: \top} \qquad \dfrac{\dfrac{}{Q\, z :: q_Q} \quad \dfrac{}{P\, z :: q_P}}{\forall z.\, Q\, z \wedge P\, z \Rightarrow R\,(x\, z) :: q_Q \wedge q_P \rightarrow q_R} \qquad \dfrac{}{Q\, y :: q_Q}}{\forall v\, x\, y.\, Q\, y \wedge (\forall z.\, Q\, z \wedge P\, z \Rightarrow R\,(x\, z)) \Rightarrow P\,(c\, v\, x\, y) :: \top \rightarrow (q_Q \wedge q_P \rightarrow q_R) \rightarrow q_Q \rightarrow q_P}$$

Lemma 5.3 (Order isomorphism). *Let $\theta, \theta_1, \theta_2$ be intersection types of type $\sigma_1 \rightarrow \cdots \rightarrow \sigma_m \rightarrow \iota$ and U, U_1, U_2 automaton clauses over $\overline{x} = x_1 \ldots x_m$ with $x_i : \sigma_i$, for all $i \in [1..m]$. The following hold:*

(i) $\mathsf{toCl}(\mathsf{fromCl}(U))(\overline{x}) = U$
(ii) $\mathsf{fromCl}(\mathsf{toCl}(\theta)(\overline{x})) = \theta$
(iii) $\theta_1 \leq \theta_2$ *if, and only if,* $\mathsf{toCl}(\theta_1)(\overline{x}) \leq_{\mathrm{a}} \mathsf{toCl}(\theta_2)(\overline{x})$
(iv) $\mathsf{fromCl}(U_1) \leq \mathsf{fromCl}(U_2)$ *if, and only if,* $U_1 \leq_{\mathrm{a}} U_2$

This isomorphism between automaton formulas and intersection types extends to intersection type systems $(\Sigma, Q, \mathsf{type})$ and $\mathsf{MSL}(\omega)$ automaton formulas V over signatures (Π, Σ'): $\mathsf{MSL}(\omega)$ predicates Π correspond to basetypes Q; automaton environment V corresponds to intersection type environment $\Gamma \uplus \mathsf{type}$, both with domain $\mathsf{dom}(\Gamma) \cup \Sigma = \Sigma'$.

Definition 5.4 (Correspondence between automaton formulas and intersection type systems). We map intersection type system $(\Sigma, Q, \mathsf{type})$ with type environment Γ to $\mathsf{MSL}(\omega)$ automaton formula $V_{\Sigma,\Gamma}$ over signatures $(\Pi_Q, \Sigma \cup \mathsf{dom}(\Gamma))$ using:

$$\Pi_Q := \{P \mid q_P \in Q\} \qquad V_{\Sigma,\Gamma} := \{\mathsf{toCl}(\mathsf{type}(f))(f) \mid f \in \Sigma\} \cup \{\mathsf{toCl}(\theta)(a) \mid a :: \theta \in \Gamma\}$$

In the converse direction, we map an $\mathsf{MSL}(\omega)$ automaton formula V over (Π, Σ) to an intersection type system $(\emptyset, Q_\Pi, \emptyset)$ with type environment Γ_V:

$$Q_\Pi := \{q_P \mid P \in \Pi\} \qquad \Gamma_V := \left\{a :: \theta \,\middle|\, a \in \mathsf{dom}(V) \wedge \mathsf{fromCl}(V_{|a}) = \theta\right\}$$

The above is not strictly a one-to-one correspondence, because the separate environments type and Γ for intersection-type 'constants' and 'variables', resp., are mapped to a single automaton formula V. This distinction cannot be recovered in the other direction. Note, however, that (T-Con) and (T-Var) both choose a strict type from $\Gamma \uplus \mathsf{type}$ for a symbol. For the typeability of a term it does not matter whether a symbol lives in the domain of Γ or type.

Since the same typing judgements hold for $(\Sigma, Q, \mathsf{type})$ with type environment Γ as for $(\emptyset, Q, \emptyset)$ with type environment $\Gamma \uplus \mathsf{type}$, we identify $\Gamma_{V_{\Sigma,\Gamma}}$ and Γ with type left implicit. Clearly, $V_{\Gamma_V} = V$, which gives us a one-to-one correspondence after all. Furthermore, because $\Gamma_{V_{\Sigma,\Gamma}}$ and Γ type the same terms, we may assume WLOG that our intersection type systems do not contain constants, thus, $V_{\Sigma,\Gamma} = V_\Gamma$, giving rise to the following lemma.

Lemma 5.5. $V_{\Gamma_V} = V$ *and* $\Gamma_{V_\Gamma} = \Gamma$.

5.1.2 Typing-Rewriting Correspondence. Unless otherwise specified, we consider only goal formulas G that result from rewriting some existential-free $\mathsf{MSL}(\omega)$ goal formula G', i.e. $V, \overline{y} \vdash G' \rhd^* G$. Such a G is itself existential-free, and all its implications have automaton bodies, as rewriting does not introduce existentials and only introduces implications with automaton bodies.

Given an isomorphic type environment and automaton formula as per Definition 5.4, we can now formalise the equivalence between typing judgements and rewrites.

Proposition 5.6 (Typing-rewriting correspondence).

(i) For all U over $\overline{y}$, there exists U' such that $U \leq_{\mathrm{a}} U'$ and $V, \overline{y} \vdash \mathsf{toCl}(\sigma_{\mathrm{t}})(t) \rhd^* U'$ if, and only if, $\Gamma_V, \overline{y} :: \mathsf{fromCl}(U) \vdash t :: \sigma_{\mathrm{t}}$.

(ii) $V \vdash \mathsf{toCl}(\sigma_{\mathrm{t}})(t) \rhd^* \mathsf{true}$ if, and only if, $\Gamma_V \vdash t :: \sigma_{\mathrm{t}}$

Thanks to $\Gamma_{V_\Gamma} = \Gamma$, the above not only provides an equivalence between V and Γ_V but also between V_Γ and Γ.

5.2 Reducing HORS Intersection Typing to MSL(ω)

One of the most well-studied problems in higher-order model checking is the safety problem for higher-order recursion schemes (HORS): does the tree $[\![\mathcal{G}]\!]$ generated by HORS $\mathcal{G}$ satisfy safety property φ? The property is typically expressed as an alternating trivial automaton – or its negation as an alternating cotrivial automaton. This problem reduces to intersection (un)typeability [Kobayashi 2009], which we shall use.

A *higher-order recursion scheme* (HORS) is a set $\mathcal{R}$ of well-typed ground definitions of type ι (i.e. the RHSs are applicative terms over the formal parameters, constants Σ_{con}, and variables $\mathcal{N}$):

$$\{f_1\,\overline{y_1} = t_1, \quad \ldots, \quad f_n\,\overline{y_n} = t_n\}$$

where $f_1, \ldots, f_n \in \mathcal{N}$ and $\overline{y_1}, \ldots, \overline{y_n} \in \mathsf{Vars}$ are vectors of distinct variables. There is a designated nullary start symbol $S : \iota$, so a HORS can be denoted by a quadruple $\mathcal{G} = \langle \mathcal{N}, \Sigma_{\mathrm{con}}, \mathcal{R}, S \rangle$.

See e.g. Ong [2006] for a full account of HORS.

For the HORS safety problem, we assume a HORS comes equipped with base types Q_ι and a *negative* typing of constants Σ_{con}, denoted by type, with respect Q_ι. This gives rise to an intersection type system $(\Sigma_{\mathrm{con}}, Q_\iota, \mathsf{type})$. Typically, we will ask whether $\vdash S :: q_0$ in this system, for some $q_0 \in Q_\iota$.

Negative Types. Negative constant types are easily computable via a DeMorgan dual; if we read a (positive) type $f :: \bigwedge_{i \in [1..n]}(\sigma_{\mathrm{t}i,1} \to \cdots \to \sigma_{\mathrm{t}i,m} \to q_P)$ as $f\, t_1 \ldots t_m :: q_P$ iff $\bigvee_{i \in [1..n]} \bigwedge_{j \in [1..m]} t_j :: \sigma_{\mathrm{t}i,j}$, then the corresponding negative type is the DeMorgan dual of this boolean formula (see e.g. Muller and Schupp [1987], who use this construction to negate alternating tree automata).

Since we use only negative types but forget about this for the remainder of the section, we shall simply write $t :: \tau$ to mean t has negative type τ.

Example 5.7. Consider integer-division operator div with an error type:

$$(\top \to \mathsf{zero} \to \mathsf{err}) \wedge (\mathsf{err} \to \top \to \mathsf{err}) \wedge (\top \to \mathsf{err} \to \mathsf{err})$$

This means that n div m :: err iff $(n :: \top \wedge m :: \mathsf{zero}) \vee (n :: \mathsf{err} \wedge m :: \top) \vee (n :: \top \wedge m :: \mathsf{err})$. The corresponding negative type can be computed via DeMorgan as $(\mathsf{err} \to \mathsf{zero} \wedge \mathsf{err} \to \mathsf{err})$, which should be read as: "n div m does not have type err if neither n or m has type err and m does not have type zero."

Example 5.8. Consider order-2 HORS $\mathcal{G}_2 = \langle \{S, F, B\}, \{\mathsf{c}, \mathsf{s}, \mathsf{z}\}, \mathcal{R}, S \rangle$ that generates a binary c-labelled spine whose left subtrees are unary $\mathsf{s}^{2^0}\,\mathsf{z}$, $\mathsf{s}^{2^1}\,\mathsf{z}$, $\mathsf{s}^{2^2}\,\mathsf{z}$, etc. in that order, with $\mathcal{R}$ given by:

$$S = F\,\mathsf{s} \qquad F\,\varphi = \mathsf{c}\,(\varphi\,\mathsf{z})\,(F\,(B\,\varphi\,\varphi)) \qquad B\,\varphi\,\psi\,x = \varphi\,(\psi\,x)$$

Let base types $Q = \{q_S, q_0, q_1, q_2, q_3\}$ count the number of s in a sequence modulo 4. Because we are considering a negative typing, the constant z has type $q_1 \wedge q_2 \wedge q_3$ instead of q_0 and s type $\bigwedge_{i \in [0..3]} (q_i \to q_{i+1 \bmod 4})$. Let q_S model "has no odd subtrees", so c has negative type:

$$\mathsf{type}(\mathrm{c}) = (q_1 \wedge q_3 \to q_S \to q_S)$$

Now we may ask whether $\vdash S :: q_S$. Note that type(c) means a c-headed tree has no odd subtrees if the left (s-headed) subtree isn't odd and the right (c-headed) subtree does not have odd subtrees.

The HORS Untypeability Problem. We call an intersection type environment Γ *$\mathcal{G}$-coconsistent* just if, (1) Γ is empty, or (2) there exist $f :: \tau \in \Gamma$ and $(f\,\overline{y} = t) \in \mathcal{G}$ such that $\Gamma \backslash \{f :: \tau\}$ is $\mathcal{G}$-coconsistent and $\Gamma \backslash \{f :: \tau\}, \overline{y} :: \overline{\sigma_{\mathrm{t}}} \vdash t :: q_P$, where $\tau = \sigma_{\mathrm{t}1} \to \cdots \to \sigma_{\mathrm{t}m} \to q_P$. Intuitively, every intersection type in a $\mathcal{G}$-coconsistent Γ is (finitely) required by some program definition $(f\,\overline{y} = t) \in \mathcal{G}$. Thus, a $\mathcal{G}$-coconsistent type environment corresponds to a finite trace of an intersection typing for $\mathcal{G}$.

We prove that an instance of the HORS untypeability problem (does there exist a $\mathcal{G}$-coconsistent type environment Γ such that $\Gamma \vdash S :: q_0$?) reduces to MSL(ω) provability by adapting the rewriting algorithm to construct an intersection type environment instead of a canonical solved form.

Theorem 5.9. *HORS intersection untypeability reduces to MSL(ω) provability.*

We convert ground definitions from $\mathcal{G}$ to definite clauses $D_{\mathcal{G}}$ by wrapping both sides in a predicate P for each base type $q_P \in Q_\iota$. This gives us $(f\,\overline{y} = t) \in \mathcal{G}$ if, and only if, $(\forall \overline{y}.\, P\,t \Rightarrow P\,(f\,\overline{y})) \in D_{\mathcal{G}}$ for all $q_P \in Q_\iota$. Furthermore, the types of Σ_{con} are added to $D_{\mathcal{G}}$ as automaton clauses, giving rise to:

$$D_{\mathcal{G}} := \{\forall \overline{y}.\, P\,t \Rightarrow P\,(f\,\overline{y}) \mid (f\,\overline{y} = t) \in \mathcal{G} \wedge t : \iota \wedge q_P \in Q_\iota\} \cup \{\mathsf{toCl}(\tau)(c) \mid c :: \tau \in \Sigma_{\mathrm{con}}\}^{6}$$

For our example $\mathcal{G}_2$, $D_{\mathcal{G}_2}$ looks as follows:

$$\begin{aligned} D_{\mathcal{G}_2} := &\{(P\,(F\,\mathrm{s}) \Rightarrow P\,S),\ (\forall \varphi.\, P\,(\mathrm{c}\,(\varphi\,\mathrm{z})\,(F\,(B\,\varphi\,\varphi)) \Rightarrow P\,(F\,\varphi)) \mid P \in \{P_S, P_0, P_1, P_2, P_3\}\} \\ &\cup \{(\forall \varphi\,\psi\,x.\, P\,(\varphi\,(\psi\,x)) \Rightarrow P\,(B\,\varphi\,\psi\,x)) \mid P \in \{P_S, P_0, P_1, P_2, P_3\}\} \\ &\cup \{P_1\,\mathrm{z},\ P_2\,\mathrm{z},\ P_3\,\mathrm{z},\ (\forall x\,y.\, P_1\,x \wedge P_3\,x \wedge P_S\,y \Rightarrow P_S(\mathrm{c}\,x\,y))\} \cup \{\forall x.\, P_i\,x \Rightarrow P_{i+1}(\mathrm{s}\,x) \mid i \in [0..3]\} \end{aligned}$$

Definition 5.10 (Typing Algorithm). Given definite formula D, we construct a type environment $\Gamma^\infty(D) = \Gamma_{\mathcal{V}(D)}$ using the (inductive) MSL(ω) rewrite algorithm:

$$\begin{array}{r|c} (\forall \overline{y}.\, G \Rightarrow P\,(f\,\overline{y})) \in D & \Gamma \subseteq \Gamma^\infty(D) \\ V_\Gamma, \overline{y} \vdash G \rhd^* U & \overline{f :: \mathsf{fromCl}(\forall \overline{y}.\, U \Rightarrow P\,(f\,\overline{y})) \in \Gamma^\infty(D)} \end{array}$$

Proof sketch. For Theorem 5.9, it suffices to show the following:

$$\exists \mathcal{G}\text{-coconsistent } \Gamma.\, \Gamma \vdash t :: \tau \text{ if, and only if, } \mathcal{V}(D_{\mathcal{G}}) \vdash \mathsf{toCl}(\tau)(t) \rhd^* \mathsf{true}$$

We rely on the correspondence between typing and rewriting (Proposition 5.6), and the fact that $\Gamma^\infty(D_{\mathcal{G}}) = \Gamma_{\mathcal{V}(D_{\mathcal{G}})}$ is the largest $\mathcal{G}$-coconsistent environment. The claim follows from the restricted case where the LHS is $\vdash S :: q_0$. ☐

The reduction from HORS untypeability to MSL(ω) provability is clearly polynomial. This gives us a lower bound on the complexity of MSL(ω) in the order n of the program and highest arity k of any function symbol, based on the known complexity of HORS untypeability.

Theorem 5.11. *Deciding MSL(ω) provability is at least* $\exp_{n-1}(k|\Pi|)$*-hard.*

[6] We allow clauses $\mathsf{toCl}(\tau)(c)$ in $D_{\mathcal{G}}$ even though they are not generally definite clauses. Because they are automaton, we could instead directly include them in the first iteration of the rewriting algorithm.

5.3 Reducing MSL(ω) to HORS Intersection Typing

For the converse reduction, we reduce MSL(ω) directly to HORS cotrivial automaton model checking. We rely on an extension of HORS by Neatherway et al. [2012] called *HORS with cases* that enables us to use nondeterminism and a case-switch on the base types Q_ι (i.e. predicates Π), due to MSL(ω) lacking a clean separation between a state-agnostic rewrite system and a property automaton.

The MSL(ω) constants true and $\wedge$ are encoded as a HORS constant and variable, resp., making clause bodies monadic. Now a clause $(\forall\overline{y}.\, P'\, t \Rightarrow P\,(f\,\overline{y}))$ can be mapped to $f\,\overline{y}\,p = t\,p'$, where p is a constant corresponding to $P \in \Pi$.

MSL(ω)-to-HORS Transformation. We transform MSL(ω) constructor types γ to HORS types γ^+ by setting $\iota^+ := \iota \to \iota$ and $(\gamma_1 \to \gamma_2)^+ := \gamma_1^+ \to \gamma_2^+$. Then, the goal transformation to HORS bodies encodes true and $\wedge$ as:

$$\mathsf{true}^+ := \mathsf{true} \qquad (G \wedge H)^+ := G^+ \wedge H^+ \qquad (P\,s)^+ := s\,P$$

where, by some abuse, true and $\wedge$ on the RHS are a HORS constant and variable, resp.

MSL(ω)-as-HORS. Given an existential-free MSL(ω) definite formula D_0 and goal formula G_0 over Π and Σ, we construct the HORS $\mathcal{G} = \langle \Sigma_{\mathsf{con}}, \mathcal{N}, \mathcal{R}, S\rangle$ defined by:

$$\Sigma_{\mathsf{con}} := \{\mathsf{true} : \iota\} \cup \{P : \iota \to o \in \Pi\}$$
$$\mathcal{N} := \{f : \gamma^+ \mid f : \gamma \in \Sigma\} \cup \{S : \iota\} \cup \{\wedge : \iota \to \iota \to \iota\}$$
$$\mathcal{R} := \{f\,\overline{x}\,P = G^+ \mid (\forall\overline{x}.\, G \Rightarrow P\,(f\,\overline{x})) \in D_0\} \cup \{S = G_0^+\} \cup \{\wedge\,\mathsf{true}\,\mathsf{true} = \mathsf{true}\}$$

The Automaton. Because MSL(ω) does not have a clean separation between automaton and state-agnostic definitions, our automaton is trivial; it consists of a single state that accepts only the non-terminating/non-finished tree $\bot$. Intuitively, G_0^+ rewrites to true precisely if $D_0 \vDash G_0$.

Proposition 5.12. *$D_0 \vDash G_0$ if, and only if, $[\![\mathcal{G}]\!] \in \mathcal{L}(A)$*

This provides the missing link for the following theorem.

Theorem 5.13. *MSL(ω) provability and HORS (cotrivial) model checking are interreducible.*

6 IMPLEMENTATION & APPLICATION

We have implemented a decision procedure for MSL(ω) satisfiability in Haskell. Recall that the full HOMSL(ω) language reduces to this fragment, see Section 3. Our implementation incrementally rewrites clause bodies towards automaton form according to the rewrite relation from Section 4, using automaton clauses that have already been discovered. When a clause body is fully rewritten so a clause is automaton, further rewrites may become possible in other clause bodies, which are then reconsidered. It is therefore important to retain partially rewritten clauses. This procedure continues until no more automaton clauses can be produced and the set of clauses has been saturated.

To assess the viability of our MSL(ω) decision procedure for higher-order verification, we study the case of socket programming in Haskell, where higher-order constraints arise naturally from the use of continuations in effectful code.

We implemented a Haskell library which provides an abstract typeclass of socket effects, one instance of which generates constraints whose satisfiability implies correct usage of the sockets. This approach alleviates the need for a heavyweight analysis front-end by exploiting a common pattern of coding with effects. Typically, a program analysis front-end would take the source code of the program as input, internalise it as an AST and then walk over the AST to generate constraints; then a separate back-end would solve the constraints. Our approach instead allows us to use a typeclass instance to generate constraints directly, without any need to process the program AST. Doing this has practical advantages, because a standalone front-end usually requires regular updating to stay in sync with the syntax of a constantly evolving programming language.

Socket API. Socket APIs require the user to adhere to a strict protocol where only certain operations are permitted in each state. Correctly tracking the state of sockets throughout a program can be difficult, much like with lazy IO, and is often the source of bugs. Our implementation allows us to track not a single resource (e.g. file handler or socket) but countably many!

A socket can be in one of the following states: *Ready*, *Bound*, *Listen*, *Open*, *Closed*. The primitives modifying the state of a socket are summarised by the automaton in Figure 9. As these primitives operate in the IO-monad, we encode them in an explicit continuation-passing style such that each primitive takes a socket and a continuation as arguments. The socket and continuation are individuals (i.e. type ι), except in the case of `Accept` that also creates a new socket and thus has a continuation of type $\iota \rightarrow \iota$. Each state is encoded as a predicate, with an additional `Untracked` predicate whose meaning we explain below.

To account for the use of countably many resources, we employ a known trick that tracks the state of just one resource and non-deterministically chooses whether to track a newly created socket (unless one is already tracked) [Cook et al. 2007; Kobayashi 2013]. In our case, the fresh socket is either (1) labelled $\underline{s}$ and subsequent operations acting on it contribute to the overall state or (2) labelled $\underline{u}$ and is untracked. This approach suffices, because for each socket there exists a branch in which its behaviour is tracked and incorrect usage violates the overall state.

The MSL(ω) Clauses. Given a socket-manipulating Haskell program, the implementation computes a two-part MSL(ω) formula that models its behaviour: (1) a formula that captures the socket protocol and (2) a clausal representation of the semantics of the program. Intuitively, a predicate is satisfied by a program when that program's usage of the tracked socket violates the protocol for the corresponding state. The `Untracked` predicate is satisfied by a program that violates the protocol *for any* socket. When predicates are supplied with the tracked socket, the clauses encode the complement of the automaton from Figure 9; otherwise the state is unchanged. Furthermore, when sockets are created in the `Untracked` state, as described above, there are two clauses to account for whether the new socket is to be tracked or not. If a socket is created in any other state, it is simply untracked to prevent junk branches where multiple sockets are tracked with overlapping states.

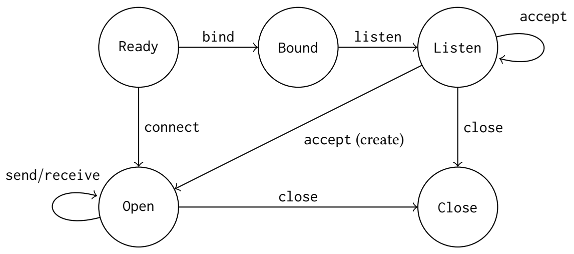

Fig. 9. Socket states and operations manipulating them

Extracting Constraints. Part (2) the two-part MSL(ω) formula is determined by the program. However, our approach does not need to process the Haskell source code and obtain an AST. Instead, we rely on a Haskell typeclass instance.

The socket primitives are provided as methods of a typeclass refining the monad class, which is further parametrised by the type of sockets. The instance of this typeclass for IO behaves in the usual manner, but we also supply an instance for analysis whose sockets are variable names and which merely accumulates the effects as raw syntax, ignoring any parameters other than the socket and continuation.

The advantage of this approach is that processing the source code of the program is not required, instead relying on normal, program evaluation to construct the constraints. One complication, however, is that we require any unbounded recursion to be made explicit to prevent an infinite evaluation of the program's definition. We provide a method capturing this as part of the typeclass interface. Recursion points are given a fresh name, to simulate as top-level definition of the program, and their bodies are analysed and attributed to those function names. Once a collection of recursive top-level definitions has been identified with an additional entry point we generate clauses unfolding each definition, without changing state, as in Theorem 5.9.

Furthermore, the program cannot arbitrarily depend on runtime data such as the value received by a socket; the effects of the program must be statically known. Branching code is not completely precluded however. Inspired by the *selective* extension of applicative functors that supports finite branching on runtime data, we add a branch combinator `branch` : `Bool` $\rightarrow fa \rightarrow fa \rightarrow fa$, encoded as multiple clauses that disregard the condition [Mokhov et al. 2019].

Examples. We tested our tool on example socket-manipulating programs taken from Stack Overflow (with values modified). When presenting these examples, we will use Haskell's do-notation and the "bind" operators ($\gg$, $\gg\!=$) for monadic actions as exposed in the user-interface, which are to be understood as syntactic-sugar for the underlying continuation passing-style. The first, with the original program on the left, violates the protocol on line 8 where it attempts to send a message over `soc` which is in the `Listening` state after line 5[7]. The tool was able to correctly detect the bug in 4.9ms, and accepted the correction (on the right) after 3.4ms.

```
1  main = do
2    soc ← socket
3    bind soc 1234
4     listen soc
5    x ← accept soc
6     forever $ do
7        receive x
8       send soc "Hi!"
```

```
1  main = do
2    soc ← socket
3    bind soc 1234
4     listen soc
5    x ← accept soc
6     forever $ do
7        receive x
8       send x "Hi!"
```

The following toy example makes use of our branching construct. When run in the IO monad, this will behave just like an if–then–else clause. For analysis, however, both branches are explored. The snippet initialises a socket and repeatedly receives a message until it is "close" when it closes the socket. In the version on the left, the loop continues regardless, thus attempting to receive from a closed socket. This implementation violates the protocol and is detected by our tool in 4.2ms. The fix, on the right, exits the loop once the socket is closed and was accepted in 5.1ms.

```
main = do
  soc ← socket
  bind soc 1234
   listen soc
  x ← accept soc
   forever $ do
     msg ← receive x
     branch (msg == "close")
        (close x)
        (pure ())
```

```
main = do
  soc ← socket
  bind soc 1234
   listen soc
  x ← accept soc
   fix $ \k → do
     msg ← receive x
     branch (msg == "close")
        (close x >> k)
        (k ())
```

[7]https://stackoverflow.com/q/62052147

7 CONCLUSION AND RELATED WORK

We have proposed new classes of constraints that are designed to capture the complex, higher-order behaviours of programs with first-class procedures. We developed their theory to (a) show decidability of the classes and (b) situate them with respect to higher-order program verification. We also described an implementation and its application to the verification of socket programming.

Complexity. Our reduction of intersection typeability to MSL(ω) satisfiability gives us $(n-1)$-EXPTIME hardness of MSL(ω) satisfiability. Furthermore, the reduction of order-n MSL(ω) to order-$(n+1)$ HORS with cases provides an $(n+1)$-EXPTIME upper bound, thanks to a result by Clairambault et al. [2018]. We derive this same naive upper bound directly from the decision procedure, where every application of (Step) or (Assm) on an order-n symbol has n-exponentially many candidate side conditions. Further study is required to obtain a tighter upper bound.

Related Work. We survey some of the work that is most closely related to our own.

Automata, Types, and Clauses. MSL was proposed independently by Weidenbach [1999] and Nielson et al. [2002] (as $\mathcal{H}1$), with Goubault-Larrecq [2005] providing the bridge between the two. Since then, it has been extended beyond Horn and with the addition of straight dismatching constraints in Teucke and Weidenbach [2017]. Recall that the solved form of clauses for the first-order $\mathcal{H}1$ fragment were named *automaton* clauses because of their shape, a connection that has also been made in Nagaya and Toyama [2002]; Weidenbach [1999]. This name is equally justified for our (higher-order) automaton clauses, since they, too, define finite tree automata [via intersection types, Broadbent and Kobayashi 2013]. The relationship between higher-order automata, types, and particular sets of clauses goes back to Frühwirth et al. [1997].

Set Constraints. Set constraints are a powerful language that has been very influential in program analysis [Aiken 1999]. They are known to be equivalent to the monadic class [Bachmair et al. 1993] and, therefore, have a very close connection with MSL. Higher-order set constraints have also been considered, defining sets of terms rather than higher-order predicates much like MSL(ω) [Goubault-Larrecq 2002]. Although the relationship between our constraints and those of *loc cit* is not well understood, we point out that their constraints are solvable in 2-NEXPTIME, whereas satisfiability in our class is $(n-1)$-EXPTIME hard.

HORS Model Checking. There is a strong connection between traditional higher-order model checking with higher-order recursion schemes [e.g. Kobayashi 2013] and MSL(ω) problems, as witnessed by their interreducibility. Many approaches to inferring and verifying types for higher-order recursion schemes have been considered, but the most closely related to our work is the saturation-based approach considered by Broadbent and Kobayashi [2013]. The main novelty of their algorithm is that typing constraints are propagated backwards starting from the final (unaccepted) states, rather than the forward from the target state. While backward propagation is analogous to goal-orientation search, attempting to derive clauses in order to rewrite the goal, their saturation-based approach is similar to our accumulation of automaton clauses in a bottom-up manner. Furthermore, follow-up work improved upon the efficiency of the saturation-based approach by representing intersection types as a type of binary decision diagrams that compactly describes a family of sets [Terao and Kobayashi 2014]. More work needs to be done to draw a detailed comparison between these algorithms and our own. Furthermore, as many of these algorithms are in their second or third generation, there will be possible optimisations that can be transferred to our own setting, in addition to novel optimisations that take advantage of our setting.

HFL Model Checking. Higher-order fixpoint logic, HFL, is a very expressive logic also designed as an appropriate language for program verification [Kobayashi 2021]. It is more expressive than higher-order (constrained) Horn clauses in general, and our fragment in particular, by supporting both the greatest and least fixpoint. This duality allows it to express liveness properties as well as safety properties. Furthermore, this logic allows for a background constraint theory. In the pure case, HFL is known to be decidable by reduction to intersection typing problem [Hosoi et al. 2019].

Refinement Type Checking and Constrained Horn Clauses. It was observed by Grebenshchikov et al. [2012] that a standard approach taken to solving refinement type inference problems, such as Jhala et al. [2011]; Terauchi [2010]; Unno and Kobayashi [2009], is essentially a reduction to constrained Horn clause solving. Although only first order, these systems of constraints are extremely expressive since they incorporate an arbitrary background theory, such as linear arithmetic or the theory of algebraic datatypes. Consequently, they are typically undecidable. Constrained Horn clauses were lifted to higher order by Cathcart Burn et al. [2017], and the theory further explored in Ong and Wagner [2019]. In a follow-up work the same authors identified a family of decidable fragments intended for applications in database aggregation [Cathcart Burn et al. 2021].

Uniform Proofs and Logic Programming. The formulation of our fragments and the proof system that underlies them follows the elegant presentation in the work of Miller and his collaborators, such as Miller and Nadathur [2012]; Miller et al. [1991]. In particular, one can recognise their *fohc*, *hohc*, and *hohh* as the underlying formalisms behind our $\mathsf{MSL}(1)$ clauses, $\mathsf{HOMSL}(\omega)$ clauses, and higher-order automaton clauses respectively. Of course, we could have presented our fragments of HOL in a more traditional format for automated reasoning (e.g. with clauses as multisets of literals), but we consider the compositional characterisation that is characteristic of Miller's work essential for a clear exposition once we have to deal with the combination of nested clauses (in the sense of hereditary Harrop) and higher-order constructs.

Constructive Logic and 'Horn Clauses as Types'. Over a series of papers, Fu, Komendantskaya, and co-authors have presented a comprehensive analysis of Horn clauses and resolution according to the propositions-as-types tradition [Farka 2020; Fu and Komendantskaya 2015, 2017; Fu et al. 2016]. Like our work, they cast resolution as a form of rewriting, studying a number of different variations on the standard approach that have been motivated by the desire to capture computations with infinite data. Using Howard's System **H** [Howard 1980], they give a type-theoretic semantics to each form of resolution, and this allows for a more meaningful notion of soundness and completeness than the traditional method using Herbrand models. Since it is in the propositions-as-types tradition, their work views a Horn clause as the type of its proofs. By contrast, we view an automaton clause with a single free variable x as a type inhabited by the terms that satisfy the clause (when substituted for x). Consequently, we do not make use of the constructive content of the resolution proofs themselves, but rather view resolution simply as a mechanism for generating a new clause from two given clauses – i.e. a way to infer new types.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the Engineering and Physical Sciences Research Council (EP/T006595/1) and the National Centre for Cyber Security via the UK Research Institute in Verified Trustworthy Software Systems. We are also very grateful for the help of the reviewers in making the paper more clear and accurate, and for suggesting related work.

REFERENCES

Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35, 2 (1999), 79–111. https://doi.org/10.1016/S0167-6423(99)00007-6

Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. 1993. Set Constraints are the Monadic Class. In *Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE, Montreal, Canada, 75–83. https://doi.org/10.1109/LICS.1993.287598

Christopher Broadbent and Naoki Kobayashi. 2013. Saturation-based model checking of higher-order recursion schemes. In *Computer Science Logic 2013 (CSL 2013)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CSL.2013.129

Florian Bruse, Martin Lange, and Étienne Lozes. 2021. The Complexity of Model-Checking Tail-Recursive Higher-Order Fixpoint Logic. *Fundam. Informaticae* 178, 1-2 (2021), 1–30. https://doi.org/10.3233/FI-2021-1996

Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. 2017. Higher-Order Constrained Horn Clauses for Verification. *Proc. ACM Program. Lang.* 2, POPL, Article 11 (Dec 2017), 28 pages. https://doi.org/10.1145/3158099

Toby Cathcart Burn, C.-H. Luke Ong, Steven J. Ramsay, and Dominik Wagner. 2021. Initial Limit Datalog: a New Extensible Class of Decidable Constrained Horn Clauses. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS52264.2021.9470527

Angelos Charalambidis, Christos Nomikos, and Panos Rondogiannis. 2019. The Expressive Power of Higher-Order Datalog. *Theory and Practice of Logic Programming* 19, 5-6 (2019), 925–940. https://doi.org/10.1017/S1471068419000279

Pierre Clairambault, Charles Grellois, and Andrzej S. Murawski. 2018. Linearity in Higher-Order Recursion Schemes. *Proc. ACM Program. Lang.* 2, POPL, Article 39 (Dec 2018), 29 pages. https://doi.org/10.1145/3158127

Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y Vardi. 2007. Proving that programs eventually do something good. *ACM SIGPLAN Notices* 42, 1 (2007), 265–276. https://doi.org/10.1145/1190216.1190257

František Farka. 2020. *Proof-Relevant Resolution: the Foundations of Constructive Proof Automation.* Ph. D. Dissertation. Heriot-Watt University, UK.

Thom Frühwirth, Moshe Vardi, and Eyal Yardeni. 1997. Logic Programs as Types for Logic Programs. *Proceedings – Symposium on Logic in Computer Science* (12 1997). https://doi.org/10.1109/LICS.1991.151654

Peng Fu and Ekaterina Komendantskaya. 2015. A Type-Theoretic Approach to Resolution. In *Logic-Based Program Synthesis and Transformation*, Moreno Falaschi (Ed.). Springer International Publishing, Cham, 91–106. https://doi.org/10.1007/978-3-319-27436-2_6

Peng Fu and Ekaterina Komendantskaya. 2017. Operational Semantics of Resolution and Productivity in Horn Clause Logic. *Form. Asp. Comput.* 29, 3 (may 2017), 453–474. https://doi.org/10.1007/s00165-016-0403-1

Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. 2016. Proof Relevant Corecursive Resolution. In *Functional and Logic Programming*, Oleg Kiselyov and Andy King (Eds.). Springer International Publishing, Cham, 126–143. https://doi.org/10.1007/978-3-319-29604-3_9

Jean Goubault-Larrecq. 2002. Higher-Order Positive Set Constraints. In *Computer Science Logic*, Julian Bradfield (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 473–489. https://doi.org/10.1007/3-540-45793-3_32

Jean Goubault-Larrecq. 2005. Deciding H1 by resolution. *Inform. Process. Lett.* 95, 3 (2005), 401–408. https://doi.org/10.1016/j.ipl.2005.04.007

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 405–416. https://doi.org/10.1145/2254064.2254112

Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. 2008. Collapsible Pushdown Automata and Recursion Schemes. In *Logic in Computer Science, LICS'08*. IEEE Computer Society, 452–461.

Haskell.org. 2013. *Iteratee I/O: The problem with lazy IO.* Retrieved 7 July 2022 from https://wiki.haskell.org/Iteratee_I/O#The_problem_with_lazy_I.2FO

Youkichi Hosoi, Naoki Kobayashi, and Takeshi Tsukada. 2019. A type-based HFL model checking algorithm. In *Asian Symposium on Programming Languages and Systems*. Springer, 136–155. https://doi.org/10.1007/978-3-030-34175-6_8

William Howard. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Seldin and R. J. Hindley (Eds.). Academic Press.

Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying Functional Programs Using Abstract Interpreters. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 470–485. https://doi.org/10.1007/978-3-642-22110-1_38

Jerome. Jochems. 2020. *Higher-order constrained Horn clauses for higher-order program verification.* Ph. D. Dissertation. Oxford University, UK.

Jerome Jochems, Eddie Jones, and Steven Ramsay. 2022. Higher-Order MSL Horn Constraints. https://doi.org/10.48550/ARXIV.2210.14649

Naoki Kobayashi. 2009. Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs. *SIGPLAN Not.* 44, 1 (Jan 2009), 416–428. https://doi.org/10.1145/1594834.1480933

Naoki Kobayashi. 2013. Model checking higher-order programs. *Journal of the ACM (JACM)* 60, 3 (2013), 1–62. https://doi.org/10.1145/2487241.2487246

Naoki Kobayashi. 2021. An Overview of the HFL Model Checking Project. *arXiv preprint arXiv:2109.04629* (2021). https://doi.org/10.4204/EPTCS.344.1

Naoki Kobayashi and C.-H. Luke Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *Logic in Computer Science, LICS 2009*. IEEE Computer Society, 179–188. https://doi.org/10.1109/LICS.2009.29

Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic*. Cambridge University Press. https://doi.org/10.1017/CBO9781139021326

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic* 51, 1 (1991), 125–157. https://doi.org/10.1016/0168-0072(91)90068-W

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. 2019. Selective applicative functors. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29. https://doi.org/10.1145/3341694

David E. Muller and Paul E. Schupp. 1987. Alternating automata on infinite trees. *Theoretical Computer Science* 54, 2 (1987), 267–276. https://doi.org/10.1016/0304-3975(87)90133-2

Takashi Nagaya and Yoshihito Toyama. 2002. Decidability for Left-Linear Growing Term Rewriting Systems. *Information and Computation* 178, 2 (2002), 499–514. https://doi.org/10.1006/inco.2002.3157

Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. 2012. A Traversal-Based Algorithm for Higher-Order Model Checking. *SIGPLAN Not.* 47, 9 (Sep 2012), 353–364. https://doi.org/10.1145/2398856.2364578

Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. 2002. Normalizable Horn clauses, strongly recognizable relations, and Spi. In *International Static Analysis Symposium*. Springer, 20–35. https://doi.org/10.1007/3-540-45789-5_5

C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. 81–90. https://doi.org/10.1109/LICS.2006.38

C.-H. Luke Ong and Dominik Wagner. 2019. HoCHC: A Refutationally Complete and Semantically Invariant System of Higher-order Logic Modulo Theories. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–14. https://doi.org/10.1109/LICS.2019.8785784

Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. 2014. A Type-Directed Abstraction Refinement Approach to Higher-Order Model Checking. In *Principles of Programming Languages, POPL'14*. ACM, 61–72. https://doi.org/10.1145/2535838.2535873

Jakob Rehof and Paweł Urzyczyn. 2011. Finite Combinatory Logic with Intersection Types. In *Typed Lambda Calculi and Applications*, C.-H. Luke Ong (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–183. https://doi.org/10.1007/978-3-642-21691-6_15

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 159–169. https://doi.org/10.1145/1375581.1375602

Sylvain Salvati and Igor Walukiewicz. 2016. Simply typed fixpoint calculus and collapsible pushdown automata. *Math. Struct. Comput. Sci.* 26, 7 (2016), 1304–1350. https://doi.org/10.1017/S0960129514000590

Taku Terao and Naoki Kobayashi. 2014. A ZDD-based efficient higher-order model checking algorithm. In *Asian Symposium on Programming Languages and Systems*. Springer, 354–371. https://doi.org/10.1007/978-3-319-12736-1_19

Tachio Terauchi. 2010. Dependent types from counterexamples. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 119–130. https://doi.org/10.1145/1706299.1706315

Andreas Teucke and Christoph Weidenbach. 2017. Decidability of the Monadic Shallow Linear First-Order Fragment with Straight Dismatching Constraints. In *Automated Deduction – CADE 26*, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 202–219. https://doi.org/10.1007/978-3-319-63046-5_13

Hiroshi Unno and Naoki Kobayashi. 2009. Dependent type inference with interpolants. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*. 277–288. https://doi.org/10.1145/1599410.1599445

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 48–61. https://doi.org/10.1145/2784731.2784745

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 209–228. https://doi.org/10.1007/978-3-

642-37036-6_13

Mahesh Viswanathan and Ramesh Viswanathan. 2004. A Higher Order Modal Fixed Point Logic. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 512–528. https://doi.org/10.1007/978-3-540-28644-8_33

Christoph Weidenbach. 1999. Towards an Automatic Analysis of Security Protocols in First-Order Logic. In *Automated Deduction — CADE-16*. Springer Berlin Heidelberg, Berlin, Heidelberg, 314–328. https://doi.org/10.1007/3-540-48660-7_29

He Zhu and Suresh Jagannathan. 2013. Compositional and Lightweight Dependent Type Inference for ML. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. 295–314. https://doi.org/10.1007/978-3-642-35873-9_19

Received 2022-07-07; accepted 2022-11-07