Higher-Order Constrained Horn Clauses for Verification

TOBY CATHCART BURN, University of Oxford, UK
C.-H. LUKE ONG, University of Oxford, UK
STEVEN J. RAMSAY*, University of Bristol, UK

Motivated by applications in automated verification of higher-order functional programs, we develop a notion of constrained Horn clauses in higher-order logic and a decision problem concerning their satisfiability. We show that, although satisfiable systems of higher-order clauses do not generally have least models, there is a notion of canonical model obtained through a reduction to a problem concerning a kind of monotone logic program. Following work in higher-order program verification, we develop a refinement type system in order to reason about and automate the search for models. This provides a sound but incomplete method for solving the decision problem. Finally, we show that there is a sense in which we can use refinement types to express properties of terms whilst staying within the higher-order constrained Horn clause framework.

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

Additional Key Words and Phrases: higher-order program verification, constrained Horn clauses, refinement types

ACM Reference Format:
Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. 2018. Higher-Order Constrained Horn Clauses for Verification. *Proc. ACM Program. Lang.* 2, POPL, Article 11 (January 2018), 28 pages. https://doi.org/10.1145/3158099

1 INTRODUCTION

There is evidence to suggest that many first-order program verification problems can be framed as solvability problems for systems of constrained Horn clauses, see Beyene, Popeea, and Rybalchenko [2013], Bjørner, McMillan, and Rybalchenko [2013b] and Bjørner, Gurfinkel, McMillan, and Rybalchenko [2015]. These systems consist of Horn clauses of first-order logic containing constraints expressed in some suitable background theory. This makes the study of these systems particularly worthwhile since they provide a purely logical basis on which to develop techniques for first-order program verification. For example, results on the development of highly efficient constrained Horn clause solvers (such as those reported by Grebenshchikov, Gupta, Lopes, Popeea, and Rybalchenko [2012], Hoder, Bjørner, and de Moura [2011] and Gurfinkel, Kahsai, Komuravelli, and Navas [2015]) can be exploited by a large number of program verification tools, each of which offloads some complex task (invariant finding is a typical example) to a solver by framing it in terms of constrained Horn clauses.

*This work was completed whilst the author was a Research Assistant at the University of Oxford.

Authors' addresses: Toby Cathcart Burn, University of Oxford, UK, toby.cathcartburn@cs.ox.ac.uk; C.-H. Luke Ong, University of Oxford, UK, luke.ong@cs.ox.ac.uk; Steven J. Ramsay, University of Bristol, UK, steven.ramsay@bristol.ac.uk.

2475-1421/2018/1-ART11
https://doi.org/10.1145/3158099

This paper concerns automated verification of higher-order, functional programs. Whilst there are approaches to the verification of functional programs in which constrained Horn clause solving plays an important role, there is inevitably a mismatch between the higher-order nature of the program and the first-order logic in which the Horn clauses are expressed, and this must be addressed in some intelligent way by the program verifier. For example, in recent work on refinement types (see e.g. Rondon, Kawaguchi, and Jhala [2008], Vazou, Bakst, and Jhala [2015] and Unno, Terauchi, and Kobayashi [2013]), a type system is used to reduce the problem of finding an invariant for the higher-order program to finding a number of first-order invariants of the ground-type data at certain program points. This latter problem can often be expressed as a system of constrained Horn clauses. When that system is solvable, the first-order invariants obtained can be composed in the type system to yield a higher-order invariant for the program (expressed as a type assignment).

In this paper we introduce *higher-order constrained Horn clauses*, an extension of the notion of constrained Horn clause to higher-order logic. This gives us a language in which we can express Horn constraints over higher-order functions (actually, higher-order relations), which is not possible to do directly with only first-order constrained Horn clauses. Although the definition is natural, it is not immediate that it is suitable for automated verification, so we go on to show three important results in that direction, namely: the existence of canonical solutions, the applicability of existing techniques to automated solving and the expressibility of program properties of higher type.

Let us elaborate on each of these and illustrate our motivation more concretely by discussing a particular example. Consider the following higher-order program:

$$\begin{array}{l}\mathsf{let}\ \mathit{add}\ x\ y = x + y\\ \mathsf{let\ rec}\ \mathit{iter}\ f\ s\ n = \mathsf{if}\ n \leq 0\ \mathsf{then}\ s\ \mathsf{else}\ f\ n\ (\mathit{iter}\ f\ s\ (n-1))\\ \mathsf{in}\ \lambda n.\ \mathsf{assert}\ (n \leq \mathit{iter}\ \mathit{add}\ 0\ n)\end{array}$$

The term *iter add 0 n* occurring in the last line of the program is the sum of the integers from 1 to n in case n is non-negative and is 0 otherwise. Let us say that the program is *safe* just in case the assertion is never violated, i.e. the summation is not smaller than n.

To verify safety, we must find an invariant that implies the required property. For our purposes, an invariant will be an over-approximation of the input-output graphs of the functions defined in the program. If we can find an over-approximation of the graph of the function *iter add 0* which does not contain any pair (n, m) with $n > m$, then we can be certain that the guard on the assertion is never violated. Hence, we seek a set of pairs of natural numbers relating n to m at least whenever *iter add* 0 n evaluates to m and which has no intersection with $>$.

The idea is to express the problem of finding such a program invariant logically, as a satisfiability problem for the following set of higher-order constrained Horn clauses:

$$\begin{array}{l}\forall xyz.\ z = x + y \Rightarrow \mathit{Add}\ x\ y\ z\\ \forall f\,snm.\ n \leq 0 \wedge m = s \Rightarrow \mathit{Iter}\ f\ s\ n\ m\\ \forall f\,snm.\ n > 0 \wedge \left(\exists p.\ \mathit{Iter}\ f\ s\ (n-1)\ p \wedge f\ n\ p\ m\right) \Rightarrow \mathit{Iter}\ f\ s\ n\ m\\ \forall nm.\ \mathit{Iter}\ \mathit{Add}\ 0\ n\ m \Rightarrow n \leq m\end{array}$$

The clauses constrain the variables $\mathit{Add} : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to o$ and Iter: $(\mathsf{int} \to \mathsf{int} \to \mathsf{int} \to o) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to o$ with respect to the theory of integer linear arithmetic, so a model is just an assignment of particular relations[1] to these variables that satisfies the formulas. The first clause constrains *Add* to be an over-approximation of the graph of the addition function *add*: whenever $z = x + y$, we at least know that x, y and z are related by *Add*. The second and third constrain *Iter* to be an over-approximation of the graph of the iteration combinator *iter*. Observe that the two

[1]Throughout this paper we will speak of relations but work with their characteristic functions, which are propositional (Boolean-valued) functions, using o for the sort of propositions.

branches of the conditional in the program appear as two clauses, expressing over-approximations in which the third input n is at most or greater than 0 respectively. The final clause ensures that, taken together, these over-approximations are yet precise enough that they exclude the possibility of relating an input n of *iter add* 0 to a smaller output m. In other words, a model of these formulas *is* an invariant, in the sense we have described above, and thus constitutes a witness to the safety of the program.

Notice that what we are describing in this example is a *compositional approach*, in which an over-approximation *Iter Add* $0 : \mathsf{int} \to \mathsf{int} \to o$ to the graph of the function *iter add* $0 : \mathsf{int} \to \mathsf{int}$ is constructed from over-approximations *Add* and *Iter* of the graphs of the functions *add* and *iter*. Consequently, where *iter* was a higher-order function, *Iter* is a higher-order relation taking a ternary relation on integers as input, and the quantification $\forall f$ is over all such ternary relations f. However, for the purposes of this paper, the details of how one obtains a system of higher-order constrained Horn clauses are not actually relevant, since we here study properties of such systems independently of how they arise.

Existence of canonical solutions. A set of higher-order constrained Horn clauses may have many models or none (consider that there are many invariants that can prove a safety property or none in case it is unprovable). One model of the above set of clauses is the following assignment of relations (expressed in higher-order logic):

$$\mathit{Add} \mapsto \lambda x\, y\, z.\, z = x + y \qquad \mathit{Iter} \mapsto \lambda f\, s\, n\, m.\, (\forall x\, y\, z.\, f\ x\ y\ z \Rightarrow 0 < x \Rightarrow y < z) \wedge 0 \leq s \Rightarrow n \leq m$$

Notice that this represents quite a large model (a coarse invariant). For example, under this assignment the relation described by *Iter Add* (-1) relates every pair of integers n and m. In the case of first-order constrained Horn clauses over the theory of integer linear arithmetic, if a set of clauses has a model, then it has a least model[2], and this least model property is at the heart of many of the applications of Horn clauses in practice. If we consider the use of constrained Horn clauses in verification, a key component of the design of many successful algorithms for solving systems of clauses (and program invariant finding more generally) is the notion of approximation or abstraction. However, to speak of approximation presupposes there is something to approximate. For program verifiers there is, for example, the set of reachable states or the set of traces of the program, and for first-order constrained Horn clause solvers there is the least model.

In contrast, in Section 4 we show that satisfiable systems of higher-order constrained Horn clauses do not necessarily have least models. The problem, which has also been observed for pure (without constraint theory) higher-order Horn clauses by Charalambidis, Handjopoulos, Rondogiannis, and Wadge [2013], can be attributed to the use of unrestricted quantification over relations. By restricting the semantics, so that interpretations range only over monotone relations (monotone propositional functions), we ensure that systems do have least solutions, but at the cost of abandoning the standard (semantics of) higher-order logic. The monotone semantics is natural but, for the purpose of specifying constraint systems, it can be unintuitive. For example, consider the formula $\forall x.\, (\exists yz.\, x\ y \wedge y\ z) \Rightarrow P\ x$ which constrains $P : ((\mathsf{int} \to o) \to o) \to o$ so that it is at least true of all non-empty sets of non-empty sets of integers [3]. In the monotone semantics, this formula is guaranteed to have a least model, but inside that model P is not true of the set $\{\{0\}\}$.

Ideally, we would like to be able to *specify* constraint systems using the standard semantics of higher-order logic, but *solve* (build solvers for) systems in the monotone semantics. In fact, we show that this is possible: we construct a pair of adjoint mappings with which the solutions to

[2]In general, one can say that for each model of the background theory, a satisfiable set of clauses has a least satisfying valuation.

[3]Viewing relations of sort $\mathsf{int} \to o$ as sets of integers.

the former can be mapped to solutions of the latter and vice versa. This allows us to reduce the problem of solving a system of constraints in the standard semantics to the problem of solving a system in the monotone semantics. Monotonicity and the fact that satisfiable monotone systems have canonical solutions are key to the rest of the work in the paper.

Applicability of existing techniques to automated solving. Many of the techniques developed originally for the automation of first-order program verification transfer well to first-order constrained Horn clause solving. Hence, to construct automated solvers for systems of higher-order clauses, we look to existing work on higher-order program verification. In automated verification for functional programs, one of the most successful techniques of recent years has been based on refinement type inference, such as reported by Rondon et al. [2008], Kobayashi, Sato, and Unno [2011], Vazou et al. [2015] and Zhu and Jagannathan [2013]. The power of the approach comes from its ability to lift rich first-order theories over data to higher types using subtyping and the dependent product.

In Section 5, we develop a refinement type system for higher-order constrained Horn clauses, in which types are assigned to the free relation variables that are being solved for. The idea is that a valid type assignment is a syntactic representation of a model. For example, the model discussed previously can be represented by the type assignment Γ_I:

$$\mathit{Add}\colon x{:}\mathsf{int} \to y{:}\mathsf{int} \to z{:}\mathsf{int} \to o\langle z = x + y\rangle$$
$$\mathit{Iter}\colon (x{:}\mathsf{int} \to y{:}\mathsf{int} \to z{:}\mathsf{int} \to o\langle 0 < x \Rightarrow y < z\rangle) \to s{:}\mathsf{int} \to n{:}\mathsf{int} \to m{:}\mathsf{int} \to o\langle 0 \leq s \Rightarrow n \leq m\rangle$$

The correspondence hinges on the definition of refinements $o\langle\varphi\rangle$ of the propositional sort o, which are parametrised by a first-order constraint formula φ describing an upper bound on the truth of any inhabitant. The dependent product and integer types are interpreted standardly, so that the first type above can be read as the set of all ternary relations on integers x, y and z that are false whenever z is not $x + y$.

The system is designed so that its soundness allows one to conclude that a given first-order constraint formula can be used to approximate a given higher-order formula[4]. Given a formula G, from the derivability of the judgement $\Gamma \vdash G : o\langle\varphi\rangle$ it follows that $G \Rightarrow \varphi$ in those interpretations of the relational variables that satisfy Γ. For example, the judgement

$$\Gamma_I,\ n\colon \mathsf{int},\ m\colon \mathsf{int} \vdash \mathit{Iter}\ \mathit{Add}\ 0\ n\ m : o\langle n \leq m\rangle$$

is derivable, from which we may conclude that $n \leq m$ is a sound abstraction of $\mathit{Iter}\ \mathit{Add}\ 0\ n\ m$ in any interpretation of *Iter* and *Add* that satisfies Γ_I. This is a powerful assertion for automated reasoning because the formula φ in refinement type $o\langle\varphi\rangle$ is a simple first-order constraint formula (typically belonging to a decidable theory) whereas the formula G in the subject is a complicated higher-order formula, possibly containing relational variables whose meanings are a function of the whole system. By adapting machinery developed for refinement type inference of functional programs, we obtain a sound (but incomplete) procedure for solving systems of higher-order constrained Horn clauses. An implementation shows the method to be feasible.

Expressibility of program properties of higher type. We say that a property is of higher type if it is a property of a higher-order function. It is possible to do whole-program verification in a higher-order setting using only properties of first-order type, because a complete program typically has a first-order type like $\mathsf{int} \to \mathsf{int}$. However, it is also natural to want to specify properties of higher types, for example properties of higher-order functions of type $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$. Even when the ultimate goal is one of whole-program verification, being able to verify properties of higher types is advantageous because it can allow a large analysis to be broken down into smaller components according to the structure of the program.

[4]Technically, the subjects of the type system are not all higher-order formulas but only the so-called *goal* formulas.

However, the kinds of higher-type properties expressible by higher-order constrained Horn clauses is not immediately clear. Therefore, we conclude Section 5 by showing that it is possible to state at least those properties that can be defined using refinement types, since their complements are expressible using goal terms.

The rest of the paper is structured as follows. In Section 2 we fix our presentation of higher-order logic and the notion of higher-order constrained Horn clause is made precise in Section 3 along with the associated definition of solvability. Section 4 introduces monotone logic programs, which are better suited to automated reasoning, and shows that solvability of higher-order constrained Horn clause problems can be reduced to solvability of these programs. This class of logic programs forms the basis for the refinement type system defined in Section 5, which yields a sound but incomplete method for showing solvability through type inference. An implementation of the method is also discussed in this section, which concludes by discussing the definability of higher-type properties defined by refinement types. Finally, in Section 6, we discuss related work and draw conclusions in Section 7. Full proofs are included in the anonymous supplementary materials.

2 HIGHER-ORDER LOGIC

We will work in a presentation of higher-order logic as a typed lambda calculus.

Sorts. Given a sort ι of individuals (for example int), and a sort o of propositions, the general *sorts* are just the simple types that can be built using the arrow: $\sigma ::= \iota \mid o \mid \sigma_1 \to \sigma_2$. The *order* of a sort σ, written $\mathrm{order}(\sigma)$, is defined as follows:

$$\mathrm{order}(\iota) = 1 \quad \mathrm{order}(o) = 1 \qquad \mathrm{order}(\sigma_1 \to \sigma_2) = \max(\mathrm{order}(\sigma_1) + 1, \mathrm{order}(\sigma_2))$$

Note: we follow the convention from logic of regarding base sorts to be of order 1. Consequently, we will consider, for example, an int variable to be of order 1 and an $(\mathsf{int} \to o) \to \mathsf{int}$ function to be of order 3.

Terms. The terms that we consider are just terms of an applied lambda calculus. We will write variables generally using x, y, z, or X, Y, Z when we want to emphasise that they are of higher-order sorts.

$$M, N \quad ::= \quad x \mid c \mid M\,N \mid \lambda x{:}\sigma.\, M$$

in which c is a constant. We assume that application associates to the left and the scope of the abstraction extends as far to the right as possible. We identify terms up to α-equivalence.

Sorting. A *sort environment*, typically Δ, is a finite sequence of pairs $x : \sigma$, all of whose subjects are required to be distinct. We assume, for each constant c, a given sort assignment σ_c. Then sorting rules for terms are, as standard, associated with the judgement $\Delta \vdash s : \sigma$ defined by:

$$\text{(SCst)}\ \frac{}{\Delta \vdash c : \sigma_c} \qquad \text{(SVar)}\ \frac{}{\Delta_1, x : \sigma, \Delta_2 \vdash x : \sigma}$$

$$\text{(SApp)}\ \frac{\Delta \vdash s : \sigma_1 \to \sigma_2 \qquad \Delta \vdash t : \sigma_1}{\Delta \vdash s\ t : \sigma_2} \qquad \text{(SAbs)}\ \frac{\Delta, x : \sigma_1 \vdash s : \sigma_2}{\Delta \vdash \lambda x.\, s : \sigma_1 \to \sigma_2}\ x \notin \mathrm{dom}(\Delta)$$

Given a sorted term $\Delta \vdash M : \sigma$, we say that a variable occurrence x in M is of *order* k just if the unique subderivation $\Delta' \vdash x : \sigma'$ rooted at this occurrence has σ' of order k. We say that a sorted term $\Delta \vdash M : \sigma$ is of *order* k just if k is the largest order of any of the variables occurring in M.

Formulas. Given a first-order signature Σ specifying a collection of base sorts and sorted constants, we can consider higher-type *formulas* over Σ, by considering terms whose constant symbols are either drawn from the signature Σ or are a member of the following set LSym of logical constant symbols:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{true}, \mathsf{false} & : & o & \neg & : & o \to o \\ \wedge, \vee, \Rightarrow & : & o \to o \to o & \forall_\sigma, \exists_\sigma & : & (\sigma \to o) \to o \end{array}$$

As usual, we write $\exists_\sigma(\lambda x{:}\sigma.\, M)$ more compactly as $\exists x{:}\sigma.\, M$ and define the set of *formulas* to be just the well-sorted terms of sort o. In the context of formulas, it is worthwhile to recognise the subset of *relational sorts*, typically ρ, which have the sort o in tail position and whose higher-order subsorts are also relational. Formally:

$$\rho ::= o \mid \iota \to \rho \mid \rho \to \rho$$

Since formulas are just terms, the notion of order carries over without modification.

Interpretation. Let A be a Σ-structure. In particular, we assume that A assigns a non-empty set A_ι to each of the base sorts $\iota \in B$ and the lattice $\mathbb{2} = \{0 \leq 1\}$ to the sort o. We define the *full sort frame* over A by induction on the sort:

$$\mathcal{S}[\![\iota]\!] \coloneqq A_\iota \qquad \mathcal{S}[\![o]\!] \coloneqq \mathbb{2} \qquad \mathcal{S}[\![\sigma_1 \to \sigma_2]\!] \coloneqq \mathcal{S}[\![\sigma_1]\!] \Rightarrow \mathcal{S}[\![\sigma_2]\!]$$

where $X \Rightarrow Y$ is the full set-theoretic function space between sets X and Y. The lattice $\mathbb{2}$ supports the following functions:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{or}(b_1)(b_2) & = & \max\{b_1, b_2\} & \mathsf{not}(b) & = & 1 - b \\ \mathsf{and}(b_1)(b_2) & = & \min\{b_1, b_2\} & \mathsf{implies}(b_1)(b_2) & = & \mathsf{or}(\mathsf{not}(b_1))(b_2) \\ \mathsf{exists}_\sigma(f) & = & \max\{f(v) \mid v \in [\![\sigma]\!]\} & \mathsf{forall}_\sigma(f) & = & \mathsf{not}(\mathsf{exists}_\sigma(\mathsf{not} \circ f)) \end{array}$$

We extend the order on $\mathbb{2}$ to order the set $\mathcal{S}[\![\rho]\!]$ of all relations of a given sort ρ pointwise, defining the order $\subseteq_\rho$ inductively on the structure of ρ:

- For all $b_1, b_2 \in \mathcal{S}[\![o]\!]$: if $b_1 \leq b_2$ then $b_1 \subseteq_o b_2$
- For all $r_1, r_2 \in \mathcal{S}[\![\iota \to \rho]\!]$: if, for all $n \in \mathcal{S}[\![\iota]\!]$, $r_1(n) \subseteq_\rho r_2(n)$, then $r_1 \subseteq_{\iota \to \rho} r_2$.
- For all $r_1, r_2 \in \mathcal{S}[\![\rho_1 \to \rho_2]\!]$: if, for all $s \in \mathcal{S}[\![\rho_1]\!]$, $r_1(s) \subseteq_{\rho_2} r_2(s)$, then $r_1 \subseteq_{\rho_1 \to \rho_2} r_2$.

This ordering determines a complete lattice structure on each $\mathcal{S}[\![\rho]\!]$, we will denote the (pointwise) join and meet by $\bigcup_\rho$ and $\bigcap_\rho$ respectively. To aid readability, we will typically omit subscripts.

We interpret a sort environment Δ by the indexed product: $\mathcal{S}[\![\Delta]\!] \coloneqq \Pi x \in \mathrm{dom}(\Delta).\mathcal{S}[\![\Delta(x)]\!]$, that is, the set of all functions on $\mathrm{dom}(\Delta)$ that map x to an element of $\mathcal{S}[\![\Delta(x)]\!]$; these functions, typically α, are called *valuations*. We similarly order $\mathcal{S}[\![\Delta]\!]$ pointwise, with $f_1 \subseteq_\Delta f_2$ just if, for all $x{:}\rho \in \Delta$, $f_1(x) \subseteq_\rho f_2(x)$; thus determining a complete lattice structure.

For the purpose of interpreting formulas, we extend the structure A to interpret the symbols from LSym according to the functions given above. The interpretation of a term $\Delta \vdash M : \sigma$ is a function $\mathcal{S}[\![\Delta \vdash M : \sigma]\!]$ (we leave A implicit) that belongs to the set $\mathcal{S}[\![\Delta]\!] \Rightarrow \mathcal{S}[\![\sigma]\!]$, and which is defined by the following equations.

$$\begin{array}{rcl} \mathcal{S}[\![\Delta \vdash x : \sigma]\!](\alpha) & = & \alpha(x) \\ \mathcal{S}[\![\Delta \vdash c : \sigma]\!](\alpha) & = & c^A \\ \mathcal{S}[\![\Delta \vdash M\,N : \sigma_2]\!](\alpha) & = & \mathcal{S}[\![\Delta \vdash M : \sigma_1 \to \sigma_2]\!](\alpha)\big(\mathcal{S}[\![\Delta \vdash N : \sigma_1]\!](\alpha)\big) \\ \mathcal{S}[\![\Delta \vdash \lambda x : \sigma_1.\, M : \sigma_1 \to \sigma_2]\!](\alpha) & = & \lambda v \in \mathcal{S}[\![\sigma_1]\!].\, \mathcal{S}[\![\Delta, x : \sigma_1 \vdash M : \sigma_2]\!](\alpha[x \mapsto v]) \end{array}$$

We will write $\mathcal{S}[\![M]\!]$ for $\mathcal{S}[\![\Delta \vdash M : \sigma]\!]$ whenever the judgement $\Delta \vdash M : \sigma$ is clear from the context.

Satisfaction. For a Σ-structure A, a formula $\Delta \vdash M : o$ and a valuation $\alpha \in \mathcal{S}[\![\Delta]\!]$, we say that $\langle A, \alpha\rangle$ *satisfies* M and write $A, \alpha \vDash M$ just if $\mathcal{S}[\![\Delta \vdash M : o]\!](\alpha) = 1$. We define entailment $M \vDash N$ between two formulas M and N in terms of satisfaction as usual.

3 HIGHER-ORDER CONSTRAINED HORN CLAUSES

We introduce a notion of constrained Horn clauses in higher-order logic.

Constraint language. Assume a fixed, first-order language over a first-order signature Σ, consisting of: distinguished subsets of first-order terms *Tm* and first-order formulas ($\varphi \in$) *Fm*, and a first-order theory $\mathcal{T}h$ in which to interpret those formulas. We refer to this first-order language as the *constraint language*, and $\mathcal{T}h$ as the *background theory*.

Atoms and constraints. An *atom* is an applicative formula of shape $X\ M_1 \cdots M_k$ in which X is a relational variable and each M_i is a term. A *constraint*, φ, is just a formula from the constraint language. For technical convenience, we assume that atoms do not contain any constants (including logical constants), and constraints do not contain any relational variables.

Constrained Horn clauses. Fix a sorting Δ of relational variables. The *constrained goal formulas* over Δ, typically G, and the *constrained definite formulas* over Δ, typically D, are the subset of all formulas defined by induction:

$$\begin{array}{lcl} G & ::= & M \mid \varphi \mid G \wedge G \mid G \vee G \mid \exists x{:}\sigma.\, G \\ D & ::= & \mathsf{true} \mid \forall x{:}\sigma.\, D \mid D \wedge D \mid G \Rightarrow X\,\overline{x} \end{array}$$

in which σ is either the sort of individuals ι or a relational sort ρ, M is an atom[5], φ a constraint and, in the last alternative, X is required to be a relational symbol inside $\mathrm{dom}(\Delta)$ and $\overline{x} = x_1 \cdots x_n$ a sequence of pairwise distinct variables. It will often be convenient to view a constrained definite formula equivalently as a conjunction of (constrained) *definite clauses*, which are those definite formulas with shape: $\forall \overline{x}.\, G \Rightarrow X\,\overline{x}$.

Remark 1. *Our class of constrained definite formulas resembles the* definitional *fragment of Wadge [1991], due to the restrictions on the shape of $X\,\overline{x}$ occurring in the head of definite clauses. However, the formalism discussed in loc. cit., which was intended as a programming language, also restricted the existential quantifiers that could occur inside goal formulas and did not consider any notion of underlying constraint language.*

Problem. A (higher-order) *Constrained Horn Clause Problem* is given by a tuple $\langle \Delta, D, G\rangle$ in which:

- Δ is a sorting of relational variables.
- $\Delta \vdash D : o$ is a constrained definite formula over Δ.
- $\Delta \vdash G : o$ is a constrained goal formula over Δ.

The problem is of *order* k if k is the largest order of the bound variables that occur in D or G. We say that such a problem is *solvable* just if, for all models A of the background theory $\mathcal{T}h$, there exists a valuation α of the variables in Δ such that $A, \alpha \vDash D$, and yet $A, \alpha \nvDash G$.

Remark 2. *The presentation of the problem follows some of the literature for the use of first-order Horn clauses in verification. The system of higher-order constrained Horn clauses is partitioned into two, distinguishing the definite clauses as a single definite formula and presenting the* negation *of non-definite clauses as a single goal formula, which is required to be* refuted *by valuations. This better reflects the distinction between the program and the property to be proven. Furthermore, solvability is defined in a way that allows for incompleteness in the background theory to be used to express*

[5]We do not require the head variable of M to be in $\mathrm{dom}(\Delta)$.

under-specification of programming language features (for example, because they are difficult to reason about precisely).

Example 3.1. Let us place the motivating system of clauses from the introduction formally into the framework. To that end, let us fix the quantifier free fragment of integer linear arithmetic (ZLA) as the underlying constraint language. The sorting Δ of relational variables (the unknowns to be solved for) are given by:

$$\begin{array}{rl} \mathit{Add}\colon & \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to o \\ \mathit{Iter}\colon & (\mathsf{int} \to \mathsf{int} \to \mathsf{int} \to o) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to o \end{array}$$

The higher-order constrained definite formula D consists of a conjunction of the following three constrained definite clauses:

$$\begin{array}{l} \forall x\, y\, z.\, z = x + y \Rightarrow \mathit{Add}\ x\ y\ z \\ \forall f\, s\, n\, m.\, n \leq 0 \wedge m = 0 \Rightarrow \mathit{Iter}\ f\ s\ n\ m \\ \forall f\, s\, n\, m.\, (\exists p.\, n > 0 \wedge \mathit{Iter}\ f\ s\ (n-1)\ p \wedge f\ n\ p\ m) \Rightarrow \mathit{Iter}\ f\ s\ n\ m \end{array}$$

Finally, the clause $\forall nm.\, \mathit{Iter}\ \mathit{Add}\ 0\ n\ m \Rightarrow n \leq m$ expressing the property of interest is negated to give goal $G = \exists n\, m.\, \mathit{Iter}\ \mathit{Add}\ 0\ n\ m \wedge m < n$. This problem is solvable. Being a complete theory, ZLA has one model up to isomorphism and, with respect to this model, the valuation given in the introduction satisfies D but refutes G.

We believe that the generalisation of constrained horn clauses to higher orders is very natural. However, our principal motivation in its study is the possibility of obtaining interesting applications in higher-order program verification (analogous to those in first-order program verification with first-order constrained Horn clauses). We interpret higher-order program verification in its broadest sense, encompassing not just purely functional languages but, more generally, problems in which satisfaction of a property depends upon an analysis of higher-order control flow. For example, an early application of first-order constrained Horn clauses in the very successful constraint logic programming (CLP) paradigm of Jaffar and Maher [1994] was the analysis of circuit designs by Heintze, Michaylov, and Stuckey [1992], for which systems of clauses were felt to be particularly suitable since they give a succinct, declarative specification of the analyses. The relative advantages of circuit design description using higher-order combinator libraries or specification languages based on higher-order programming, such as that of Bjesse, Claessen, Sheeran, and Singh [1998], are well documented, and systems of higher-order of constrained Horn clauses would therefore be a natural setting in which to verify the properties of such designs.

4 MONOTONE MODELS

One of the attractive features of *first-order* constrained Horn clauses is that, for any given choice of interpretation of the background theory, every definite formula (set of definite clauses) possesses a unique, least model. Consequently, it follows that there is a solution to a first-order Horn clause problem $\langle \Delta, D, G \rangle$ iff for each model of the background theory, the least model of D refutes G. This reformulation of the problem is of great practical benefit because it allows for the design of algorithms that, at least conceptually, exploit the canonicity. For example, at the heart of the design of many successful algorithms for first-order Horn clause solving (and program invariant finding more generally) is the notion of approximation or abstraction. However, to speak of approximation presupposes there is something to approximate. For program verifiers there is, for example, the set of reachable states or the set of traces of the program, and for first-order constrained Horn clause solvers there is the least model.

The fact that first-order definite formulas possess a least model is paid for by restrictions placed on the syntax. By forbidding negative logical connectives in goal formulas, it can be guaranteed that the unknown relation symbols in any definite clause occur positively exactly once, and hence obtaining the consequences of a given formula is a monotone operation. We have made the same syntactic restrictions in our definition of higher-order constrained Horn formulas, but we do not obtain the same outcome.

Theorem 4.1. *Higher-order constrained definite formulas do not necessarily possess least models.*

Proof. Consider the sorting Δ_{one} of relational variables P: $((\mathsf{one} \to o) \to o) \to o$ and $Q : \mathsf{one} \to o$ and the definite formula D_{one}:

$$\Delta \vdash \forall x.\, x\, Q \Rightarrow P\, x : o$$

over a finite constraint language consisting of the sort one of individuals and no functions, relations or constants of any kind. The language is interpreted in the background theory axiomatised by the sentence $\forall xy.x = y$, so that all models consist of a single individual $\mathcal{S}[\![\mathsf{one}]\!] = \{\star\}$. Let us use $\mathbf{0}$ to denote the mapping $\star \mapsto 0$ and $\mathbf{1}$ denote the mapping $\star \mapsto 1$, both of which together comprise the set $\mathcal{S}[\![\mathsf{one} \to o]\!]$; and let us name the elements of $\mathcal{S}[\![(\mathsf{one} \to o) \to o]\!]$ as follows:

$$\mathsf{a} := \begin{array}{ccc} \mathbf{0} & \mapsto & 0 \\ \mathbf{1} & \mapsto & 1 \end{array} \qquad \mathsf{b} := \begin{array}{ccc} \mathbf{0} & \mapsto & 0 \\ \mathbf{1} & \mapsto & 0 \end{array} \qquad \mathsf{c} := \begin{array}{ccc} \mathbf{0} & \mapsto & 1 \\ \mathbf{1} & \mapsto & 1 \end{array} \qquad \mathsf{d} := \begin{array}{ccc} \mathbf{0} & \mapsto & 1 \\ \mathbf{1} & \mapsto & 0 \end{array}$$

Then we can describe minimal models α_1 and α_2 by the following equations:

$$\begin{array}{cc} \alpha_1(Q) = \mathbf{0} & \\ \alpha_1(P)(\mathbf{a}) = 0 & \alpha_1(P)(\mathbf{b}) = 0 \\ \alpha_1(P)(\mathbf{c}) = 1 & \alpha_1(P)(\mathbf{d}) = 1 \end{array} \qquad \begin{array}{cc} \alpha_2(Q) = \mathbf{1} & \\ \alpha_2(P)(\mathbf{a}) = 1 & \alpha_2(P)(\mathbf{b}) = 0 \\ \alpha_2(P)(\mathbf{c}) = 1 & \alpha_2(P)(\mathbf{d}) = 0 \end{array}$$

It is easy to verify that there are no models smaller than these and yet they are unrelated, so there is no least model. □

A similar observation has been made in the pure (without constraint theory) setting by Charalambidis et al. [2013].

Some consideration of the proof of this theorem leads to the observation that, despite an embargo on negative logical connectives in goal formulas, it may still be the case that unknown relation symbols occur negatively in goal formulas (and hence may occur positively more than once in a definite clause). For example, consider the definite clause from above, namely: $\forall x.\, x\, Q \Rightarrow P\, x$. Whether or not Q can be said to occur positively in the goal formula $x\, Q$ depends on the action of x. If we consider the subterm $\mathcal{S}[\![x\, Q]\!]$ as a function of x and Q, then it is monotone in Q only when the function assigned to x is itself monotone. By contrast, if x is assigned an antitone function, as is the case when x takes on the value $\mathbf{d}$, then $\mathcal{S}[\![x\, Q]\!]$ will be antitone in Q; for example $\alpha_1(Q) \subseteq \alpha_2(Q)$ but $\mathcal{S}[\![x\, Q]\!](\alpha_2[x \mapsto \mathbf{d}]) \subseteq \mathcal{S}[\![x\, Q]\!](\alpha_1[x \mapsto \mathbf{d}])$.

4.1 Logic Programs

As we show in the following section, by restricting to an interpretation in which every function is monotone (in the logical order) we can obtain a problem in which there is a notion of least solution[6]. However, in doing so it seems that we sacrifice some of the logical purity of our original problem: if the universe of our interpretation contains only monotone functions, it does not include the function implies and so it becomes unclear how to interpret definite formulas. Consequently, it requires a new definition of what it means to be a model of a formula. Rather, the version of the

[6]Recall that *least* (respectively *monotone*) here refers to smallest in (preservation of) the *logical* order, i.e. with respect to inclusion of relations.

problem we obtain by restricting to a monotone interpretation is much more closely related to work on the extensional semantics of higher-order *logic programs*, such as that of Wadge [1991] and Charalambidis et al. [2013], which emphasises the role of Horn clauses as definitions of rules. Hence, we present the monotone restriction in those terms.

Goal terms. The class of well-sorted *goal terms* $\Delta \vdash G : \rho$ is given by the sorting judgements defined by the rules below, in which c is one of $\wedge$, $\vee$ or $\exists_\sigma$ and here, and throughout the rules, σ is required to stand for either the sort of individuals ι or otherwise some relational sort. It is easily verified that the constrained goal formulas are a propositional sorted subset of the goal terms. From now on we shall use G, H and K to stand for arbitrary goal *terms* and disambiguate as necessary.

$$\text{(GCst)}\ \frac{}{\Delta \vdash c : \rho_c}\ c \in \{\wedge, \vee, \exists_\iota\} \cup \{\exists_\rho \mid \rho\} \qquad \text{(GVar)}\ \frac{}{\Delta_1, x : \rho, \Delta_2 \vdash x : \rho}$$

$$\text{(GConstr)}\ \frac{}{\Delta \vdash \varphi : o}\ \Delta \vdash \varphi : o \in Fm \qquad \text{(GAbs)}\ \frac{\Delta, x : \sigma \vdash G : \rho}{\Delta \vdash \lambda x.\, G : \sigma \to \rho}\ x \notin \mathrm{dom}(\Delta)$$

$$\text{(GAppl)}\ \frac{\Delta \vdash G : \iota \to \rho}{\Delta \vdash G\, N : \rho}\ \Delta \vdash N : \iota \in Tm \qquad \text{(GAppR)}\ \frac{\Delta \vdash G : \rho_1 \to \rho_2 \qquad \Delta \vdash H : \rho_1}{\Delta \vdash G\, H : \rho_2}$$

Logic programs. A higher-order, constrained *logic program*, P, over a sort environment $\Delta = x_1 : \rho_1, \ldots, x_m : \rho_m$ is just a finite system of (mutual) recursive definitions of shape:

$$x_1 : \rho_1 = G_1, \quad \ldots, \quad x_m : \rho_m = G_m$$

Such a program is well sorted when, for each $1 \leq i \leq m$, $\Delta \vdash G_i : \rho_i$. Since each x_i is distinct, we will sometimes regard a program P as a finite map from variables to terms, defined so that $P(x_i) = G_i$. We will write $\vdash P : \Delta$ to abbreviate that P is a well-sorted program over Δ.

Standard interpretation. Logic programs can be interpreted in the standard semantics by interpreting the right-hand sides of the equations using the term semantics given in Section 2. The program P itself then gives rise to the functional $T^{\mathcal{S}}_{P:\Delta} : \mathcal{S}[\![\Delta]\!] \Rightarrow \mathcal{S}[\![\Delta]\!]$, sometimes called the *one-step consequence operator* in the literature on the semantics of logic programming, which is defined by: $T^{\mathcal{S}}_{P:\Delta}(\alpha)(x) = \mathcal{S}[\![\Delta \vdash P(x) : \Delta(x)]\!](\alpha)$.

The logic program of a definite formula. Every definite formula D gives rise to a logic program, which is obtained by collapsing clauses that share the same head $X\ \overline{x}$ by taking the disjunction of their bodies, and viewing the resulting expression as a recursive definition of X. The formulation as logic program is more convenient in two ways. First, it is a more natural object to which to assign a monotone interpretation since we have eliminated implication, which does not act monotonically in its first argument, in favour of definitional equality. Second, looking ahead to Section 5, the syntactic structure of logic programs allows for a more transparent definition of a type system.

To that end, fix a definite formula $\Delta \vdash D : o$. We assume, without loss of generality[7], that D has the shape:

$$\forall \overline{x_{r_1}}.\, G_1 \Rightarrow X_{r_1}\ \overline{x_{r_1}} \quad \wedge \quad \cdots \quad \wedge \quad \forall \overline{x_{r_\ell}}.\, G_\ell \Rightarrow X_{r_\ell}\ \overline{x_{r_\ell}}$$

[7] Observe that such a shape can always be obtained by applying standard logical equivalences.

$$\begin{array}{rcl}
\mathcal{M}[\![\Delta \vdash x : \rho]\!](\alpha) & = & \alpha(x) \\
\mathcal{M}[\![\Delta \vdash \varphi : o]\!](\alpha) & = & \mathcal{S}[\![\Delta \vdash \varphi : o]\!](\alpha) \\
\mathcal{M}[\![\Delta \vdash G\,H : \rho_2]\!](\alpha) & = & \mathcal{M}[\![\Delta \vdash G : \rho_1 \to \rho_2]\!](\alpha)(\mathcal{M}[\![\Delta \vdash H : \sigma_1]\!](\alpha)) \\
\mathcal{M}[\![\Delta \vdash G\,N : \rho]\!](\alpha) & = & \mathcal{M}[\![\Delta \vdash G : \iota \to \rho]\!](\alpha)(\mathcal{S}[\![\Delta \vdash N : \iota]\!](\alpha)) \\
\mathcal{M}[\![\Delta \vdash \lambda x : \sigma.\, G : \sigma]\!](\alpha) & = & \lambda x' \in \mathcal{M}[\![\sigma]\!].\, \mathcal{M}[\![\Delta, x : \sigma \vdash G : \sigma]\!](\alpha[x \mapsto x']) \\
\mathcal{M}[\![\Delta \vdash \wedge : o \to o \to o]\!](\alpha) & = & \mathsf{and} \\
\mathcal{M}[\![\Delta \vdash \vee : o \to o \to o]\!](\alpha) & = & \mathsf{or} \\
\mathcal{M}[\![\Delta \vdash \exists_\sigma : (\sigma \to o) \to o]\!](\alpha) & = & \mathsf{mexists}_\sigma
\end{array}$$

Fig. 1. Monotone semantics of goal terms.

over a sort environment $\Delta = \{X_1\colon \rho_1, \ldots, X_k\colon \rho_k\}$, i.e. $\{1, \ldots, k\} = \{r_1, \ldots, r_\ell\}$. We construct a program over Δ, called the *logic program of* D and denoted P_D, as follows:

$$X_1 = \lambda\overline{x_1}.\, G'_1, \quad \ldots, \quad X_k = \lambda\overline{x_k}.\, G'_k$$

where $G'_j = \bigvee\{G_i \mid r_i = j\}$. Note that $\{G_i \mid r_i = j\}$ are exactly the bodies of all the definite clauses in D whose heads are X_j. The fact that $\vdash P_D : \Delta$ follows immediately from the well-sortedness of D.

Example 4.2. The definite formula component of the Horn clause problem from Example 3.1 is transformed into the following logic program P:

$$Add = \lambda x\, y\, z.\, z = x + y \qquad Iter = \lambda f\, s\, n\, m.\, (n \leq 0 \wedge m = s) \vee (\exists p.\, 0 < n \wedge Iter\, f\, s\, (n-1)\, p \wedge f\, n\, p\, m)$$

Characterisation. If we were to consider only the standard interpretation then the foregoing development of logic programs would have limited usefulness. As is well known at first-order, the definite formula and the program derived from it essentially define the same class of objects.

Lemma 4.3. *For definite formula D, the prefixed points of $T^{\mathcal{S}}_{P_D}$ are exactly the models of D.*

In contrast to the first-order case, it follows that $T^{\mathcal{S}}_{P_D}$ does not have a least (pre-)fixed point[8]. Indeed, for reasons already outlined, this functional is not generally monotone. However, it will play an important role in Section 4.3.

4.2 Monotone Semantics

The advantage of logic programs is that they have a natural, monotone interpretation.

Monotone sort frame. We start from the interpretation of the background theory A, regarding A_ι as a discrete poset. We then define the *monotone sort frame* over A by induction:

$$\mathcal{M}[\![\iota]\!] \coloneqq A_\iota \qquad \mathcal{M}[\![o]\!] \coloneqq \mathbb{2} \qquad \mathcal{M}[\![\sigma_1 \to \sigma_2]\!] \coloneqq \mathcal{M}[\![\sigma_1]\!] \Rightarrow_m \mathcal{M}[\![\sigma_2]\!]$$

where $X \Rightarrow_m Y$ is the monotone function space between posets X and Y, i.e. the set of all functions $f \in X \Rightarrow Y$ that have the property that $x_1 \leq x_2$ implies $f(x_1) \leq f(x_2)$. It is easy to verify that this function space is itself a poset with respect to the pointwise ordering. Of course, in case X is discrete poset A_ι, this coincides with the full function space. We extend the lattice structure of $\mathbb{2}$ to all relations $\mathcal{M}[\![\rho]\!]$, analogously to the case of the full function space (and we reuse the same notation since there will be no confusion); and we similarly define $\mathcal{M}[\![\Delta]\!] \coloneqq \Pi x \in \mathrm{dom}(\Delta).\, \mathcal{M}[\![\Delta(x)]\!]$.

It is worth considering the implications of monotonicity in the special case of relations, i.e. propositional functions. A relation r is an element of $X_1 \Rightarrow_m \cdots \Rightarrow_m X_k \Rightarrow_m \mathbb{2}$ just if it is *upward closed*: whenever r is true of $x_1, \ldots, x_k$ ($x_i \in X_i$), and $x'_1, \ldots, x'_k$ ($x'_i \in X_i$) has the property that

[8] In this paper we use the term *prefixed point* to refer to those x for which $f(x) \leq x$

$x_i \subseteq x'_i$, then r must also be true of $x'_1, \ldots, x'_k$. In particular, when $r \in X \Rightarrow_m 2$, then r can be thought of as an upward closed set of elements of X.

Monotone interpretation. The interpretation of goal terms is defined in Figure 1. As for the standard interpretation, we assume a fixed interpretation A of the background theory, which is left implicit in the notation. Whilst the standard interpretation of the positive logical constants for conjunction and disjunction will suffice, the interpretation of existential quantification needs to be relativised to the monotone setting: $\text{mexists}_\sigma(r) = max\{r(d) \mid d \in \mathcal{M}[\![\sigma]\!]\}$. As for standard semantics, we write $\mathcal{M}[\![G]\!]$ for $\mathcal{M}[\![\Delta \vdash G : \rho]\!]$, whenever the judgement $\Delta \vdash G : \rho$ is clear from the context.

Since the implication function implies is not monotone (in its first argument), definite formulas are not interpretable in a monotone frame. However, it is possible to interpret logic programs. To that end, we define the functional $T^{\mathcal{M}}_{P:\Delta}$ on semantic environments by: $T^{\mathcal{M}}_{P:\Delta}(\alpha)(x) = \mathcal{M}[\![\Delta \vdash P(x) : \Delta(x)]\!](\alpha)$. In analogy with the Horn clause problem, we call a prefixed point of $T^{\mathcal{M}}_{P:\Delta}$ a *model* of the program P. This construction preserves the logical order.

Lemma 4.4. $\mathcal{M}[\![\Delta \vdash G : \rho]\!] \in \mathcal{M}[\![\Delta]\!] \Rightarrow_m \mathcal{M}[\![\rho]\!]$ *and* $T^{\mathcal{M}}_{P:\Delta} \in \mathcal{M}[\![\Delta]\!] \Rightarrow_m \mathcal{M}[\![\Delta]\!]$.

Proof. Immediately follows from the fact that mexists, and and or are monotone and all the constructions are monotone combinations. □

It follows from the Knaster-Tarski theorem that, unlike the functional arising from the standard interpretation, the monotone functional $T^{\mathcal{M}}_{P:\Delta}$ has a least fixed point, which we shall write $\mu T^{\mathcal{M}}_{P:\Delta}$. Consequently, logic programs $\vdash P : \Delta$ have a canonical monotone interpretation, $[\![\vdash P : \Delta]\!]$, which we define as $\mu T^{\mathcal{M}}_{P:\Delta}$.

Monotone problem. By analogy with the first-order case, we are led to the following monotone version of the higher-order constrained Horn clause problem. A *Monotone Logic Program Safety Problem* (more often just *monotone problem*) is a triple (Δ, P, G) consisting of a sorting of relational variables Δ, a logic program $\vdash P : \Delta$ and a goal $\Delta \vdash G : o$. The problem is solvable just if, for all models of the background theory, there is a prefixed point α of $T^{\mathcal{M}}_{P:\Delta}$ such that $\mathcal{M}[\![G]\!](\alpha) = 0$.

4.3 Canonical Embedding

In the monotone problem we have obtained a notion of safety problem that admits a least solution. Due to the monotonicity of $\mathcal{M}[\![G]\!]$, there is a prefix point witnessing solvability iff the least prefix point is such a witness, i.e. iff $\mathcal{M}[\![G]\!](\mathcal{M}[\![P]\!]) = 0$. This clears the way for our algorithmic work in Section 5, which consists of apparatus in which to construct sound approximations of $\mathcal{M}[\![P]\!]$. However, the price we have had to pay seems severe, since we have all but abandoned our original problem definition.

The monotone logic program safety problem lacks the logical purity of the higher-order constrained Horn clause problem, which is stated crisply in terms of the standard interpretation of higher-order logic and the usual notion of models of formulas. In the context of program verification, the monotone problem appears quite natural, but if we look further afield, to e.g. traditional applications of constrained Horn clauses in constraint satisfaction, it seems a little awkward. For example, the significance of allowing only monotone solutions seems unclear if one is looking to state a scheduling problem for a haulage company or a packing problem for a factory.

Ideally, we would like to *specify* constraint systems using the standard Horn clause problem, with its clean logical semantics, but *solve* instances of the monotone problem, which is easier to analyse, due to monotonicity and the existence of canonical models. In fact, we shall show that

this is possible: every solution to the Horn clause problem $\langle \Delta, D, G\rangle$ determines a solution to the monotone problem $\langle \Delta, P_D, G\rangle$ and vice versa (Theorem 4.10).

Transferring solutions. Let us begin by considering what a mapping between solutions of $\langle \Delta, P_D, G\rangle$ and solutions of $\langle \Delta, D, G\rangle$ would look like. In both cases, a solution is a model: the former is a mapping from variables to monotone relations and the latter is a mapping from (the same) variables to arbitrary relations.

At first glance, it might appear that one can transfer models of P_D straightforwardly to models of D, because monotone relations are, in particular, relations. However, the situation is a little more difficult. Although the solution space of the Horn clause problem is larger, more is required of a valuation in order to qualify as a model because the constraints of the Horn clause problem, which involve universal quantification over all relations, are more difficult to satisfy than the equations of the monotone problem, which involve (implicitly) quantification over only the monotone relations.

To see this concretely, it is useful to consider the simpler case in which all relations are required to be unary, i.e. ρ is of shape $(\cdots((\iota \to o) \to o) \to \cdots) \to o$. In the unary case, we can think of a relation simply as describing a set of objects, where those objects may themselves be sets of objects. For example, $\mathcal{S}[\![(\iota \to o) \to o]\!]$ describes the collection of all sets of sets of individuals. On the other hand, the constraint on monotonicity of relations has the consequence that, if we think of $\mathcal{M}[\![(\iota \to o) \to o]\!]$ as describing a collection of sets, it is the collection only consisting of those sets of sets of individuals that are upward closed. That is, a set s is in $\mathcal{M}[\![(\iota \to o) \to o]\!]$ just if, whenever a set of individuals t is in s and $t \subseteq u$ then u is also in s. In general, we can think of $\mathcal{S}[\![\sigma \to o]\!]$ as the collection of all sets of objects from $\mathcal{S}[\![\sigma]\!]$, and $\mathcal{M}[\![\sigma \to o]\!]$ as the collection of hereditarily upward-closed sets. Now consider the logic program $P = \lambda x.\,\mathsf{true}$, in which P is of sort $((\mathsf{int} \to o) \to o) \to o$. One model of this program is to take for P the set of *all* upward-closed sets of sets of integers, which is a relation in $\mathcal{M}[\![((\mathsf{int} \to o) \to o) \to o]\!]$. However, the set of all upward-closed sets of sets of integers is not a model of the corresponding formula $\forall x.\,\mathsf{true} \Rightarrow P\ x$ in the standard semantics, because it does not contain, for example, the set $\{\{0\}\}$ which is not upward closed (i.e. its characteristic function is not a *monotone* Boolean-valued function), yet the universal quantification requires it.

So, although there is a canonical inclusion of $\mathcal{M}[\![\rho]\!]$ into $\mathcal{S}[\![\rho]\!]$, it does not extend to map models of P_D to models of D in general. If we return to thinking of the elements of $((\mathsf{int} \to o) \to o) \to o$ formally as Boolean-valued functions, the inclusion described above is mapping the monotone function $r \in \mathcal{M}[\![(\mathsf{int} \to o) \to o)]\!] \Rightarrow_m 2$, which satisfies $r(t) = 1$ for all $t \in \mathcal{M}[\![(\mathsf{int} \to o) \to o]\!]$, to the function $J(r) \in \mathcal{S}[\![(\mathsf{int} \to o) \to o]\!] \Rightarrow 2$, which satisfies, for all $t \in \mathcal{S}[\![(\mathsf{int} \to o) \to o)]\!]$:

$$J(r)(t) = \begin{cases} r(t) & \text{if } t \in \mathcal{M}[\![(\mathsf{int} \to o) \to o)]\!] \\ 0 & \text{otherwise} \end{cases}$$

In other words, it lifts a function whose domain consists only of hereditarily monotone relations to a function whose domain consists of all relations simply by mapping non-hereditarily monotone inputs to 0. We could equally well consider the dual, in which all such inputs were mapped to 1, but the image of the mapping would typically not refute the goal G because the models so constructed are too large.

This counterexample suggests that we require a mapping of monotone relations $r \in \mathcal{M}[\![((\mathsf{int} \to o) \to o) \to o]\!]$ to standard relations $J(r) \in \mathcal{S}[\![((\mathsf{int} \to o) \to o) \to o]\!]$ that is a little more sophisticated in the action of $J(r)$ on inputs that are not hereditarily monotone. Instead of mapping all such inputs to 0 or all such inputs to 1 we shall determine the value of $J(r)$ on some non-monotone input $t \in \mathcal{S}[\![(\mathsf{int} \to o) \to o]\!]$ by considering the value of r on a monotone input $U(t) \in \mathcal{M}[\![(\mathsf{int} \to o) \to o]\!]$ which is somehow *close* to t. In fact there are two canonical choices of hereditarily monotone

relations close to a given relation t, which are obtained as, respectively, the largest monotone relation included in t and the smallest monotone relation in which t is included. We will describe the situation in general using Galois connection.

Galois connection. A pair of functions $f : P \to Q$ and $g : Q \to P$ between partial orders P and Q is a *Galois connection* just if, for all $x \in P$ and $y \in Q$: $f(x) \leq y$ iff $x \leq g(y)$. In such a situation we write $f \dashv g$ and f is said to be the *left adjoint* of g, and g the *right adjoint* of f. First, it is easy to verify that if $f \dashv g$ then f and g are monotone.

Proposition 4.5. *Given a pair of monotone maps $f : P \to Q$ and $g : Q \to P$, the following are equivalent:*

(1) The pair (f, g) is a Galois connection.
(2) $f \circ g \leq 1_Q$ and $g \circ f \leq 1_P$.
(3) For all $x \in P$, $\inf \{y \in Q \mid x \leq g(y)\}$ is defined and equal to $f(x)$; and for all $y \in Q$, $\sup \{x \in P \mid f(x) \leq y\}$ is defined and equal to $g(y)$.

Further, if any one of the above conditions holds, then

(4) f preserves all existing suprema, and g preserves all existing infima.
(5) $f = f \circ g \circ f$ and $g = g \circ f \circ g$.

To see the Proposition, just view the pair (f, g) as functors on categories [Mac Lane 1971]; then they forms a Galois connection exactly when they are an adjunction pair. The following facts are easy to verify:

(i) If P is a complete lattice and $f : P \to Q$ preserves all joins, then f is a left adjoint.
(ii) If Q is a complete lattice and $g : Q \to P$ preserves all meets, then g is an right adjoint.
(iii) If $f_1 : P \to Q$, $g_1 : Q \to P$, $f_2 : Q \to R$ and $g_2 : R \to Q$ with $f_1 \dashv g_1$ and $f_2 \dashv g_2$ then it follows that $f_1 \circ f_2 \dashv g_1 \circ g_2$, is a Galois connection between partial orders P and R.
(iv) If $f_1 : P_1 \to Q_1$, $g_1 : Q_1 \to P_1$, $f_2 : P_2 \to Q_2$ and $g_2 : Q_2 \to P_2$ with $f_1 \dashv g_1$ and $f_2 \dashv g_2$ then it follows that the pair of functions $f : [P_1 \Rightarrow_m P_2] \to [Q_1 \Rightarrow_m Q_2]$ and $g : [Q_1 \Rightarrow_m Q_2] \to [P_1 \Rightarrow_m P_2]$, defined by: $f(h) = f_1 \circ h \circ g_2$ and $g(k) = g_1 \circ k \circ f_2$ is a Galois connection $f \dashv g$ between the corresponding monotone function spaces (ordered pointwise).

Facts (i) and (ii) are just the Adjoint Functor Theorem (see e.g. Mac Lane [1971]) specialised to the case of preorders. Notice that a special case of (iv) is the construction of a Galois connection $f \dashv g$ between the (full) function spaces $A \Rightarrow P$ and $A \Rightarrow Q$ (with pointwise order) for fixed set A and partial orders P and Q, whenever there is a Galois connection $f_2 \dashv g_2$ between P and Q. This is because there is always a trivial Galois connection $\mathrm{id} \dashv \mathrm{id}$ on any set A by viewing it as a discrete partial order.

Embedding the monotone relations. For general ρ, every complete lattice of monotone relations $\mathcal{M}[\![\rho]\!]$ can be embedded in the complete lattice of all relations $\mathcal{S}[\![\rho]\!]$ in the following two ways.

$$\mathcal{S}[\![\rho]\!] \underset{\mathsf{L}_\rho}{\overset{\mathsf{I}_\rho}{\leftrightarrows}} \mathcal{M}[\![\rho]\!] \underset{\mathsf{J}_\rho}{\overset{\mathsf{U}_\rho}{\leftrightarrows}} \mathcal{S}[\![\rho]\!] \tag{1}$$

We define the family of right adjoints I_ρ and the family of left adjoints J_ρ, by induction on the sort ρ. In the definition, L_ρ is the uniquely determined left adjoint of I_ρ and U_ρ is the uniquely determined right adjoint of J_ρ.

$$\begin{array}{rclcrcl} \mathsf{I}_o(b) & = & b & \qquad & \mathsf{J}_o(b) & = & b \\ \mathsf{I}_{\iota \to \rho}(r) & = & \mathsf{I}_\rho \circ r & & \mathsf{J}_{\iota \to \rho}(r) & = & \mathsf{J}_\rho \circ r \\ \mathsf{I}_{\rho_1 \to \rho_2}(r) & = & \mathsf{I}_{\rho_2} \circ r \circ \mathsf{L}_{\rho_1} & & \mathsf{J}_{\rho_1 \to \rho_2}(r) & = & \mathsf{J}_{\rho_2} \circ r \circ \mathsf{U}_{\rho_1} \end{array}$$

We briefly discuss this definition before verifying its correctness. It is worth observing that, rather than defining $\mathsf{I}_{\rho_1\to\rho_2}$ and $\mathsf{J}_{\rho_1\to\rho_2}$ using the induced left and right adjoints at ρ_1, we could have given the definition explicitly (recalling Galois connection properties (ii) and (iii)) by:

$$\mathsf{I}_{\rho_1\to\rho_2}(r)(s) = \mathsf{I}_{\rho_2}\big(r(\bigcap\{t \mid s \subseteq \mathsf{I}_{\rho_1}(t)\})\big) \quad \text{and} \quad \mathsf{J}_{\rho_1\to\rho_2}(r)(s) = \mathsf{J}_{\rho_2}\big(r(\bigcup\{t \mid \mathsf{J}_{\rho_1} \subseteq s\})\big).$$

We have not given the definition in this way because the proofs that follow only require the adjunction properties of L_{ρ_1} and U_{ρ_1}, and not any explicit characterisation. To unpack the definition a little more, suppose ρ is restricted to unary relations and consider the first few elements of this inductive family. When ρ is either o or $\iota \to o$, $\mathcal{S}[\![\rho]\!] = \mathcal{M}[\![\rho]\!]$, and I and J are both the identity. Consequently, they are both left and right adjoint to themselves, so that L and U are also both the identity. When ρ is $(\iota \to o) \to o$, by definition $\mathsf{J}_\rho(r) = \mathsf{J}_o \circ r \circ \mathsf{U}_{\iota\to o}$ but, as discussed, both of J_o and $\mathsf{U}_{\iota\to o}$ are identities on their respective domains, so $\mathsf{J}_\rho(r)$ is just r. However, $\mathcal{M}[\![(\iota \to o) \to o]\!]$ is strictly contained within $\mathcal{S}[\![(\iota \to o) \to o]\!]$, so J_ρ is merely an inclusion and, consequently, the induced right adjoint U_ρ is more interesting. Using Galois connection property (ii) it can be computed explicitly, revealing that it maps each $s \in \mathcal{S}[\![\rho]\!]$ to $\bigcup\{t \in \mathcal{M}[\![\rho]\!] \mid \mathsf{J}_\rho(t) \subseteq s\}$. But, we have seen that $\mathsf{J}_{(\iota\to o)\to o}(t) = t$, so it follows that $\mathsf{U}_\rho(s)$ is just the largest monotone relation included in s, and we arrive back at the discussion with which we started this subsection. The following proof gives more insight on the structure of the mappings.

Lemma 4.6. *For each ρ, (i) $\mathsf{L}_\rho \dashv \mathsf{I}_\rho$ and (ii) $\mathsf{J}_\rho \dashv \mathsf{U}_\rho$ are well-defined Galois connections.*

Proof. We prove only (i) because the proof of (ii) is analogous. We show that $\mathsf{I}_\rho : \mathcal{M}[\![\rho]\!] \to \mathcal{S}[\![\rho]\!]$ is a well-defined right adjoint by induction on ρ.

- When ρ is o, $\mathcal{M}[\![o]\!] = \mathcal{S}[\![o]\!]$ and I_o is the identity, which has left adjoint also the identity.
- When ρ is of shape $\iota \to \rho_2$, it follows from the induction hypothesis that $\mathsf{I}_{\rho_2} : \mathcal{M}[\![\rho_2]\!] \to \mathcal{S}[\![\rho_2]\!]$ is a well-defined right adjoint. It follows from Galois connection property (vi) that the mapping $r \mapsto \mathsf{I}_{\rho_2} \circ r : \mathcal{M}[\![\iota \to \rho_2]\!] \to \mathcal{S}[\![\iota \to \rho_2]\!]$ is a well-defined right adjoint.
- Finally, when ρ has shape $\rho_1 \to \rho_2$, we decompose the definition of I_ρ as follows:

$$\mathcal{M}[\![\rho_1]\!] \Rightarrow_m \mathcal{M}[\![\rho_2]\!] \xrightarrow{r \mapsto \mathsf{I}_{\rho_2} \circ r \circ \mathsf{L}_{\rho_1}} \mathcal{S}[\![\rho_1]\!] \Rightarrow_m \mathcal{S}[\![\rho_2]\!] \xrightarrow{s \mapsto s} \mathcal{S}[\![\rho_1]\!] \Rightarrow \mathcal{S}[\![\rho_2]\!]$$

  It follows from the induction hypothesis that $\mathsf{I}_{\rho_1} : \mathcal{M}[\![\rho_1]\!] \to \mathcal{S}[\![\rho_1]\!]$ and $\mathsf{I}_{\rho_2} : \mathcal{M}[\![\rho_2]\!] \to \mathcal{S}[\![\rho_2]\!]$ are both well-defined right-adjoints, from which we may infer the existence of left adjoint $\mathsf{L}_{\rho_1} : \mathcal{S}[\![\rho_1]\!] \to \mathcal{M}[\![\rho_1]\!]$. It follows from Galois connection property (iv) that the mapping $r \mapsto \mathsf{I}_{\rho_2} \circ r \circ \mathsf{L}_{\rho_1} : \mathcal{M}[\![\rho_1]\!] \Rightarrow_m \mathcal{M}[\![\rho_2]\!] \to \mathcal{S}[\![\rho_1]\!] \Rightarrow_m \mathcal{S}[\![\rho_2]\!]$ is a well-defined right adjoint (with codomain the monotone function space). Finally, observe that there is a canonical inclusion between the monotone and full function spaces which, since it trivially preserves meets, is as an right adjoint according to Galois connection property (ii). The result follows since right adjoints compose (Galois connection property (iii)).

□

The Galois connections give a canonical way to move between the universes of monotone and arbitrary relations; we extend them to mappings on valuations $\alpha \in \mathcal{M}[\![\Delta]\!]$ by:

$$\mathsf{I}_\Delta(\alpha)(x) = \begin{cases} \alpha(x) & \text{if } \Delta(x) = \iota \\ \mathsf{I}_{\Delta(x)}(\alpha(x)) & \text{otherwise} \end{cases} \qquad \mathsf{J}_\Delta(\alpha)(x) = \begin{cases} \alpha(x) & \text{if } \Delta(x) = \iota \\ \mathsf{J}_{\Delta(x)}(\alpha(x)) & \text{otherwise} \end{cases}$$

The action is pointwise on relations and trivial on individuals. It is easy to verify that each I_Δ and J_Δ are right and left adjoints respectively.

Corollary 4.7. *For each sorting Δ, (i) $\mathsf{L}_\Delta \dashv \mathsf{I}_\Delta$ and (ii) $\mathsf{J}_\Delta \dashv \mathsf{U}_\Delta$ are well-defined Galois connections.*

We now have a canonical way to move between monotone and arbitrary valuations, but our aim was to be able to map models of P_D to models of D (and vice versa), and we do not yet have any evidence that our mappings are at all useful in this respect. In both cases, models are prefixed points of certain functionals so we look for conditions which ensure that mappings preserve the property of being a prefix point. One such condition is the following: if $F : P \to Q$ is monotone, $T_1 : P \to P$ and $T_2 : Q \to Q$ then F will send prefixed points of T_1 to prefixed points of T_2 whenever it satisfies $T_2 \circ F \subseteq F \circ T_1$ (in the pointwise order). This is because if $T_1(x) \leq x$ then $F(T_1(x)) \leq F(x)$ by monotonicity, but also $T_2(F(x)) \leq F(T_1(x))$ by the assumption so that $T_2(F(x)) \leq F(x)$.

In the following we will prove that the right adjoints I and U preserve prefix points by showing that $T^{\mathcal{M}} \circ \mathsf{U} \subseteq \mathsf{U} \circ T^{\mathcal{S}}$ and $T^{\mathcal{S}} \circ \mathsf{I} \subseteq \mathsf{I} \circ T^{\mathcal{M}}$. The meat of the definitions of $T^{\mathcal{S}}$ and $T^{\mathcal{M}}$ lies in the semantics of goal terms, so we first show that, for all goal terms G, $\mathcal{M}[\![G]\!] \circ \mathsf{U} \subseteq \mathsf{U} \circ \mathcal{S}[\![G]\!]$ and $\mathcal{S}[\![G]\!] \circ \mathsf{I} \subseteq \mathsf{I} \circ \mathcal{M}[\![G]\!]$. However, we rephrase $\mathcal{M}[\![G]\!] \circ \mathsf{U} \subseteq \mathsf{U} \circ \mathcal{S}[\![G]\!]$ equivalently as $\mathsf{J} \circ \mathcal{M}[\![G]\!] \circ \mathsf{U} \subseteq \mathcal{S}[\![G]\!]$ and $\mathcal{S}[\![G]\!] \circ \mathsf{I} \subseteq \mathsf{I} \circ \mathcal{M}[\![G]\!]$ equivalently as $\mathcal{S}[\![G]\!] \subseteq \mathsf{I} \circ \mathcal{M}[\![G]\!] \circ \mathsf{L}$, which allows for a straightforward induction.

Lemma 4.8. *For all goal terms* $\Delta \vdash G : \rho$, $\mathsf{J}_\rho \circ \mathcal{M}[\![G]\!] \circ \mathsf{U}_\Delta \subseteq \mathcal{S}[\![G]\!] \subseteq \mathsf{I}_\rho \circ \mathcal{M}[\![G]\!] \circ \mathsf{L}_\Delta$.

The fact that the two right adjoints map between prefix points now follows immediately once the equivalence of our rephrasings have been verified.

Lemma 4.9 (Model translation). *Fix a program* $\vdash P : \Delta$.

(i) If β *is a prefixed point of* $T^{\mathcal{S}}_{P:\Delta}$ *then* $\mathsf{U}_\Delta(\beta)$ *is a prefixed point of* $T^{\mathcal{M}}_{P:\Delta}$.

(ii) If α *is a prefixed point of* $T^{\mathcal{M}}_{P:\Delta}$, *then* $\mathsf{I}_\Delta(\alpha)$ *is a prefixed point of* $T^{\mathcal{S}}_{P:\Delta}$.

Proof. It follows from Lemma 4.8 and Lemma 4.7 that, for any goal term G:

$$\mathcal{M}[\![G]\!](\mathsf{U}(\beta)) \subseteq \mathsf{U}(\mathcal{S}[\![G]\!](\beta)) \quad \text{and} \quad \mathcal{S}[\![G]\!](\mathsf{I}(\alpha)) \subseteq \mathsf{I}(\mathcal{M}[\![G]\!](\alpha))$$

The first follows from $\mathsf{J} \circ \mathcal{M}[\![G]\!] \circ \mathsf{U} \subseteq \mathcal{S}[\![G]\!]$ since J is left adjoint. The second follows from $\mathcal{S}[\![G]\!] \subseteq \mathsf{I} \circ \mathcal{M}[\![G]\!] \circ \mathsf{L}$ by pre-composing with I on both sides and noting that $\mathsf{L} \circ \mathsf{I}$ is deflationary. By definition: $T^{\mathcal{M}}(\alpha)(x) = \mathcal{M}[\![P(x)]\!](\alpha)$ and $T^{\mathcal{S}}(\beta)(x) = \mathcal{S}[\![P(x)]\!](\beta)$, so that we can deduce the following from the above inclusions:

$$T^{\mathcal{M}}(\mathsf{U}(\beta)) \subseteq \mathsf{U}(T^{\mathcal{S}}(\beta)) \quad \text{and} \quad T^{\mathcal{S}}(\mathsf{I}(\alpha)) \subseteq \mathsf{I}(T^{\mathcal{M}}(\alpha))$$

For part (i), it only remains to observe that if $T^{\mathcal{S}}(\beta) \subseteq \beta$ then, by monotonicity, $\mathsf{U}(T^{\mathcal{S}}(\beta)) \subseteq \mathsf{U}(\beta)$ and, by the above inclusion, $T^{\mathcal{M}}(\mathsf{U}(\beta)) \subseteq \mathsf{U}(\beta)$. Part (ii) is analogous. □

Moreover, these mappings preserve refutation of the goal. Intuitively, we think of $\mathsf{U}(\beta)$ (respectively $\mathsf{I}(\alpha)$) as the largest monotone (respectively standard) valuation that is smaller than β (respectively α). So, (as is made precise in Lemma 4.8) if the latter refutes a goal, so should the former. Hence, we obtain the following problem reduction.

Theorem 4.10. *The higher-order constrained Horn clause problem* $\langle \Delta, D, G \rangle$ *is solvable, iff the monotone logic program safety problem* $\langle \Delta, P_D, G \rangle$ *is solvable, and iff in all models of the background theory* $\mathcal{M}[\![G]\!](\mathcal{M}[\![P_D]\!]) = 0$.

Proof. We prove a chain of implications.

- Assume that $\langle \Delta, D, G \rangle$ is solvable, so that for each model A of the background theory, there is a valuation β and A, $\beta \vDash D$ and A, $\beta \nvDash G$, i.e. $\mathcal{S}[\![G]\!](\beta) = 0$. Fix such a model A of the background theory and then let β be the witness given above. Then it follows from Lemma 4.8 that $\mathsf{J}(\mathcal{M}[\![G]\!](\mathsf{U}(\beta))) \subseteq 0$, i.e. $\mathcal{M}[\![G]\!](\mathsf{U}(\beta)) = 0$. Since β is a model of D, it follows from Lemma 4.3 that it is also a prefixed point of $T^{\mathcal{S}}_{P_D}$ and hence $\mathsf{U}(\beta)$ is a prefixed point of $T^{\mathcal{M}}_{P_D}$ by Lemma 4.9. Therefore, $\langle \Delta, P_D, G \rangle$ is also solvable.

- If $\langle \Delta, P_D, G\rangle$ is solvable, then, for each model A of the background theory, there is a prefix point α of $T^{\mathcal{M}}_{P_D}$ and $\mathcal{M}[\![G]\!](\alpha) = 0$. However, $\mathcal{M}[\![P_D]\!]$ is, by definition, the least prefixed point so $\mathcal{M}[\![P_D]\!] \subseteq \mathsf{U}(\alpha)$ and hence $\mathcal{M}[\![G]\!](\mathcal{M}[\![P_D]\!]) = 0$ follows by monotonicity.
- Finally assume that in all models of the background theory $\mathcal{M}[\![G]\!](\mathcal{M}[\![P_D]\!]) = 0$. Fix such a model of the background theory. We claim that $\mathsf{L}(\mathcal{S}[\![G]\!](\mathsf{I}(\mathcal{M}[\![P_D]\!]))) \subseteq 0$ follows from Lemma 4.8, which is to say that $\mathcal{S}[\![G]\!](\mathsf{I}(\mathcal{M}[\![P_D]\!])) = 0$. To see this, observe that $\mathcal{S}[\![G]\!](\mathsf{I}(\mathcal{M}[\![P_D]\!])) \subseteq \mathsf{I}(\mathcal{M}[\![G]\!](\mathcal{M}[\![P_D]\!]))$ follows as in the proof of Lemma 4.9 and note that I is right adjoint. Since $\mathcal{M}[\![P_D]\!]$ is a prefixed point of $T^{\mathcal{M}}_{P_D:\Delta}$, it follows from Lemma 4.9 that $\mathsf{I}(\mathcal{M}[\![P_D]\!])$ is a prefixed point of $T^{\mathcal{S}}_{P_D}$. Finally, by Lemma 4.3 it is therefore a model of D, so $\langle \Delta, D, G\rangle$ is solvable.

□

5 REFINEMENT TYPE ASSIGNMENT

Having reduced the higher-order Horn clause problem to a problem about the least-fixpoint semantics of higher-order logic programs, we now consider the task of automating reasoning about such programs. For this purpose, we look to work on refinement type systems, which are one of the most successful approaches to automatically obtaining invariants for higher-order, functional programs. In this section, we develop a refinement type system for monotone logic programs.

Elimination of higher-order existentials. Due to monotonicity, it is possible to eliminate higher-order existential quantification from goal terms, simplifying the design of the type system. Given any goal term of shape $\exists x : \rho.\, G$, observe that $\mathcal{M}[\![G]\!]$ is a monotone function of x. If there is a relation that can be used as a witness for x then, by monotonicity, so too can any larger relation. In particular, G is true of some x of sort ρ iff G is true of the universal relation of sort ρ, that is, the relation u_ρ satisfying $u_\rho(r_1)\cdots(r_k) = 1$ for all $r_1, \ldots, r_k$. Moreover, the universal relation of sort ρ is itself definable by a goal term, it is just the term $U_\rho \coloneqq \lambda x_1 \ldots x_k.\, \mathsf{true}$. Consequently, $\mathcal{M}[\![\exists x : \rho.\, G : o]\!] = \mathcal{M}[\![G[U_\rho/x]]\!]$, in which the instance of existential quantification over relations has been eliminated by a syntactic substitution. For the rest of the paper, we assume without loss of generality that monotone logic programs contain only existential quantification over individuals.

5.1 Syntax

The refinement types are built out of constraint formulas which are combined using the dependent arrow. For the purposes of this section, we shall assume that the constraint language is closed under conjunction and closed under (well-sorted) substitution, in the sense that, for every constraint formula $\Delta,\, x{:}\,\iota \vdash \varphi : o \in \mathit{Fm}$ and term $\Delta \vdash N : \iota \in \mathit{Tm}$, it follows that $\varphi[N/x] \in \mathit{Fm}$.

Types. The restricted syntax of goal terms allows us to make several simplifications to our refinement type system, in comparison to those in the literature. The first is that we only allow refinement of the propositional sort o and we only allow dependence on the sort of individuals ι. Formally, we define the set of type expressions according to the following grammar:

$$(\textsc{Type}) \quad T \quad ::= \quad o\langle\varphi\rangle \mid x{:}\iota \to T \mid T_1 \to T_2$$

in which $\varphi \in \mathit{Fm}$ is a constraint formula. We make this definition under the assumption that both kinds of arrow associate to the right and we identify types up to α-equivalence.

Refinement. We restrict attention to those types that we consider to be well-formed, which is defined by a system of judgements $\Delta \vdash T :: \rho$, in which Δ is a sort environment, T is a type and ρ is a relational sort. In case such a judgement is provable we say that T *refines* ρ.

$$\frac{}{\Delta \vdash o\langle\varphi\rangle :: o}\ \Delta \vdash \varphi : o \in Fm \qquad \frac{\Delta, x{:}\,\iota \vdash T :: \rho}{\Delta \vdash x{:}\iota \to T :: \iota \to \rho} \qquad \frac{\Delta \vdash T_1 :: \rho_1 \qquad \Delta \vdash T_2 :: \rho_2}{\Delta \vdash T_1 \to T_2 :: \rho_1 \to \rho_2}$$

Type environments. A *type environment* Γ is a finite sequence of pairs of variables and types, $x : T$, such that the variable subjects are pairwise distinct. We write the empty sort environment as ϵ. We place similar well-formedness restrictions on environments, using the judgement $\vdash \Gamma :: \Delta$, in which Γ is a type environment and Δ a sort environment. In case such a judgement is provable we say that Γ *refines* Δ.

$$\frac{}{\vdash \epsilon :: \epsilon} \qquad \frac{\vdash \Gamma :: \Delta}{\vdash (\Gamma,\, x : \iota) :: (\Delta,\, x : \iota)} \qquad \frac{\vdash \Gamma :: \Delta \qquad \Delta \vdash T :: \rho}{\vdash (\Gamma,\, x : T) :: (\Delta,\, x : \rho)}$$

Since the variable subjects of a type environment are required to be distinct, we will frequently view such an environment as a finite map from variables to types. Thus, whenever x is a subject in Γ, we will write $\Gamma(x)$ for the type assigned to x and $\mathrm{dom}(\Gamma)$ for its set of subjects.

Subtype theory. Much of the power of refinement types is derived from the associated subtype theory, which imports wholesale the reasoning apparatus of the underlying constraint language. Subtyping between types is the set of inequalities given by the judgement form $\vdash T_1 \sqsubseteq T_2$ defined inductively.

$$\frac{}{\vdash o\langle\varphi\rangle \sqsubseteq o\langle\psi\rangle}\ \mathcal{T}h \vDash \varphi \Rightarrow \psi \qquad \frac{\vdash T_1 \sqsubseteq T_2}{\vdash x{:}\iota \to T_1 \sqsubseteq x{:}\iota \to T_2} \qquad \frac{\vdash T_1' \sqsubseteq T_1 \qquad \vdash T_2 \sqsubseteq T_2'}{\vdash T_1 \to T_2 \sqsubseteq T_1' \to T_2'}$$

It is natural to view $\sqsubseteq$ as a preorder and useful to distinguish the extremal elements. We write $\top_\rho$ and $\bot_\rho$ for the families of refinement types defined inductively as follows:

$$\begin{array}{rclcrcl} \top_o & = & o\langle\mathsf{true}\rangle & & \bot_o & = & o\langle\mathsf{false}\rangle \\ \top_{\iota\to\rho} & = & z{:}\iota \to \top_\rho & & \bot_{\iota\to\rho} & = & z{:}\iota \to \bot_\rho \\ \top_{\rho_1\to\rho_2} & = & \bot_{\rho_1} \to \top_{\rho_2} & & \bot_{\rho_1\to\rho_2} & = & \top_{\rho_1} \to \bot_{\rho_2} \end{array}$$

in which the variables z are chosen to be suitably fresh. It is clear that, by construction, $\vdash \top_\rho :: \rho$ and $\vdash \bot_\rho :: \rho$.

Type assignment. Type assignment for goal terms $\Delta \vdash G : \sigma$ and programs $\vdash P : \Delta$ are defined by a system of judgements of the forms $\Gamma \vdash G : T$ and $\vdash P : \Gamma$ respectively, in which type environment Γ and type T are required to satisfy $\vdash \Gamma :: \Delta$ and $\Delta \vdash T :: \sigma$. The system is defined in Figure 2.

Application and abstraction. There are two versions of each of abstraction and application, corresponding to the fact that we have chosen to emphasize the difference between dependence of a result type on an argument of individual sort and non-dependence on arguments of relational sort.

In common with other refinement type systems in the literature but unlike more general dependent type systems, our types are not closed under substitution of arbitrary terms of the programming language. This can be reconciled with the usual rule for dependent application, in which substitution $T[N/x]$ into a type T occurs, in a number of ways. For example, in the system of Rondon et al. [2008], terms N of the programming language that are substituted into a type T through application are understood using uninterpreted function symbols in the logic; in the system of Unno and Kobayashi [2009], the rule for application trivialises the substitution by requiring that T does not contain the dependent variable x and; in the system of Terauchi [2010], the operand N is required to be a variable (which can be guaranteed when the program is assumed to be in A-normal

$$\text{(TVar)}\ \frac{}{\Gamma_1,\, x{:}\, T,\, \Gamma_2 \vdash x : T} \qquad \text{(TConstraint)}\ \frac{}{\Gamma \vdash \varphi : o\langle\varphi\rangle}\ \varphi \in Fm$$

$$\text{(TSub)}\ \frac{\Gamma \vdash G : T_1 \qquad \vdash T_1 \sqsubseteq T_2}{\Gamma \vdash G : T_2} \qquad \text{(TExists)}\ \frac{\Gamma, x : \iota \vdash G : o\langle\varphi\rangle}{\Gamma \vdash \exists x{:}\iota.\, G : o\langle\psi\rangle}\ \mathcal{T}h \vDash \varphi \Rightarrow \psi$$

$$\text{(TAnd)}\ \frac{\Gamma \vdash G : o\langle\varphi\rangle \qquad \Gamma \vdash H : o\langle\psi\rangle}{\Gamma \vdash G \wedge H : o\langle\varphi \wedge \psi\rangle} \qquad \text{(TOr)}\ \frac{\Gamma \vdash G : o\langle\varphi\rangle \qquad \Gamma \vdash H : o\langle\psi\rangle}{\Gamma \vdash G \vee H : o\langle\chi\rangle}\ \begin{array}{l}\mathcal{T}h \vDash \varphi \Rightarrow \chi \\ \mathcal{T}h \vDash \psi \Rightarrow \chi\end{array}$$

$$\text{(TAbsR)}\ \frac{\Gamma, x{:}\, T_1 \vdash G : T_2}{\Gamma \vdash \lambda x{:}\rho.\, G : T_1 \to T_2} \qquad \text{(TAppR)}\ \frac{\Gamma \vdash G : T_1 \to T_2 \qquad \Gamma \vdash H : T_1}{\Gamma \vdash G\, H : T_2}$$

$$\text{(TAbsI)}\ \frac{\Gamma, x{:}\, \iota \vdash G : T}{\Gamma \vdash \lambda x{:}\iota.\, G : x{:}\iota \to T} \qquad \text{(TAppI)}\ \frac{\Gamma \vdash G : x{:}\iota \to T \qquad \vdash \Gamma :: \Delta}{\Gamma \vdash G\, N : T[N/x]}$$

Fig. 2. Type assignment for goal terms.

form). In our case, dependence can only occur at sort ι and, since the subjects of the system are goal terms, all subterms of sort ι are necessarily already terms of the constraint language, so we avoid the need for any further conditions.

Subtyping. Our subsumption rule (TSub) is quite standard, we just note that the fact that there are no non-trivial refinements of the base sort ι has the consequence that there is no significant advantage for the subtype judgement to refer to the type context Γ, which is why it has been formulated in this context-free way. Similar comments can be made about avoiding the need to distinguish between base and function types in (TVar) .

Constraints and logical constants. To understand the rules for typing constraint, existential, conjunctive and disjunctive formulas, it is instructive to assign a meaning to the judgement $\Gamma \vdash G : o\langle\varphi\rangle$. One should view this judgement as asserting that: in those valuations that satisfy Γ, G implies φ. This statement is made precise in Lemma 5.2 once the semantics of types has been introduced. Under this reading, we can view the type system as a mechanism for concluding assertions of the form "goal formula G is approximated by constraint φ" or "constraint φ is an abstraction of goal formula G". This is a useful assertion for automated reasoning because it is relating the complicated formula G, which may include higher-order relation symbols whose meanings are defined recursively by the program, to the much more tractable constraint formula φ, which is drawn from a (typically decidable) first-order theory.

This view helps to clarify the intuition behind the rules for assigning types to formulas headed by a constant. In particular, for goal terms G that are themselves formulas of the constraint language, the rule (TConstraint) loses no information in the abstraction. Note that we give only a rule for typing existential quantification at base sort ι, which is justified by the remarks at the start of this section. Finally, observe that the side condition on (TExists) is equivalent to the condition $\mathcal{T}h \vDash (\exists x.\, \varphi) \Rightarrow \psi$ since, due to the well-formedness of the judgement, ψ cannot contain x freely. We use the side condition in the given form because the constraint language may not allow existential quantification. A similar remark may be made concerning the rule (TOr) and the ability to express disjunction.

To programs $\vdash P : \Delta$, of shape $x_1 = G_1, \ldots, x_m = G_m$, we assign type environments according to the rule:

$$\text{(TProg)}\quad \frac{\Gamma \vdash G_1 : \Gamma(x_1) \qquad \cdots \qquad \Gamma \vdash G_m : \Gamma(x_m)}{\vdash x_1 = G_1, \quad \ldots, \quad x_m = G_m : \Gamma}$$

Once we have defined the relational semantics of types, in which a type environment Γ is interpreted as a valuation $\mathcal{M}(\!|\Gamma|\!)$, the soundness of this rule will guarantee that $\mathcal{M}(\!|\Gamma|\!)$ is a prefix point of $T^{\mathcal{M}}_{P:\Delta}$, and hence is an over-approximation of $\mathcal{M}[\![P]\!]$.

Example 5.1. Using (TProg) the program in Example 4.2 can be assigned the type environment Γ_I from the introduction. For example, the type of *Iter* can be justified from the following judgements. First, for the subterms $n \leq 0$ and $m = 0$ in the body of *Iter* we can apply the (TConstraint) rule to immediately derive the judgements:

$$\Gamma' \vdash n \leq 0 : o\langle n \leq 0\rangle \quad \text{and} \quad \Gamma' \vdash m = s : o\langle m = s\rangle$$

in which Γ' is the type environment Γ, f: x:int $\to$ y:int $\to$ z:int $\to$ $o\langle 0 < x \Rightarrow y < z\rangle$, s: int, n: int, m: int. Then using (TAnd), we can derive the judgement: $\Gamma' \vdash n \leq 0 \wedge m = s : o\langle n \leq 0 \wedge m = s\rangle$. Moreover, since also $\mathsf{ZLA} \vDash n \leq 0 \wedge m = s \Rightarrow 0 \leq s \Rightarrow n \leq m$, it follows from the subsumption rule (TSub) that the judgement:

$$\Gamma' \vdash n \leq 0 \wedge m = s : o\langle 0 \leq s \Rightarrow n \leq m\rangle \tag{2}$$

is also derivable. Next, consider the subterm *Iter* $f\ s\ (n-1)\ p$. Observe that $\vdash o\langle z = x + y\rangle \sqsubseteq o\langle 0 < x \Rightarrow y < z\rangle$ so, using (TVar) and (TSub) we can assign to f the type

$$x\text{:int} \to y\text{:int} \to z\text{:int} \to o\langle 0 < x \Rightarrow y < z\rangle$$

Hence, it follows from the (TAppR) and (TAppI) rules that we can derive the judgement: $\Gamma' \vdash \mathit{Iter}\, f\, s\, (n-1)\, p : o\langle 0 \leq s \Rightarrow n - 1 \leq p\rangle$. To the term $f\ n\ p\ m$ we can assign the type $o\langle 0 < n \Rightarrow p < m\rangle$. Hence, by (TAnd) and then (TExists) we can conclude the judgement:

$$\Gamma' \vdash \exists p.\, 0 < n \wedge \mathit{Iter}\, f\ s\, (n-1)\ p \wedge f\ n\ p\ m : o\langle 0 \leq s \Rightarrow n \leq m\rangle \tag{3}$$

since $\mathsf{ZLA} \vDash (\exists p.\, 0 < n \wedge (0 \leq s \Rightarrow n - 1 \leq p) \wedge (0 < n \Rightarrow p < m)) \Rightarrow 0 \leq s \Rightarrow n \leq m$. Using (1), (2) and (TOr) we can therefore derive an overall type for the body of the definition of *Iter* as $o\langle 0 \leq s \Rightarrow n \leq r\rangle$. The desired type follows from applications of (TAbsI) and (TAbsR).

5.2 Semantics

We ascribe two meanings to types. The first is the usual semantics in which types are some kind of set and, as is typical, such sets have the structure of an order ideal. The second is specific to our setting and exploits the fact that every type refines a relational sort. The second semantics assigns to each type a specific relation. This makes later developments, such as the demonstration of the soundness of type assignment, simpler and also makes a link to the notion of symbolic model from the first-order case.

Ideal semantics of types. The *ideal semantics* of refinement types is defined so as to map a well-formed typing sequent $\Delta \vdash T :: \rho$ and appropriate valuation α to a subset $\mathcal{M}[\![\Delta \vdash T :: \rho]\!](\alpha) \subseteq \mathcal{M}[\![\rho]\!]$.

$$\begin{aligned}
\mathcal{M}[\![\Delta \vdash o\langle\varphi\rangle :: o]\!](\alpha) &= \{0, \mathcal{S}[\![\Delta \vdash \varphi : o]\!](\alpha)\} \\
\mathcal{M}[\![\Delta \vdash x{:}\iota \to T :: \iota \to \rho]\!](\alpha) &= \{r \mid \forall n \in A_\iota.\ r(n) \in \mathcal{M}[\![\Delta, x : \iota \vdash T :: \rho]\!](\alpha[x \mapsto n])\} \\
\mathcal{M}[\![\Delta \vdash T_1 \to T_2 :: \rho_1 \to \rho_2]\!](\alpha) &= \{f \mid \forall r \in \mathcal{M}[\![\Delta \vdash T_1 :: \rho_1]\!](\alpha).\ f(r) \in \mathcal{M}[\![\Delta \vdash T_2 :: \rho_2]\!](\alpha)\}
\end{aligned}$$

We write $\mathcal{M}[\![T]\!](\alpha)$ for $\mathcal{M}[\![\Delta \vdash T :: \rho]\!]$ when the judgement $\Delta \vdash T :: \rho$ is clear from the context. We are now in a position to make precise the remark following the definition of type assignment.

Lemma 5.2. $\mathcal{M}[\![G]\!](\alpha) \in \mathcal{M}[\![o\langle\varphi\rangle]\!](\alpha)$ *iff* $\alpha \vDash G \Rightarrow \varphi$.

The soundness of type assignment will guarantee that, additionally, $\Gamma \vdash G : o\langle\varphi\rangle$ implies that $\mathcal{M}[\![G]\!](\alpha) \in \mathcal{M}[\![o\langle\varphi\rangle]\!](\alpha)$ for all appropriate α.

Relational semantics of types. We will also consider a monotone *relational* semantics of types. Fix a sort environment Δ. Given an appropriate first-order valuation α, we can associate relations $\mathcal{M}(\!|\Delta \vdash T :: \rho|\!)(\alpha) \in \mathcal{M}[\![\rho]\!]$ to each well formed type $\Delta \vdash T : \rho$.

$$\begin{array}{rcll}
\mathcal{M}(\!|\Delta \vdash o\langle\varphi\rangle :: o|\!)(\alpha) & = & \mathcal{S}[\![\Delta \vdash \varphi : o]\!](\alpha) & \\
\mathcal{M}(\!|\Delta \vdash x{:}\iota \to T :: \iota \to \rho|\!)(\alpha)(n) & = & \mathcal{M}(\!|\Delta, x : \iota \vdash T :: \rho|\!)(\alpha[x \mapsto n]) & \\
\mathcal{M}(\!|\Delta \vdash T_1 \to T_2 :: \rho_1 \to \rho_2|\!)(\alpha)(r) & = & \mathcal{M}(\!|\Delta \vdash T_2 :: \rho_2|\!)(\alpha) & \text{if } r \subseteq \mathcal{M}(\!|\Delta \vdash T_1 :: \rho_1|\!)(\alpha) \\
\mathcal{M}(\!|\Delta \vdash T_1 \to T_2 :: \rho_1 \to \rho_2|\!)(\alpha)(r) & = & \mathcal{M}(\!|\epsilon \vdash \top_{\rho_2} :: \rho_2|\!)(\emptyset) & \text{otherwise}
\end{array}$$

We write $\mathcal{M}(\!|T|\!)$ for $\mathcal{M}(\!|\Delta \vdash T :: \rho|\!)$ whenever the judgement $\Delta \vdash T :: \rho$ is clear from the context. It follows straightforwardly from the definitions that $\mathcal{M}(\!|T|\!)(\alpha)$ is a monotone relation.

Symbolic models. Additionally, we can consider a type environment under a relational interpretation. We define, for all $x \in \mathrm{dom}(\Delta)$, $\mathcal{M}(\!|\vdash \Gamma :: \Delta|\!)(x) = \mathcal{M}(\!|\Gamma(x)|\!)(\emptyset)$. We write $\mathcal{M}(\!|\Gamma|\!)$ for $\mathcal{M}(\!|\vdash \Gamma :: \Delta|\!)$ whenever the judgement $\vdash \Gamma :: \Delta$ is otherwise clear from the context. This object is a monotone valuation and hence, following the results of Section 4, this semantics makes precise the idea of a type environment Γ as a (candidate) *symbolic model*, that is, a finite representation of a model designed for the purposes of automated reasoning.

Relationship. The ideal and the relational semantics are closely related: the relational interpretation $\mathcal{M}(\!|T|\!)(\alpha)$ of a type T is the largest element of the ideal $\mathcal{M}[\![T]\!](\alpha)$. Thus we see also that $\mathcal{M}[\![T]\!](\alpha)$ is principal. For each relation $r \in \mathcal{M}[\![\rho]\!]$ let us write $\Downarrow_\rho r$ for the downward closure of $\{r\}$ in $\mathcal{M}[\![\rho]\!]$ (we will typically omit the subscript).

Lemma 5.3. *For all types* $\Delta \vdash T :: \rho$ *and* $\alpha \in \mathcal{S}[\![\Delta]\!]$, $\mathcal{M}[\![T]\!](\alpha) = \Downarrow \mathcal{M}(\!|T|\!)(\alpha)$.

5.3 Soundness

We return to the discussion of type assignment by defining semantic judgements paralleling those for the type system. First let us make a preliminary definition. Let $\vdash \Gamma :: \Delta$, then we say that valuation $\alpha \in \mathcal{M}[\![\Delta]\!]$ *satisfies* Γ just if, for all $x \in \mathrm{dom}(\Delta)$, $\alpha(x) \in \mathcal{M}[\![\Gamma(x)]\!]$. We write the following judgement forms:

$$\vDash T_1 \sqsubseteq T_2 \qquad\qquad \Gamma \vDash G : T \qquad\qquad \vDash P : \Gamma$$

In the first, we assume that $\Delta \vdash T_1 : \rho$ and $\Delta \vdash T_2 : \rho$ are types of the same sort. Then the meaning of $\vDash T_1 \sqsubseteq T_2$ is that, for all $\alpha \in \mathcal{M}[\![\Delta]\!]$, $\mathcal{M}[\![T_1]\!](\alpha) \subseteq \mathcal{M}[\![T_2]\!](\alpha)$. In the second, we assume that $\vdash \Gamma :: \Delta$, $\Delta \vdash G : \rho$ and $\Delta \vdash T :: \rho$. Then the meaning of $\Gamma \vDash G : T$ is that, for all valuations $\alpha \in \mathcal{M}[\![\Delta]\!]$, if α satisfies Γ then $\mathcal{M}[\![G]\!](\alpha) \in \mathcal{M}[\![T]\!](\alpha)$. Finally, in the third, we assume that $\vdash P : \Delta$ and $\vdash \Gamma :: \Delta$. Then the meaning of $\vDash P : \Gamma$ is that $\mathcal{M}[\![P]\!]$ satisfies Γ.

Theorem 5.4 (Soundness of Type Assignment).

(i) If $\vdash T_1 \sqsubseteq T_2$ *then* $\vDash T_1 \sqsubseteq T_2$.
(ii) If $\Gamma \vdash G : T$ *then* $\Gamma \vDash G : T$.
(iii) If $\vdash P : \Gamma$ *then* $\vDash P : \Gamma$.

Proof sketch. The first and second claims are proven by straightforward inductions on the relevant judgements. In the third case it follows from (ii) and the definition of (TProg) that

$\mathcal{M}(\!|\Gamma|\!)$ satisfies Γ and is therefore a prefixed point of $T^{\mathcal{M}}_{P:\Delta}$. Since $\mathcal{M}[\![P]\!]$ is the least such, the result follows. □

This allows for a sound approach to solving systems of higher-order constrained Horn clauses. Given an instance of the problem $\langle \Delta, D, G\rangle$, we construct the corresponding logic program $\langle \Delta, P_D, G\rangle$. If there is a type environment $\vdash \Gamma :: \Delta$ such that $\vdash P_D : \Gamma$ and $\Gamma \vdash G : o\langle\mathsf{false}\rangle$ then it follows from Theorem 5.4 that, for each model A of the background theory, $\mathcal{M}(\!|\Gamma|\!)$ (with constants interpreted with respect to A) is a valuation that satisfies D but refutes G. The approach is, however, incomplete. Consider the following instance, adapted from an example of [Unno et al. 2013] showing the incompleteness of refinement type systems for higher-order program verification.

Example 5.5. The higher-order constrained Horn clause problem $\langle \Delta, D, G\rangle$ specified by, respectively:

$$\begin{array}{ll} \mathit{Leq} : \mathsf{int} \to \mathsf{int} \to o & \forall i\ j.\ i \leq j \Rightarrow \mathit{Leq}\ i\ j \\ \mathit{Holds} : (\mathsf{int} \to o) \to \mathsf{int} \to o & \forall p\ n.\ p\ n \Rightarrow \mathit{Holds}\ p\ n \end{array} \qquad \exists i.\ \mathit{Holds}\ (\mathit{Leq}\ i)\ (i-1)$$

is solvable in the theory of integer linear arithmetic (ZLA), since there is no integer i smaller than its predecessor. However, the logic program P_D defined as $\mathit{Leq} = \lambda ij.\ i \leq j$, $\mathit{Holds} = \lambda pn.\ pn$ is not typable. This is because if there were a type environment $\vdash \Gamma :: \Delta$ for which $\vdash P_D : \Gamma$ and $\Gamma \vdash G : o\langle\mathsf{false}\rangle$, it would have shape:

$$\mathit{Leq} : i{:}\mathsf{int} \to j{:}\mathsf{int} \to o\langle\chi\rangle \qquad \mathit{Holds} : (x{:}\mathsf{int} \to o\langle\varphi\rangle) \to n{:}\mathsf{int} \to o\langle\psi\rangle$$

for some formulas $i{:}\ \mathsf{int}, j{:}\ \mathsf{int} \vdash \chi : o$ and $x{:}\ \mathsf{int} \vdash \varphi : o$ and $n{:}\ \mathsf{int} \vdash \psi : o$. These formulas would necessarily satisfy the following conditions: (1) $\mathsf{ZLA} \vDash i \leq j \Rightarrow \chi$, (2) $\mathsf{ZLA} \vDash \varphi[n/x] \Rightarrow \psi$, (3) $\mathsf{ZLA} \vDash \chi[x/j] \Rightarrow \varphi$, and (4) $\mathsf{ZLA} \vDash (\exists i.\ \varphi[i-1/n]) \Rightarrow \mathsf{false}$, implied by the definition of the type system. It follows from (1) and (3) that also $\mathsf{ZLA} \vDash i \leq x \Rightarrow \varphi$. Since φ does not contain i freely, it follows that this is equivalent to $\mathsf{ZLA} \vDash (\exists i.\ i \leq x) \Rightarrow \varphi$, itself equivalent to $\mathsf{ZLA} \vDash \varphi$. However, this contradicts (4), so there can be no such type assignment.

It seems likely that, to obtain a sound and (relatively) complete approach, one could adapt the development described by Unno et al. [2013], at the cost of complicating the system a little.

5.4 Automation

This approach to solving the higher-order constrained Horn clause problems relies on finding a refinement type environment to act as a witness. It is well understood that, for refinement type systems following a certain pattern, typability (i.e. the existence of a type environment satisfying some properties) can be reduced to first-order constrained Horn clause solving, see e.g. the work of Terauchi [2010] Bjørner, McMillan, and Rybalchenko [2012] or Jhala, Majumdar, and Rybalchenko [2011]. Hence, the search for a witnessing environment can be automated using standard techniques (an explicit definition is included in the supplementary materials).

In order to check that the approach is feasible, we have implemented a prototype tool and used it to automatically verify a few small systems of clauses. The tool is written in Haskell and uses the Parsec library for parsing input in a mathematical syntax using unicode characters or ascii equivalents. Output is either a similar format or conforms to SMT-LIB in a manner that lets Z3 [De Moura and Bjørner 2008] solve the system of first order clauses.

The test cases are obtained from that subset of the functional program verification problems given in the work of Kobayashi et al. [2011] which do not make local assertions, re-expressed as higher-order Horn clause problems according to the method sketched by example in Section 1. The prototype[9] and the exact suite of test cases is available at http://github.com/penteract/

[9] A basic web interface to the prototype tool is available at http://mjolnir.cs.ox.ac.uk/horus.

HigherOrderHornRefinement. In all but one of the examples the prototype takes around 0.01s to transform the system of clauses and Z3 takes around 0.02s to solve the resulting first-order system. The remaining example, named neg, suffers from our choice of refinement type system. It is solved by the system described in *loc. cit.*, which allows type-level intersection, but its solution cannot be described using the types of our system.[10]

5.5 Expressibility of Type Assertions

It is possible to express the complement of a type using a goal formula. For example, according to the forgoing semantics, the type

$$T \coloneqq (x{:}\iota \to y{:}\iota \to o\langle x \equiv y \bmod 2\rangle) \to z{:}\iota \to o\langle z \neq 0\rangle$$

represents the set $\mathcal{M}[\![T]\!](\emptyset)$ of all monotone relations of sort $(\iota \to \iota \to o) \to \iota \to o$ that relate all parity-preserving inputs to non-zero outputs. The complement of this set of relations can be defined (in the monotone semantics) by the goal term $\mathsf{Com}(T) \coloneqq \lambda z.\, z\,(\lambda xy.\, x \equiv y \bmod 2)\; 0$ which classifies such relations according to whether or not they relate the particular parity-preserving relation $\lambda xy.\, x \equiv y \bmod 2$ to 0. The mode of definition we have in mind is the following.

Goal definability. We say that a relation $r \in \mathcal{M}[\![\rho]\!]$ is *goal term definable* (more briefly *G-definable*) just if there exists a closed goal term $\vdash H : \rho$ and $\mathcal{M}[\![H]\!](\emptyset) = r$. We say that a class of relations is *G*-definable just if the characteristic predicate of the class is *G*-definable.

Returning to the example, if a given relation r does not relate $\lambda xy.\, x \equiv y \bmod 2$ to 0, then r is not a member of $\mathcal{M}[\![T]\!](\emptyset)$. On the other hand, if r is not a member $\mathcal{M}[\![T]\!](\emptyset)$, then there is some parity-preserving relation s that is related by r to 0. Since r is monotone, it follows that r also relates all relations larger than s to 0. Since $\lambda xy.\, x \equiv y \bmod 2$ is the largest of the parity-preserving relations, it follows that r relates $\lambda xy.\, x \equiv y \bmod 2$ to 0. Hence the set

$$\{r \in \mathcal{M}[\![(\iota \to \iota \to o) \to \iota \to o]\!] \mid r \notin \mathcal{M}[\![T]\!](\emptyset)\}$$

is *G*-definable by the goal term $\mathsf{Com}(T)$.

Definability of type complements and the relational semantics. Any *G*-definition $\mathsf{Com}(T)$ of the complement of T is intertwined with a *G*-definition of the largest element of T: to understand when a relation r is not in the type $T_1 \to T_2$ you must understand when there exists a relation $s \in \mathcal{M}[\![T_1]\!](\emptyset)$ such that $r(s)$ is not in $\mathcal{M}[\![T_2]\!](\emptyset)$, i.e. when the property $\mathcal{M}[\![\mathsf{Com}(T)]\!](\emptyset)(r(s))$ holds. However, since $\mathcal{M}[\![\mathsf{Com}(T)]\!](\emptyset)$ and r are monotone, if $\mathcal{M}[\![\mathsf{Com}(T)]\!](\emptyset)(r(s))$ holds then $\mathcal{M}[\![\mathsf{Com}(T)]\!](r(s'))$ holds for any $s' \supseteq s$. Consequently, there exists an $s \in \mathcal{M}[\![T_1]\!](\emptyset)$ satisfying the property iff the largest element of $\mathcal{M}[\![T_1]\!](\emptyset)$ satisfies the property. This leads to the definitions by mutual induction on type:

$$\begin{array}{ll}
\mathsf{Com}(o\langle\varphi\rangle) = \lambda z.\, z \wedge \neg\varphi & \mathsf{Lar}(G)(o\langle\varphi\rangle) = G \vee \varphi \\
\mathsf{Com}(x{:}\iota \to T) = \lambda z.\, \exists x{:}\iota.\, \mathsf{Com}(T)\,(z\,x) & \mathsf{Lar}(G)(x{:}\iota \to T) = \lambda z.\, \mathsf{Lar}(G)(T[z/x]) \\
\mathsf{Com}(T_1 \to T_2) = \lambda z.\, \mathsf{Com}(T_2)\,(z\,\mathsf{Lar}(\mathsf{false})(T_1)) & \mathsf{Lar}(G)(T_1 \to T_2) = \lambda z.\, \mathsf{Lar}(G \vee \mathsf{Com}(T_1)\,z)(T_2)
\end{array}$$

in which $\mathsf{Com}(T)$ is a *G*-definition of the complement of the class $\mathcal{M}[\![T]\!](\emptyset)$. The definition of Lar is parametrised by a goal formula representing domain conditions that are accumulated during the analysis of function types. It follows that $\mathsf{Lar}(\mathsf{false})(T)$ is a *G*-definition of the largest element of the class $\mathcal{M}[\![T]\!](\emptyset)$.

[10] We know of no specific difficulty with incorporating intersection types into our system, but it would not involve any new ideas that are not already better demonstrated in [Terauchi 2010; Unno and Kobayashi 2009].

Lemma 5.6. *For any closed type* $\vdash T :: \rho$*:*

(i) The goal term $\mathsf{Com}(T)$ *is a G-definition of the class* $\{r \in \mathcal{M}[\![\rho]\!] \mid r \notin \mathcal{M}[\![T]\!](\emptyset)\}$.

(ii) The goal term $\mathsf{Lar}(\mathsf{false})(T)$ *is a G-definition of the relation* $\mathcal{M}(\!|T|\!)(\emptyset)$.

This gives a sense in which we can express the higher-type property $G : T$ within the higher-order constrained Horn clause framework, namely as the goal formula $\mathsf{Com}(T)\ G$ defining the negation of the type assertion (recall that in the higher-order Horn clause problem we aim to refute the negation of property to be proven, expressed as a goal formula). Furthermore, we can use this result in order to justify an extension of the syntax of higher-order constrained Horn clauses with a new kind of *type-guarded* existential quantification. This allows us to state goal formulas like $\exists x{:}T.G$, i.e. there exists a relation in (the set defined by) refinement type T that moreover satisfies G. The full development is contained in the anonymous supplementary materials.

6 RELATED WORK

Constrained Horn-clause solving in first-order program verification. Our motivation comes mainly from the use of constrained Horn clauses to express problems in the verification of first-order, imperative programs. The papers of Bjørner et al. [2012] and Bjørner et al. [2015] argue the case for the approach and provide a good overview of it. One of the best recommendations for the approach is the selection of highly efficient solvers that are available, such as [Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Hoder et al. 2011], which we exploit in this work as part of the automation of our prototype solver for higher-order clauses.

Higher-order logic programming. Work on higher-order logic programming is typically concerned with programming language design and implementation. Consequently, one of the main themes of the work discussed in the following is that of (i) finding a good semantics for higher-order logic programming languages, and one of the central criteria for a good semantics is that (ii) it lends itself well to developing techniques for enumerating answers to queries. In contrast to (i), our work is about a particular satisfiability problem of logic, namely the existence of a model of a formula satisfying certain criteria (modulo a background theory). In higher-order logic programming (e.g. Nadathur and Miller [1990], Chen, Kifer, and Warren [1993] and Charalambidis et al. [2013]) it often does not make sense, a priori, to ask about the existence of models because the semantics of the logic programming language is fixed once and for all as part of its definition. One can ask whether the set of answers to a program query is empty, but to understand this as a logical question about the models of a formula is only possible through a result such as our contribution in Section 4. In contrast to (ii), our goal is to develop techniques, such as the type system in Section 5, for the dual problem, namely to show that a Horn formula is satisfiable (recall that we require the existence of a model refuting the goal G, and the goal is the negation of a Horn clause $G \Rightarrow \mathsf{false}$). Query answering (unsatisfiability) is recursively enumerable, whereas satisfiability, in particular where there is a background theory, is typically harder.

Recent work on extensional semantics for higher-order logic programming started with Wadge [1991] and continued with, for example, Charalambidis et al. [2013] and Charalambidis, Ésik, and Rondogiannis [2014]. Wadge was the first to observe that relational variables appearing as arguments in the head of clauses is problematic. This line of work gives a denotational semantics to higher-order logic programs and so is very closely related to the monotone semantics of logic programs that we use in Section 4, except that there is no treatment of constraint theories (discussed in the following paragraph). Unlike our work, Charalambidis et al. [2013] are very careful to ensure the algebraicity of their domains so that they can build a sophisticated system of query answering based on enumerating compact elements. If we were to extend our work to encompass a treatment of counterexamples, then we would want to exploit algebraicity in a similar manner. Their work

does not make any connection between their denotational semantics of higher-order logic programs and satisfiability in standard higher-order logic; it seems likely that our result could be adapted to their setting.

Higher-order constraint logic programming. Another way in which all of the foregoing work differs from our own is that, in the work we have mentioned so far, there is no treatment of constraints. Whilst first-order constraint logic programming is a very well developed area (an old, but good survey is that of Jaffar and Maher [1994]), there is very little existing work on the higher-order extension. Lipton and Nieva [2007] give a Kripke semantics for a λProlog-like language extended with a generic framework for constraints. In contrast to our work, the underlying higher-order logic is intuitionistic and the precise notion of model is bespoke to the paper.

Interpretations of higher-order logic. Under the standard semantics, an interpretation of a higher-order theory consists of a choice of universe A_ι in which to interpret the sort of individuals ι and an interpretation of the constants of the theory. In particular, the interpretation $\mathcal{S}[\![\sigma_1 \to \sigma_2]\!]$ of any arrow sort $\sigma_1 \to \sigma_2$ is fixed by the choice of A_ι. In Henkin (or general) semantics, an interpretation consists of all of the above but, additionally, also a choice of interpretation for each of the infinitely many arrow sorts (under some natural restrictions regarding definability of elements). For example, there are Henkin interpretations in which the collection $[\![\mathsf{int} \to o]\!]$ does not contain all sets of integers. We choose to frame the higher-order constrained Horn clause problem using standard semantics because it is already established in verification. For example, when monadic second order logic (MSO) is used to express verification problems on transition systems, the second-order variables range over all sets of the states[11]. Consequently, the standard semantics seems the most appropriate starting point for work on higher-order constrained Horn clauses in verification.

Automated verification of functional programs. Two of the most well-studied approaches to the automated verification of functional programs are based on higher-order model checking [Kobayashi 2013; Kobayashi and Ong 2009; Ong 2006] and refinement types [Jhala et al. 2011; Rondon et al. 2008; Unno et al. 2013; Vazou et al. 2015; Zhu and Jagannathan 2013]. A method to verify higher-order programs more directly using first-order constrained Horn clauses has been suggested by Bjørner, McMillan, and Rybalchenko [2013a].

In higher-order model checking, the problem of verifying a higher-order program is reformulated as the problem of verifying a property of the tree generated by a higher-order recursion scheme, see e.g. Kobayashi et al. [2011]. The approach has the advantage of being based around a natural, decidable problem, which is an attractive target for the construction of efficient solvers [Broadbent, Carayol, Hague, and Serre 2013; Broadbent and Kobayashi 2013; Ramsay, Neatherway, and Ong 2014]. By contrast, we propose to investigate higher-order program verification based around the higher-order constrained Horn clause problem. Although this problem is generally undecidable, it has the advantage of being able to express (background theory) constraints directly and so has the potential to be a better setting in which to search for higher-order program invariants.

In approaches based on refinement types, a type system is used to the reduce the problem of finding an invariant for the higher-order program, to finding a number of first-order invariants of the ground-type data at certain program points, which can often be expressed as first-order constrained Horn clause solving. As exemplified by the LiquidHaskell system of Vazou, Seidel, Jhala, Vytiniotis, and Peyton-Jones [2014], one advantage of using a type system directly is that it can very naturally encompass all the features of modern programming languages. We have not

[11] Or all finite sets of states in the case of weak MSO, but in both cases the domain is fixed by the setting. A Henkin semantics formulation of the problem would allow for a solution to the satisfiability problem to specify its own domain for the second order variables.

addressed the problem of how best to frame a higher-order program verification problem using higher-order clauses (excepting our motivating sketch in the introduction) but it does not seem as clear as for approaches using refinement types. On the other hand, the reduction to first-order invariants that underlies refinement type approaches has a cost in expressibility. In principle, it is possible to overcome this deficiency, for example by employing types in which higher-order invariants can be encoded as first-order statements of arithmetic [Unno et al. 2013]; we mention also the scheme of Bjørner et al. [2013a] which proposes to view higher-order invariants as first-order statements about closures (encoded as data structures). However, in both cases it seems plausible that working directly in higher-order logic may lead to the development of more transparent and generic techniques. To benefit from this would necessitate a move to a different technology from our system of refinement types used for solving.

7 CONCLUSION AND FUTURE WORK

In this work, we have presented our notion of higher-order constrained Horn clauses and the first foundational results, with an emphasis on making connections to existing work in the verification of higher-order programs. By analogy with the situation for first-order program verification, we believe that higher-order constrained Horn clauses can be an attractive, programming-language independent setting for developing automated techniques for the verification of higher-order programs. For example, such constraints could be used to more directly and more accurately express verification conditions (e.g. arising from a type system or control/data flow analysis) associated with higher-order programs. Let us conclude by giving some more justification to this belief through a discussion of future work.

We have shown, in Example 5.5, that our method for solving is bound by the same limitations as typical refinement type systems in the literature. However, there is a lot of scope to develop new approaches to solving which may not suffer in the same way. A general result in this direction would be to show that these limitations are not intrinsic to the higher-order constrained Horn clause problem. For example, it seems plausible that higher-order clauses are expressive (in the sense of Cook) for suitable Hoare logics over higher-order programs, mirroring the case at first-order [Bjørner et al. 2015; Blass and Gurevich 1987]. A specific technique for solving, which we plan to pursue, is an approach for reducing higher-order clauses to first-order clauses with datatypes, using the ideas of Bjørner et al. [2013a] (which is itself in the spirit of the defunctionalisation of Reynolds [1972]).

In defunctionalisation, as in our method of Section 5, the goal is to reduce the problem to that of first-order clauses by doing some reasoning about the behaviour of higher-order functions. Without further investigation, it is unclear whether this reduction should happen at the level of programs (as is the case in e.g. [Bjørner et al. 2013a]) or at the level of a higher-order intermediate representation such as higher-order constrained Horn clauses (as we have suggested in the introduction).

There is good reason to believe that this reasoning is generic, in the sense that it is essentially the same whether it is an ML or a Haskell program being verified or whether the desired safety property is about avoiding pattern match exceptions or array out of bounds errors. For example, there are well known correspondences between methods for analysing higher-order control flow, such as between type systems and flow analyses, see the work of Heintze [1995] and Palsberg [1998]. This suggests that it is worthwhile to investigate whether this reasoning can be done once and for all on the intermediate representation, rather than be re-implemented in each different analysis of each different higher-order programming language. This way, improvements to efficiency of the higher-order reasoning in the higher-order constrained Horn clause solver can be shared by all verification tools that use it. In short, these are the advantages that derive from a separation of concerns.

Finally, we make the observation that higher-order constraints may be useful even in the verification of first-order procedures. For example, in refinement type systems, a refinement of a base type is typically a predicate on values of the type. Therefore, it seems reasonable that a refinement of a type *constructor* should be a (Boolean-valued) function on predicates, i.e. a higher-order relation. It would be interesting to develop a type inference algorithm that can reason about refinements of type constructors (a concern that is orthogonal to the existence of higher-order procedures) using higher-order constraints, and to understand the connections with the work of Vazou, Rondon, and Jhala [2013].

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees and Wied Pakusa for many suggestions that helped improve the presentation of the paper. The second and third authors gratefully acknowledge the financial support of EPSRC through grant EP/M023974/1. The first author gratefully acknowledges the support of Merton College through their Summer Projects Scheme. Part of this research was done while visiting the Institute for Mathematical Sciences, National University of Singapore in 2016.

REFERENCES

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 869–882.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. 174–184.

Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. 24–51.

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2012. Program Verification as Satisfiability Modulo Theories. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*. 3–11.

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013a. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. *CoRR* abs/1306.5264 (2013).

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013b. On Solving Universally Quantified Horn Clauses. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 105–125.

Andreas Blass and Yuri Gurevich. 1987. Computation Theory and Logic. Springer-Verlag, London, UK, UK, Chapter Existential Fixed-point Logic, 20–36.

Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. 2013. C-SHORe: a collapsible approach to verifying higher-order programs. In *International Conference on Functional Programming, ICFP'13*. ACM, 13–24.

Christopher H. Broadbent and Naoki Kobayashi. 2013. Saturation-based model checking of higher-order recursion schemes. In *Computer Science Logic, CSL'13 (LIPIcs)*, Vol. 23. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 129–148.

Angelos Charalambidis, Zoltán Ésik, and Panos Rondogiannis. 2014. Minimum Model Semantics for Extensional Higher-order Logic Programming with Negation. *TPLP* 14, 4-5 (2014), 725–737.

Angelos Charalambidis, Konstantinos Handjopoulos, Panagiotis Rondogiannis, and William W. Wadge. 2013. Extensional Higher-Order Logic Programming. *ACM Trans. Comput. Log.* 14, 3 (2013), 21.

Weidong Chen, Michael Kifer, and David S. Warren. 1993. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming* 15, 3 (1993), 187–230.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 549–551.

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 343–361.

Nevin Heintze. 1995. Control-Flow Analysis and Type Systems. In *Proceedings of the Static Analysis Symposium, SAS'95 (Lecture Notes in Computer Science)*. Springer, 189–206.

Nevin Heintze, Spiro Michaylov, and Peter Stuckey. 1992. CLP(âĎIJ) and some electrical engineering problems. *Journal of Automated Reasoning* 9, 2 (1992), 231–260.

Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. 2011. μZ- An Efficient Engine for Fixed Points with Constraints. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 457–462.

Joxan Jaffar and Michael J. Maher. 1994. Constraint Logic Programming: A Survey. *J. Log. Program.* 19/20 (1994), 503–581.

Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying Functional Programs Using Abstract Interpreters. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 470–485.

Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), 20:1–20:62.

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.

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Programming Languages Design and Implementation, PLDI'11*. ACM, 222–233.

James Lipton and Susana Nieva. 2007. Higher-Order Logic Programming Languages with Constraints: A Semantics. In *Typed Lambda Calculi and Applications: 8th International Conference, TLCA 2007, Paris, France,June 26-28, 2007. Proceedings*, Simona Ronchi Della Rocca (Ed.). Springer Berlin Heidelberg.

Saunders Mac Lane. 1971. *Categories for the Working Mathematician*. Springer.

Gopalan Nadathur and Dale Miller. 1990. Higher-Order Horn Clauses. *J. ACM* 37, 4 (1990), 777–814.

C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *Logic In Computer Science, LICS'06*. IEEE Computer Society, 81–90.

Jens Palsberg. 1998. Equality-based flow analysis versus recursive types. *ACM Trans. Program. Lang. Syst.* 20, 6 (1998), 1251–1264.

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.

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72)*. 717–740.

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.

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.

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.

Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. 2013. Automating Relatively Complete Verification of Higher-order Functional Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 75–86.

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.

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.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. 269–282.

William W. Wadge. 1991. Higher-Order Horn Logic Programming. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*. 289–303.

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.