A Type-Directed Abstraction Refinement Approach to Higher-Order Model Checking

Steven J. Ramsay

University of Oxford

steven.ramsay@cs.ox.ac.uk

Robin P. Neatherway

University of Oxford

robin.neatherway@cs.ox.ac.uk

C.-H. Luke Ong

University of Oxford

luke.ong@cs.ox.ac.uk

Abstract

The trivial-automaton model checking problem for higher-order recursion schemes has become a widely studied object in connection with the automatic verification of higher-order programs. The problem is formidably hard[1]: despite considerable progress in recent years, no decision procedures have been demonstrated to scale robustly beyond recursion schemes that comprise more than a few hundred rewrite rules. We present a new, fixed-parameter polynomial time algorithm, based on a novel, type directed form of abstraction refinement in which behaviours of a scheme are distinguished by the abstraction according to the intersection types that they inhabit (the properties that they satisfy). Unlike other intersection type approaches, our algorithm reasons both about acceptance by the property automaton and acceptance by its dual, simultaneously, in order to minimize the amount of work done by converging on the solution to a problem instance from both sides. We have constructed PREFACE, a prototype implementation of the algorithm, and assembled an extensive body of evidence to demonstrate empirically that the algorithm readily scales to recursion schemes of several thousand rules, well beyond the capabilities of current state-of-the-art higher-order model checkers.

Categories and Subject Descriptors   F [*3*]: 1;   D [*2*]: 4

Keywords   higher-order model checking; intersection types; abstraction refinement

1. Introduction

Higher-order model checking, or the model checking problem for trees generated by higher-order recursion schemes (HORS), is a widely studied decision problem in connection with the theory and practice of the verification of higher-order programs. Since HORS are simultaneously very expressive [22], algorithmically well-behaved [21], and able to accurately model higher-order control flow [8], they are an appealing target for algorithmic verification procedures for functional programs [12, 16, 17, 23, 27]. Indeed, in a precise sense, HORS are the higher-order analogue of Boolean programs, which have played a very successful rôle in the verification of first order, imperative programs [1].

It is for these reasons, and despite the severe worst-case complexity of the problem[1], that several ingenious algorithms [3, 4, 12, 13, 20] have recently been developed with the aim of solving the higher-order model checking problem for many "practical" instances. However, the state of this effort is summarised well by the authors of [3]:

> "The state-of-the-art model checker TRECS [12] can handle a few hundred lines of HORS generated from various program verification problems. It is, however, not scalable enough to support automated verification of thousands or millions of lines of code. Thus, obtaining a better higher-order model checker is a grand challenge in the field..."

Our main contribution is a new algorithm for higher-order model checking and a large body of evidence to show empirically that it scales well to HORS consisting of several thousand rules. In contrast, the largest instances considered in the literature to date are of the order of several hundred rules. By way of an example, the order-2 benchmark $\mathcal{G}_{2,10000}$ of Kobayashi [12], which consists of 10006 rules, can be processed by our prototype implementation in less than one minute.

Our algorithm, which decides the HORS model checking problem with respect to alternating trivial tree automata, has been designed to be scalable. Since the inherent worst-case complexity of HORS model checking is extreme, to have any chance at all of solving non-trivial instances, one has to work in the belief that those instances that are met in practice are not pathological. Hence, it is essential to ensure that only work that is *relevant* to deciding the particular instance at hand is actually computed. To help achieve this goal, our algorithm is designed in the abstraction refinement paradigm [6]. Initially a relatively cheap but coarse-grained approximation to the problem is processed and, as much as possible, detail is only added by successive iterations where the problem instance necessitates it. Moreover, it can be shown that our algorithm is *fixed-parameter polynomial time* in the size of the scheme; the parameters that are fixed are the order and arity of the scheme, and the size of the tree automaton.

Our algorithm exploits the characterisation of higher-order model checking as an intersection type inference problem [11, 14], representing the state of knowledge about the behaviours of the recursion scheme as a pair of type environments, called the *context*, which assigns intersection types to the non-terminals of the scheme. As the algorithm progresses, the number of types (and hence state of knowledge) in the environments increases, until after some finite number of iterations there will be enough type information to decide the property one way or the other. Furthermore, this limit context will form a certificate of the decision that is independently verifiable by intersection type checking.

---

[1] n-EXPTIME complete for recursion schemes of order n [15, 21]

In order to gain more information and thus populate the context, each iteration consists of constructing a sound abstraction of the configuration graph [12] of the scheme. Since recursion schemes have no facility to inspect the data that they operate over, the behaviours of the scheme arise from the complex interactions between higher-order functions. Hence, we have designed this abstraction around a traditional CFA [10], but with an important twist: in our abstraction, parameters to function calls are distinguished according to the intersection types that they inhabit, in other words, according to the properties that they satisfy. This is, in turn, a function of the context and hence, as the algorithm progresses and the size of the context increases, so the abstractions become more precise, as they are able to distinguish more instances of function calls.

Such an abstract configuration graph is a concise but approximate representation of all the possible reduction sequences of the scheme. Through its analysis, the algorithm can classify certain behaviours that can be seen to generate trees that are accepted by the property automaton and certain other behaviours that can be seen to generate trees that are rejected by the property automaton. From the former it is able to extract new "acceptance" types and from the latter new "rejection" types and both are added to the context ready to proceed with the next iteration. Indeed, a key feature of the algorithm, and a novelty among intersection type based decision procedures, is that it uses types to reason both about property automaton acceptance *and* rejection, simultaneously.

We have implemented the algorithm in a tool, PREFACE, and evaluated its performance over the several hundred problem instances that are now either recorded in the literature or which have resulted from verification tools for higher-order programs. These instances range from a few tens of rules to several thousands and from first order schemes up to fifth order, and a few beyond. The results show very clearly that, whilst PREFACE is sometimes a little slower than other model checkers on examples up to around one hundred rules, its great strength is in solving examples of many hundreds of rules, where it performs consistently better than other model checkers, and several thousands of rules, which are typically instances that it alone can solve. To help analyse the weaknesses of the algorithm, we have also constructed a family of examples which are deliberately difficult for our tool to solve (yet sometimes much easier for other tools reported in the literature).

Outline The rest of the article is structured as follows. In section 2 we fix notation and preliminary definitions. In section 3 we give an informal outline of the algorithm by means of an example. In section 4 the algorithm is defined formally. In section 5 we discuss our prototype implementation and present a digest of the empirical evaluation and associated analysis. In section 6 we discuss related work. All proofs of claims in the text are relegated to the appendix of the long version of this work [25], which also includes a guided run of the algorithm on a second example instance and a full transcript of the empirical evaluation.

2. Preliminaries

We assume throughout a denumerable set $(F, G, H \in) \mathcal{F}$ of *function symbols* and a disjoint, denumerable set $(x, y, z \in) \mathcal{V}$ of *variables*.

Labelled trees Let A be a set without restriction. An *A-labelled tree* is a partial function $T : \mathbb{N}^* \to A$ whose domain is prefix closed. In case the set A is *ranked*, that is, each symbol $a \in A$ has a specified arity $\mathsf{arity}(a) \in \mathbb{N}$, then the tree T can be said to be *well-ranked* just if, whenever $T(w) = a$ and $\mathsf{arity}(a) = n$ then: $w \cdot i \in \mathsf{dom}(T)$ iff $i \in [1..n]$.

Simple kinds. The *simple kinds*[2] over the kind of trees o, denoted $(\kappa \in) \mathbb{S}$, are formed by the grammar $\kappa ::= o \mid \kappa_1 \to \kappa_2$. As usual, we use parentheses to disambiguate the structure of such expressions, observing that the arrow associates to the right. The *arity* and *order* of a simple kind are natural numbers defined as usual. If a simple kind has order 0 (and hence has arity 0) we say that it is *ground*.

Raw terms. Let $(a, b, c \in) \Sigma$ be a set of atomic constants. The set of *raw terms over* Σ, denoted $(s, t, u, v \in) T_\Sigma(\mathcal{F}, \mathcal{V})$, is defined by the grammar:

$$s, t \quad ::= \quad x \mid F \mid c \mid s\, t$$

The *free variables* of a term t, denoted $\mathsf{FV}(t)$, is just the set of variables that occur in t. A term t with $\mathsf{FV}(t)$ empty is called *closed* and the set of all closed terms is denoted $T_\Sigma(\mathcal{F})$. We denote the set of closed terms which, moreover, contain no occurrences of function symbols by T_Σ. In case the atomic constants are said to be *kinded* we assert that there is an associated kinding function kind which maps each constant $c \in \Sigma$ to a first-order kind in $\mathbb{S}$.

Kinded terms. A kind environment Δ, is a finite, partial function from $\mathcal{V} \cup \mathcal{F}$ to $\mathbb{S}$. A *kind judgement* is an expression of the form $\Delta \vdash t : \kappa$. We omit the standard definition of kind assignment to terms, i.e. simple type assignment.

Recursion Schemes. A higher-order recursion scheme (HORS) $\mathcal{G}$ is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ in which:

- The alphabet of *terminal symbols* $(a, b, c \in) \Sigma$ is a finite set of first-order, kinded constants.
- The alphabet of *non-terminal symbols*, $(F, G, H \in) \mathcal{N}$, is a finite set of kinded function symbols, disjoint from Σ. We will sometimes view $\mathcal{N}$ as a kind environment mapping non-terminals $F \in \mathcal{N}$ to their kinds $\mathsf{kind}(F)$.
- The *rewrite rules*, $\mathcal{R}$, comprise a function mapping each non-terminal symbol F of kind $\kappa_1 \to \cdots \to \kappa_n \to o$ to an expression $\lambda x_1 \cdots x_n. t$, such that:

$$x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash t : o$$

  is a provable assignment of kinds to terms. We will often write $(F, \lambda x_1, \ldots, x_n. t) \in \mathcal{R}$ as an equation $F = \lambda x_1, \ldots, x_n. t$.
- The *start symbol*, $S \in \mathcal{N}$, is a non-terminal symbol of kind o.

Each recursion scheme is assigned an *order* which is given by the maximum order of (the kind of) its non-terminal symbols. Recursion schemes have a simple notion of reduction, which is defined as the contextual closure of the following rule:

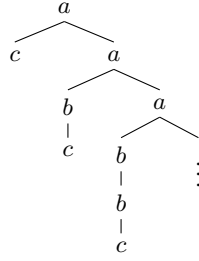$$\frac{\mathcal{R}(F) = \lambda x_1 \cdots x_n. t}{F\, s_1 \cdots s_n \Rightarrow t[s_1/x_1, \ldots, s_n/x_n]}$$

For the purposes of model checking, we are interested in the trees generated by the scheme. The *value tree* of a scheme $\mathcal{G}$, denoted $\mathsf{Tree}(\mathcal{G})$ is the (possibly infinite) term tree obtained by reducing the start symbol *ad infinitum*. To account for the possibility of infinite, unproductive recursion, the value tree is defined as follows. First, introduce a new symbol of zero arity, $\bot$, into Σ and consider the least preordering of Σ that asserts $\bot \leq a$ for all $a \in \Sigma$. Next, for each closed term t of ground type, define $t^\bot$ recursively by: (i) $(F\, s_1 \cdots s_n)^\bot = \bot$ and (ii) $(a\, s_1 \cdots s_n)^\bot = a\, s_1^\bot \cdots s_n^\bot$. This mapping sends each term to well-ranked tree in the complete partial order of $\Sigma \cup \bot$-labelled trees, which are ordered by letting $T_1 \sqsubseteq T_2$ just if, for all $\pi \in \mathsf{dom}(T_1)$, $T_1(\pi) \leq T_2(\pi)$. Finally, we set $\mathsf{Tree}(\mathcal{G}) = \bigsqcup \{t^\bot \mid S \Rightarrow^* t\}$.

---

[2] These are nothing more than the simple types over the base type o, but we prefer to use the word *kind* to avoid conflict with intersection *type* later.

Example 1. Consider the first-order scheme over terminal symbols $a : o \to o \to o$, $b : o \to o$ and $c : o$ in which the non-terminal symbols $S : o$ and $F : o \to o$ are defined by the equations:
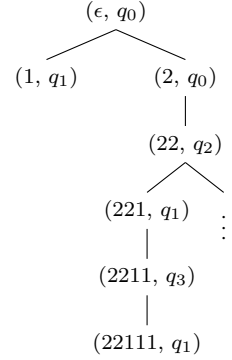
$$
\begin{aligned}
S &= F\ c \\
F &= \lambda x.\ a\ x\ (F\ (b\ x))
\end{aligned}
$$

By reducing *ad infinitum*, the start symbol S generates an infinite a, b and c-labelled tree, a prefix of which is depicted on the right. This tree has no $\bot$-labelled nodes since every redex contraction produces a new terminal symbol in head position.

$$
\begin{array}{c}
a \\
c \qquad a \\
\quad b \qquad a \\
\quad c \quad b \qquad \vdots \\
\qquad b \\
\qquad c
\end{array}
$$

Positive Boolean formulae. Given a finite set X, the *positive Boolean formulas over X*, denoted ($\phi \in$) $\mathsf{B}^+(X)$, are defined by the grammar $\phi ::= \mathsf{t} \mid \mathsf{f} \mid x \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$. Given a positive Boolean formula ϕ, an *assignment* is a finite subset S of X. An assignment S is said to be a *satisfying assignment* for ϕ, written $S \models \phi$, when assigning t to elements of S and f to elements of $X \setminus S$ makes ϕ true.

Alternating (co-)trivial tree automata. An alternating (co-)trivial tree automaton (ATT) $\mathcal{A}$ is a tuple $\langle \Sigma, Q, \delta, q_0, F \rangle$ in which Σ, is a finite set of ranked constants, ($q \in$) Q, is a finite set of *states*, the *transition function*, δ, is a function in $\Pi_{(q,a) \in Q \times \Sigma} \cdot \mathsf{B}^+([1..\mathsf{arity}(a)] \times Q)$, the *initial state* is $q_0 \in Q$ and the *accepting states*, F, are either all of Q or empty. In case $F = Q$, we say that the ATT has a *trivial* acceptance condition, otherwise $F = \emptyset$ and we say that it has a *co-trivial* acceptance condition. More often than not we will simply introduce a given automaton as a trivial or a co-trivial automaton and omit the final component. Furthermore, when specifying particular automata, we will elide clauses of the transition function whose image is f.

Given a Σ-ranked and labelled tree T, a *run tree of $\mathcal{A}$ on T* is a $(\mathsf{dom}(T) \times Q)$-labelled, unranked tree R satisfying:

(APT-1) $R(\epsilon) = (\epsilon, q_0)$

(APT-2) For all $w \in \mathbb{N}^*$, if $R(w) = (w', q)$ then there is some set S that satisfies $\delta(q, T(w'))$ and, for all $(i, q') \in S$, there exists some $j \in \mathbb{N}$ such that $R(w \cdot j) = (w' \cdot i, q')$.

We say that a run tree R is *accepting* just if, on every infinite branch of R, there is some state $q \in F$ which occurs infinitely often. The *language* of an ATT $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of Σ-ranked and labelled trees T for which there exists an accepting run-tree on T. We define the *complement* of $\mathcal{A}$, denoted $\mathcal{A}^c$, by the standard de Morgan dual construction, which ensures that $\mathcal{L}(\mathcal{A})^c = \mathcal{L}(\mathcal{A}^c)$ [19]. Note, the dual of an ATT with a trivial acceptance condition is an ATT with a co-trivial acceptance condition.

Example 2. Consider the ATT over the states q_0, q_1, q_2 and q_3, in which the transition function is defined by the following clauses:

$$
\begin{aligned}
\delta(q_0, a) &= ((1, q_1) \wedge (2, q_0)) \vee (2, q_2) \\
\delta(q_1, b) &= (1, q_3) \\
\delta(q_1, c) &= \mathsf{t} \\
\delta(q_2, a) &= (1, q_1) \wedge (2, q_0) \\
\delta(q_3, b) &= (1, q_1)
\end{aligned}
$$

This ATT accepts those trees that have an infinite a-labelled spine and of every two consecutive branches off the spine, at least one is required to be labelled by an even number of b nodes terminated by a c. A prefix of the run tree over the tree generated by the scheme in Example 1 is depicted to the right.

$$
\begin{array}{c}
(\epsilon, q_0) \\
(1, q_1) \qquad (2, q_0) \\
\qquad (22, q_2) \\
\quad (221, q_1) \qquad \vdots \\
\quad (2211, q_3) \\
\quad (22111, q_1)
\end{array}
$$

Higher-order model checking. We define $\mathcal{A}^\bot$ as the ATT $\mathcal{A}$ augmented with extra transitions so as to accept the symbol $\bot$ from every state. The ATT-model checking problem for HORS is, given a HORS $\mathcal{G}$ and an ATT $\mathcal{A}$, to determine the truth of the assertion $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\bot)$.

Intersection types. In what follows fix an ATT $\mathcal{A}$. We consider *intersection types* [7]. As is usual in the higher-order model checking literature, we make a distinction between strict types and intersection types (borrowing the terminology from van Bakel [28]). The *intersection types* over $\mathcal{A}$, denoted $\mathbb{I}_\mathcal{A}$, are defined simultaneously with the *strict types* over $\mathcal{A}$ by the following grammar:

$$
\begin{array}{lll}
(\text{STRICT TYPES}) & \tau & ::= \quad q \mid \sigma \to \tau \\
(\text{INTERSECTION TYPES}) & \sigma & ::= \quad \bigwedge_{i=1}^n \tau_i
\end{array}
$$

in which $q \in Q$ and $n \geq 0$. We will use τ and σ to denote strict and intersection types respectively. When we are agnostic about whether a particular expression is either a strict type or an intersection type we will say it is simply a *type* and denote it by θ. When no confusion can arise, we will typically write $\top$ for the empty intersection, an intersection containing two elements infix as $\tau_1 \wedge \tau_2$ and an intersection of the singleton set containing τ simply as τ. Given an intersection $\sigma = \bigwedge_{i=1}^n \tau_i$ we will often identify σ with its set of conjuncts $\{\tau_1, \ldots, \tau_n\}$, writing assertions such as $\tau \in \sigma$ and $\sigma_1 \subseteq \sigma_2$ with the obvious interpretation. Finally, we shall have no qualms about constructing the intersection of intersection types, since this can be given naturally as the intersection of the union over their respective strict conjuncts.

Intersection subtyping. There is a natural subtype preorder on intersection types, which was first explicitly considered by Barendregt, Coppo and Dezani-Ciancaglini in [2]. We shall use the following variant, defined inductively by the following clauses.

(Q-BAS) $q \leq q$

(Q-ARR) if $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$ then $\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$

(Q-PRJ) for all $i \in [1..n]$, $\bigwedge_{j=1}^n \tau_j \leq \tau_i$

(Q-GLB) if, for all $i \in [1..n]$, $\sigma \leq \tau_i$, then $\sigma \leq \bigwedge_{j=1}^n \tau_j$

(Q-TRS) if $\theta_1 \leq \theta_2$ and $\theta_2 \leq \theta_3$ then $\theta_1 \leq \theta_3$

Intersection type environment. An *intersection type environment* Γ is a finite, partial function from $\mathcal{F} \cup \mathcal{V}$ to $\mathbb{I}_\mathcal{A}$. We will often view type environments as total functions assigning $\Gamma(F) = \top$ whenever $F \notin \mathsf{dom}(\Gamma)$. We will write $\Gamma_1 \uplus \Gamma_2$ for the operation sometimes called *type environment multiplication*, which is just the pointwise combination of environments defined by:

$$
(\Gamma_1 \uplus \Gamma_2)(F) = \Gamma_1(F) \wedge \Gamma_2(F)
$$

$$\frac{}{\Gamma, x : \bigwedge_{i=1}^{n} \tau_i \vdash x : \tau_i} \text{ (T-VAR)}$$

$$\frac{}{\Gamma, F : \bigwedge_{i=1}^{n} \tau_i \vdash F : \tau_i} \text{ (T-FUN)}$$

$$\frac{S \models \delta(q, c)}{\Gamma \vdash c : \bigwedge(S|_1) \to \cdots \to \bigwedge(S|_n) \to q} \text{ (T-CST)}$$

$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \tau' \; [\forall \tau' \in \sigma'] \quad \sigma' \leq \sigma}{\Gamma \vdash s \, t : \tau} \text{ (T-APP)}$$

Figure 1. Assignment of types to terms.

$$\frac{}{q :: o} \text{ (K-BAS)}$$

$$\frac{\sigma :: \kappa_1 \qquad \tau :: \kappa_2}{\sigma \to \tau :: \kappa_1 \to \kappa_2} \text{ (K-ARR)}$$

$$\frac{\tau_i :: \kappa \; (i \in [1..n])}{\bigwedge_{i=1}^{n} \tau_i :: \kappa} \text{ (K-INT)}$$

Figure 2. Assignment of kinds to types.

and write $\Gamma_1 \not\subseteq \Gamma_2$ just if there is some Γ' and $\Gamma_1 \uplus \Gamma' = \Gamma_2$. We will write $\Gamma \restriction X$ for the restriction of Γ to only those typings whose subject lies in X. Finally, we also extend the subtype relation to environments pointwise, writing $\Gamma_1 \leq \Gamma_2$ just if, $\text{dom}(\Gamma_2) \subseteq \text{dom}(\Gamma_1)$ and, for all $\xi \in \text{dom}(\Gamma_2)$, $\Gamma_1(\xi) \leq \Gamma_2(\xi)$.

Intersection type assignment. An *intersection type judgement* is an expression of the form $\Gamma \vdash t : \tau$ (with τ a strict type) whose derivations are defined inductively by the system in Figure 1. Note that in that system, we use the notation $S|_i$ to denote the set $\{q \mid (i, q) \in S\}$. Given a type environment Γ and a term t, we define the set of all strict types assignable to t under Γ by:

$$\mathbb{T}(\Gamma)(t) = \{\tau \mid \Gamma \vdash t : \tau\}$$

The type system is induced by the property automaton $\mathcal{A}$, which features in the premise to the rule (T-CST). This rule acts to give a meaning to intersection types that can be thought of as follows. Each base type q is the type of all terms t that generate (via infinite reduction) trees that are accepted by the automaton from state q. An intersection such as $q_1 \wedge q_2$, is the type of all terms that generate trees that are accepted from state q_1 and from state q_2. Finally, an arrow such as $q_1 \wedge q_2 \to q$ is the type of those terms which, when applied to a term that generates a tree accepted from q_1 and q_2, will, as an application, generate a tree accepted from state q.

In this work we will be concerned both with type assignment in the intersection type system induced by property automaton $\mathcal{A}$ and type assignment in the intersection type system induced by the dual of this automaton $\mathcal{A}^c$. Hence, whenever needed we will try to disambiguate which notion of type assignment we are referring to by annotating the notation with $\mathcal{A}$ or $\mathcal{A}^c$.

Intersection refinement types. The *intersection refinement types* over Q are those types θ for which there is some kind κ such that $\theta :: \kappa$, pronounced "σ refines κ", is provable in the system of kind assignment below. The *strict refinement types* over Q are defined as the obvious restriction of this system. We lift the refinement relation to environments by writing $\Gamma :: \Delta$ just if, for all $F : \sigma \in \Gamma$, there is a typing $F : \kappa \in \Delta$ and $\sigma :: \kappa$.

Type environment consistency. We say that an intersection type environment Γ is $(\mathcal{G}, \mathcal{A})$-*consistent* just if, for each typing $F : \sigma \in \Gamma$ such that $F \in \text{dom}(\mathcal{N})$, there is a *possibly infinite* witness, rooted at $\Gamma \triangleright F : \sigma$, and built according to the following system:

$$\mathcal{R}(F) = \lambda x_1 \dots x_n. \, t$$
$$\frac{\Gamma, x_1 : \sigma_1, \dots, x_1 : \sigma_n \vdash t : q \qquad \Gamma \triangleright G : \sigma \; (\forall G : \sigma \in \Gamma)}{\Gamma \triangleright F : \sigma_1 \to \cdots \to \sigma_n \to q}$$

$$\frac{\Gamma_1 \not\subseteq \Gamma \quad \Gamma_1 \triangleright F : \tau_1 \quad \cdots \quad \Gamma_n \not\subseteq \Gamma \quad \Gamma_1 \triangleright F : \tau_n}{\Gamma \triangleright F : \bigwedge_{i=1}^{n} \tau_i}$$

Similarly, we say that an intersection type environment Γ is $(\mathcal{G}, \mathcal{A})$-*co-consistent* just if, for each typing $F : \sigma \in \Gamma$ there is a *strictly finite* witness built from the above system. The next theorem follows from Kobayashi and Ong [14].

Theorem 1. *Fix a scheme $\mathcal{G}$ and ATT $\mathcal{A}$.*

(i) $\text{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$ *iff there exists* $(\mathcal{G}, \mathcal{A})$-*consistent* $\Gamma :: \mathcal{N}$ *and* $q_0 \in \Gamma(S)$.

(ii) $\text{Tree}(\mathcal{G}) \in \mathcal{L}((\mathcal{A}^\perp)^c)$ *iff there exists* $(\mathcal{G}, \mathcal{A}^c)$-*co-consistent* $\Gamma :: \mathcal{N}$ *and* $q_0 \in \Gamma(S)$.

3. Type directed abstraction refinement

The starting point for the algorithm is the characterisation of the trivial automaton model checking problem for recursion schemes given in Theorem 1. Our algorithm tries to prove (a) $\text{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$ and (b) $\text{Tree}(\mathcal{G}) \in \mathcal{L}((\mathcal{A}^\perp)^c)$ simultaneously, by iteratively constructing two type environments $\Gamma_\exists$ and $\Gamma_\forall$ which are possible witnesses to (a) and (b) respectively. Since an invariant of the algorithm is that $\Gamma_\exists$ is always $(\mathcal{G}, \mathcal{A})$-consistent and $\Gamma_\forall$ is always $(\mathcal{G}, \mathcal{A}^c)$-co-consistent, it follows that at most one of the two environments can prove the type assignment $S : q_0$; in fact, upon termination, exactly one of the two will do so. Since the two type systems $\mathcal{A}$ and $\mathcal{A}^c$ share the same underlying set of types, let us call an intersection type an *acceptance type* when we regard it as part of the system $\mathcal{A}$ and let us call it a *rejection type* when we regard it as part of the system $\mathcal{A}^c$.

The algorithm starts with $\Gamma_\exists^0 = \Gamma_\forall^0 = \emptyset$, which is trivially (co-)consistent. On each iteration, new type assignments are inferred which will be added to one or the other of the environments. The way that these new types are inferred is by, on each iteration, constructing and interrogating an auxiliary structure, called the *abstract configuration graph*. As its name suggests, this graph is an abstraction and the precision of the abstraction is a function of the size of the type environments $\Gamma_\exists$ and $\Gamma_\forall$. The more precise the abstraction the more useful the type information that can be deduced by interrogating it. Hence, starting from the empty *context*, $C_0 = \langle \Gamma_\exists^0, \Gamma_\forall^0 \rangle$, the abstraction refinement cycle continues as follows. In iteration $i + 1$ with context $C_i = \langle \Gamma_\exists^i, \Gamma_\forall^i \rangle$, the abstract configuration graph is constructed. Two subgraphs are then carved out called the accepting and rejecting regions. These regions are the parts of the graph from which it is possible to obtain consistent acceptance typings and co-consistent rejection typings respectively. Types are then extracted from the regions and added to context C_i to form a strictly larger context $C_{i+1} = \langle \Gamma_\exists^{i+1}, \Gamma_\forall^{i+1} \rangle$. If one of these two environments can already type $S : q_0$ then the algorithm terminates. Otherwise the cycle repeats and, since C_{i+1} is strictly larger, the abstract configuration graph constructed in iteration $i+2$ will be strictly more precise, and new type information will be deduced. Since there are only finitely many intersection refinement
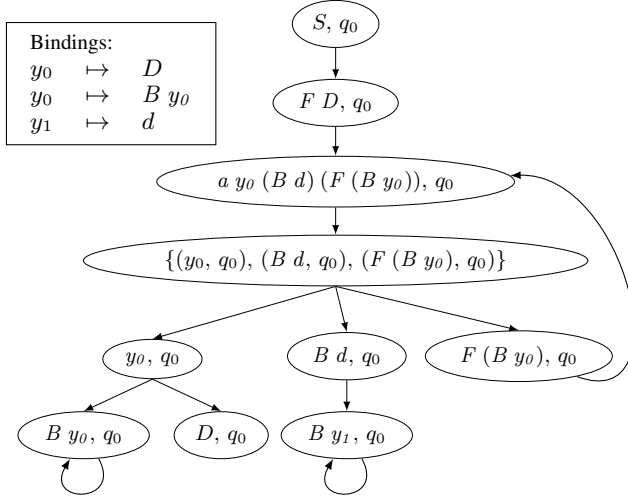
Bindings:
$y_0 \mapsto D$
$y_0 \mapsto B\, y_0$
$y_1 \mapsto d$

S, q_0

$F\, D, q_0$

$a\, y_0\, (B\, d)\, (F\, (B\, y_0)), q_0$

$\{(y_0, q_0), (B\, d, q_0), (F\, (B\, y_0), q_0)\}$

y_0, q_0

$B\, d, q_0$

$F\, (B\, y_0), q_0$

$B\, y_0, q_0$

D, q_0

$B\, y_1, q_0$

Figure 3. An abstract configuration graph.

types associated with a given scheme one of the environments will be eventually become saturated and thus witness the corresponding assertion (a) or (b).

The configuration graph of a model checking problem instance was introduced by Kobayashi and Ong in [12, 14]. It is a kind of product construction, pairing up the reduction relation of the scheme and the transition function of the property automaton and it takes the shape of a rooted, directed graph. Since we are solving the alternating trivial automaton model checking problem for recursion schemes, the configuration graphs that we are interested in are something in between the simple ones of [12], which are appropriate to deterministic trivial automaton properties, and the much more complicated ones of [14], which are appropriate to alternating parity automaton properties. Moreover, we shall not be interested in the configuration graphs themselves, which are in general infinite, but rather in finite abstractions.

Consider the following model checking instance, consisting of recursion scheme $\mathcal{G}$ over the terminal symbols a and d and a two-state trivial automaton $\mathcal{A}$:

$$
\begin{aligned}
S &= F\, D & \delta(q_1, d) &= \mathsf{t} \\
F &= \lambda x.\, a\, x\, (B\, d)\, (F\, (B\, x)) & \delta(q_0, a) &= (1, q_0) \\
B &= \lambda z.\, B\, z & &\wedge (2, q_0) \\
D &= d & &\wedge (3, q_0)
\end{aligned}
$$

As described above, the algorithm begins with the initial context $C_0 = \langle \Gamma_\exists^0, \Gamma_\forall^0 \rangle$ consisting of $\Gamma_\exists^0 = \emptyset$ and $\Gamma_\forall^0 = \emptyset$. In order to infer new type assignments it constructs the abstract configuration graph $\mathsf{ACG}(C_0)$, which is given in full in Figure 3. The construction of the graph starts at the root, which is a vertex labelled (S, q_0). Every vertex of the graph is either labelled by a pair (t, q) of a ground kind term and a state, called a *configuration*, or by a finite set of such configurations. If there is a vertex (t, q) in a graph $\mathsf{ACG}(\langle \Gamma_\exists, \Gamma_\forall \rangle)$, then it should be read as "neither $\Gamma_\exists \vdash_{\mathcal{A}} t : q$ nor $\Gamma_\forall \vdash_{\mathcal{A}^c} t : q$ is provable". Hence, the starting point for the graph $\mathsf{ACG}(C_0)$ is the fact that $S : q_0$ is not provable in $\emptyset$.

The shape of the successors of a vertex, if it has any, depend upon whether the vertex is a configuration or a set and in case of the former, the syntactic class of which its head symbol is a member. In this case the root of the graph is a configuration and its head symbol is S which is a non-terminal. Hence the term part of this configuration is a redex. In such cases, the vertex has at most one successor and that successor *represents* the contraction of the redex. The contraction of the redex S is $F\, D$ and, because neither $\Gamma_\exists^0 \vdash_{\mathcal{A}} F\, D : q_0$ nor $\Gamma_\forall^0 \vdash_{\mathcal{A}^c} F\, D : q_0$ is provable, it has

a successor, which is labelled by $(F\, D, q_0)$. Since this vertex is of the same form, it also has a single successor which represents the contraction of the redex, however the situation is more complicated because this redex involves parameters. It is along such edges that the abstraction happens: rather than substituting actual parameters for formals in the successor, formals are substituted by special variables which are used by the abstraction to represent sets of terms. So we create a new variable y_0 and make a note $y_0 \mapsto D$ that one of the possible instantiations of y_0 is D. In this way, we are always able to recover the real contraction $a\, D\, (B\, d)\, (F\, (B\, D))$ from the abstract one $a\, y_0\, (B\, d)\, (F\, (B\, y_0))$ by rewriting occurrences of the variables y_0 using the *bindings* $y_0 \mapsto D$. This ensures we are building a *sound* abstraction, in the sense of incorporating all the behaviours of the original. This kind of abstraction is, in essence, a traditional control flow analysis [10], but here we go one step further, which is critical in order to obtain completeness. The extra step that we take is to record the acceptance type and rejection type of the new variable y_0. The acceptance type (respectively rejection type) of y_0 is just the intersection of types assignable to the term that it has been introduced to represent, in other words, $\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists^0)(D) = \top$ (respectively $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall^0)(D) = \top$). The types of such variables will later determine when new variables should be created in order to represent actual parameters, as y_0 was here, or if variables created previously should be reused.

The frontier of our construction so far consists of a terminal headed configuration. The successors of a configuration of the form $(a\, s_1\, \cdots\, s_n, q)$ depend upon the satisfying assignments to $\delta(q, a)$. For each satisfying assignment S there is one successor, which is the set of configurations $\{(s_i, q') \mid (i, q') \in S\}$. In this case, $\delta(q, a)$ is satisfied just if the first argument of a is accepted from q_0, the second argument is accepted from q_0 and the third argument is accepted from q_0. Consequently, the set is $\{(y_0, q_0), (B\, d, q_0), (F\, (B\, y_0), q_0)\}$. Such a set $\{(t_1, q_1), \ldots, (t_n, q_n)\}$ should be read as "there is some i such that $\Gamma_\exists \vdash_{\mathcal{A}} t_i : q_i$ is not provable". The successors of the set are those configurations (t, q) in the set which are not provable in either $\Gamma_\exists$ or $\Gamma_\forall$.

So let us now consider the frontier configuration $(F\, (B\, y_0), q_0)$. Since it is a non-terminal headed configuration, if it has a successor then the successor abstractly represents the contraction in the same way as before. However, since the actual parameter $B\, y_0$ has the same acceptance type and rejection type as the previous one that we considered, D (and is the same kind), we will not create a new variable to represent $B\, y_0$, but we will *reuse* the variable y_0 that we used to represent actual parameter D. So the abstract contraction is the term $a\, y_0\, (B\, d)\, (F\, (B\, y_0))$ and we form a loop in the graph. To ensure the abstraction remains sound we note that another possible instantiation of y_0 is $B\, y_0$, but observe that this merging of vertices resulting from reusing y_0 has caused some loss of information. Now the term $a\, y_0\, (B\, d)\, (F\, (B\, y_0))$ represents many different concrete instances, including some that are not possible in the original problem, such as $a\, D\, (B\, d)\, (F\, (B\, (B\, D)))$ which results from rewriting the leftmost occurrence of y_0 to D and the rightmost occurrence to $B\, D$. The actual parameters D and $B\, y_0$ have become confused because, according to the current context C_0, they have the same acceptance and rejection types. In this way, *types direct the abstraction* and refinement will occur because type information in the context increases.

By contrast, the vertex $(B\, d, q_0)$ has an actual parameter whose acceptance and rejection types are non-trivial. The terminal d has acceptance type $\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists^0)(d) = q_1$ and rejection type $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall^0)(d) = q_0$. We have not yet created a new variable with those types, so we do so now, assigning the new variable y_1 the acceptance type q_1 and the rejection type q_0. We record that d is a possible instantiation of y_1 and label the successor $(B\, y_1, q_0)$.

Following the same method, this new vertex has itself as successor. Officially we ought to note that y_1 is a possible instance of y_1, but such trivial circularities will make no difference to the outcome so we will omit it for brevity.

Let us now consider the frontier vertex (y_0, q_0), which is variable headed. At variable headed configurations the consequences of the abstraction are felt. A vertex of the form $(y\, s_1 \cdots s_n, q)$ has a successor $(t\, s_1 \cdots s_n, q)$ for each binding $y \mapsto t$. For reasons that we have already discussed, its child $(B\, y_0, q_0)$ has itself as a successor. Its other child is the vertex (D, q_0) whose contraction is (d, q_0), however, because $\Gamma_\forall^0 \vdash_{\mathcal{A}^c} d : q_0$ we do not add vertex (d, q_0) to the graph and so (D, q_0) has no successors. We say that (D, q_0) is a *rejecting leaf*. This completes the construction of the abstract configuration graph.

We now consider any rejecting leaves in the graph. Rejecting leaves are configurations $(F\, s_1 \cdots s_n, q)$ for which the corresponding type assignment $F\, s_1 \cdots s_n : q$ is not provable in either $\Gamma_\exists$ nor $\Gamma_\forall$, yet the type assignment associated with the contraction is provable in $\Gamma_\forall$. This points directly to a weakness in the context. In our example, $\Gamma_\forall^0 \vdash_{\mathcal{A}^c} d : q_0$ is provable, so d is a tree rejected by $\mathcal{A}^\perp$ and D reduces to d in one step, yet $\Gamma_\forall \vdash_{\mathcal{A}^c} D : q_0$ is not provable! The rejection environment $\Gamma_\forall$ *ought* to type $D : q_0$, since D generates a tree that is rejected by $\mathcal{A}^\perp$, but it doesn't. Hence, we have discovered a new rejection type assignment. The vertices that we classify as representing typing assignments that ought to have been provable under the rejecting environment are collectively called the *rejecting region* and from this region new rejection types are extracted. Where the rejecting region is, roughly speaking, all the vertices that "definitely" can reach rejecting leaves, the *accepting region* is all the vertices that "definitely" cannot reach rejecting leaves. New acceptance types are extracted from the accepting region. In this case, the rejecting region is the single vertex (D, q_0) and the accepting region is all the B headed configurations.

Type extraction from the regions follows a similar approach to that defined in [14]. Briefly, a type is assigned to each prefix s of the term component of each vertex $(s\, t_1 \cdots t_n, q)$ in the accepting region. The type assigned to s is constructed by recursively computing the type σ assigned to t_1 by considering all the ways in which t_1 is used within the region (the vertices where t_1 is itself a prefix), recursively computing the type τ assigned to the prefix $s\, t_1$ and then forging the arrow $\sigma \to \tau$. The base case is where the entire term component of the pair, considered as a trivial prefix of itself, is assigned the type q. The rejecting region is handled similarly, but care must be taken to only consider uses of t_1 in vertices that are strictly closer to the leaves so that a well founded co-consistency argument can be given in the end. The type assignments to prefixes that are themselves non-terminal symbols are extracted and added to the appropriate environments. In this case, there is only one vertex in the rejecting region, so its only prefix is assigned the type $D : q_0$. There are three vertices in the accepting region, of which the prefix B of $(B\, d, q_0)$ is assigned acceptance type $q_1 \to q_0$ since d is known to have type q_0, prefix B of $(B\, y_0)$ is assigned type $\top \to q_0$ since there are no uses of y_0 in the region and, for the same reason, prefix B of $(B\, y_1)$ is assigned the type $\top \to q_0$. The new context is therefore $C_1 = \langle \{B : (\top \to q_0) \wedge (q_1 \to q_0)\}, \{D : q_0\} \rangle$.

Since the new context neither types $S : q_0$ as accepting nor as rejecting, the process repeats, but this time with more type information, so more of the parameters to calls will be distinguished. In the second iteration the rejecting region is the entire graph. Consequently, $S : q_0$ is added to the rejecting environment and the algorithm terminates, correctly deducing that the input is a no-instance.

4. An abstraction refinement algorithm

We now present the algorithm formally. We first give the definition of the key construction, the abstract configuration graph. We then describe how from the graph one can carve out the accepting and rejecting regions and the notion of type extraction appropriate to each. Finally we show how graph construction, regioning and type extraction come together to form a single iteration and we present the algorithm as the repetition of this process.

Assumption. *For the rest of this section, assume a recursion scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ and an alternating trivial automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$. By an abuse, we will refer to the type system induced by $\mathcal{A}$ simply as $\mathcal{A}$ and the type system induced by $\mathcal{A}^c$ simply as $\mathcal{A}^c$.*

4.1 Construction of abstraction

The abstraction is based on a traditional CFA [10], in the sense of over-approximating reduction using an abstract environment. An important twist on the usual formulation is that here every variable in the environment is associated with a kind and a pair of intersection types.

Typed variables. The main mechanism for abstraction will be a set of typed variables. By means of an abstract environment (to be described shortly) each variable represents the set of terms that can be obtained from it by repeated substitution. Each variable has three pieces of associated type information: an acceptance type, a rejection type and a kind. An essential part of the algorithm is in ensuring that type information is invariant across the abstraction, i.e. if a variable abstracts a set of terms, then every term in the set shares the same acceptance type, rejection type and kind as the variable (according to the current context).

Definition 1. *Let us say that a kind κ is an* argument kind *just if there is some binding $F : \kappa_1 \to \cdots \to \kappa_n \to o \in \mathcal{N}$ and some i such that $\kappa = \kappa_i$. Let* var *be a bijection, mapping the finite set of all triples of the form $(\sigma_\mathsf{A}, \sigma_\mathsf{R}, \kappa)$ consisting of kinded types $\sigma_\mathsf{A} :: \kappa$ and $\sigma_\mathsf{R} :: \kappa$,where κ is an argument kind, to a finite set of term variables $\mathcal{Y} \subseteq \mathcal{V}$. Given such a variable $y \in \mathcal{Y}$, we will write $\mathsf{A}(y)$ for the first component of $\mathsf{var}^{-1}(y)$, $\mathsf{R}(y)$ for the second and $\mathsf{K}(y)$ for the third.*

Type context. The algorithm is ultimately concerned with constructing a pair of type environments $\langle \Gamma_\exists, \Gamma_\forall \rangle$ such that $\Gamma_\exists$ is $(\mathcal{G}, \mathcal{A})$-consistent and $\Gamma_\forall$ $(\mathcal{G}, \mathcal{A}^c)$-co-consistent. We will speak of $\Gamma_\exists$ as the "acceptance" type environment and $\Gamma_\forall$ as the "rejection" type environment. Furthermore, we stipulate that every such pair of environments, which we shall call a *type context*, understands the basic assumptions we have made about the typed variables, i.e. the type information contained in A and R (as defined in Definition 1, but viewed as type environments for the typed variables) is also contained in $\Gamma_\exists$ and $\Gamma_\forall$ respectively.

Definition 2. *A* type context *$C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ is a pair of intersection type environments for which the following conditions hold:*

(i) $\Gamma_\exists :: \mathcal{N} \cup \mathsf{K}$
(ii) $\Gamma_\forall :: \mathcal{N} \cup \mathsf{K}$
(iii) For all $y \in \mathcal{Y}$, $\Gamma_\exists(y) = \mathsf{A}(y)$ and $\Gamma_\forall(y) = \mathsf{R}(y)$

Abstract configurations. As mentioned in the previous section, the abstraction itself is a finite representation of the possibly infinite *configuration graph*, as defined by [14]. In this *concrete* configuration graph, the configurations are pairs of a *closed* term (a reduct of the start symbol of the scheme) and a state of the automaton, and the edges that connect them must respect the constraints of both the reduction relation of the scheme and of the transition function of the automaton. A configuration (t, q) can be read as an

assertion: the tree generated by t is accepted by $\mathcal{A}^\perp$ from state q. In the *abstract* configuration graph, defined shortly, configurations are still pairs of term and state, but now the term is abstract, which in our setting means that it can contain free occurrences of typed variables.

Definition 3. *An abstract configuration is a pair (t, q) in which $\mathcal{N} \cup \{y : \mathsf{K}(y) \mid y \in \mathsf{FV}(t)\} \vdash t : o$ is a term and $q \in Q$ is a state. We say that a term s is a* prefix *of a term t just if t has the form $s \, t_1 \cdots t_n$ for some $n \in \mathbb{N}$. A configuration prefix is a pair (c, s) in which c is a configuration of shape (t, q) and s is a prefix of t.*

Abstract typability. The central idea of the algorithm is that the type bindings contained in the context constitute a concise summary of all the information that has been gathered about the scheme and its reducts, as far as acceptance by the property automaton is concerned. We will use the type context to judge whether the assertions represented by configurations are true or not, based on the following simple notion of typability.

Definition 4. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and let (t, q) be an abstract configuration. We say that (t, q) is C-accepted just if $\Gamma_\exists \vdash_{\mathcal{A}} t : q$. We say that (t, q) is C-rejected just if $\Gamma_\forall \vdash_{\mathcal{A}^c} t : q$. We say that (t, q) is C-unknown just if it is neither C-accepted nor C-rejected.*

Until the very last iteration of the algorithm, the configuration (S, q_0), which is the root of the abstract configuration graph, will be C-unknown to all the associated contexts C, but after the last iteration enough type information will have been contributed to the final context C' in order that (S, q_0) will be seen to be either C'-accepting or C'-rejecting.

Abstract configuration graph. The vertices of the abstract configuration graph are either abstract configurations or finite sets of abstract configurations. Viewed as an assertion, a vertex which is a finite set of configurations $\{(s_1, q_1), \ldots, (s_n, q_n)\}$ should be interpreted *conjunctively*, i.e. as requiring that for each $i \in [1..n]$, s_i generates a tree that is accepted from state q_i.

Definition 5. *An abstract configuration graph A is a tuple $\langle V, E, B \rangle$ in which $\langle V, E \rangle$ is a directed graph and B is a set of mappings from variables $y \in \mathcal{Y}$ to terms $t \in T_\Sigma(\mathcal{Y}, \mathcal{N})$. Each vertex $v \in V$ is either (i) an abstract configuration or (ii) a finite set of abstract configurations; and edges $E \subseteq V \times V$ are unlabelled. Given a typing context C, the* abstract configuration graph of C, denoted $\mathsf{ACG}(C)$, *is the abstract configuration graph $\langle V_C, E_C, B_C \rangle$ defined inductively by the system in Figure 4.*

The set B of *bindings* acts as the abstract environment for the purposes of defining the abstraction. We consider motivation of each of the rules of the inductive definition in turn. First, the rule (G1) defines the root of the graph. The premise ensures that, if we already know that S generates a tree that is either accepted from q_0 or rejected from q_0 then we need not do any state space exploration. This kind of premise is common to many of the rules to ensure that work is not done unnecessarily. In fact, one can state an invariant about the abstract typability of the vertices in any abstract configuration graph:

Lemma 1. *Let C be a context. For each configuration $c \in V_C$, c is C-unknown.*

In case (S, q_0) were C-accepting or C-rejecting, the graph would be empty and the sequence of contexts will stabilise.

Rule (G2) simulates the contraction of a redex, but it does so in an abstract way. To apply the rule requires that a configuration $(F \, s_1 \cdots s_n, q)$ containing a redex occurs in the graph. The consequence is that an abstraction of the contraction of that redex is added as a new configuration. However, it is abstract because,

---

(G1) Whenever all the following are true:

    (i) (S, q_0) is C-unknown

  then all the following are also true:

    • $(S, q_0) \in V_C$

(G2) Whenever all the following are true:

    (i) $(F \, s_1 \cdots s_n, q) \in V_C$

    (ii) $\mathcal{R}(F) = \lambda x_1 \cdots x_n. \, t$

    (iii) $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_n \to o$

    (iv) $(t[y_1/x_1, \cdots, y_n/x_n], q)$ is C-unknown

    (v) for each $i \in [1..n]$,
        $y_i = \mathsf{var}(\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i), \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i), \kappa_i)$

  then all the following are also true:

    • $(t[y_1/x_1, \cdots, y_n/x_n], q) \in V_C$

    • $\langle (F \, s_1 \cdots s_n, q), (t[y_1/x_1, \cdots, y_n/x_n], q) \rangle \in E_C$

    • for each $i \in [1..n]$: $y_i \mapsto s_i \in B_C$

(G3) Whenever all the following are true:

    (i) $(a \, s_1 \cdots s_n, q) \in V_C$

    (ii) $S \models \delta(q, a)$

    (iii) for all $(i, q') \in S$, (s_i, q') is not C-rejected

  then all the following are also true:

    • $\{(s_i, q') \mid (i, q') \in S\} \in V_C$

    • $\langle (a \, s_1 \cdots s_n, q), \{(s_i, q') \mid (i, q') \in S\} \rangle \in E_C$.

(G4) Whenever all the following are true:

    (i) $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in V_C$

    (ii) $i \in [1..n]$

    (iii) (s_i, q_i) is not C-accepted

  then all the following are also true:

    • $(s_i, q_i) \in V_C$

    • $\langle \{(s_1, q_1), \ldots, (s_n, q_n)\}, (s_i, q_i) \rangle \in E_C$

(G5) Whenever all the following are true:

    (i) $(y \, s_1 \cdots s_n, q) \in V_C$

    (ii) $y \mapsto t \in B_C$

  then all the following are also true:

    • $(t \, s_1 \cdots s_n, q) \in V_C$

    • $\langle (y \, s_1 \cdots s_n, q), (t \, s_1 \cdots s_n, q) \rangle \in E_C$

Figure 4. Abstract configuration graph construction.

---

rather than substituting actual parameters for formals, typed variables are substituted for the formals. These typed variables must be appropriate for the actuals that they abstract, hence there is the constraint that, if y_i abstracts actual parameter s_i, then it had better be that $y_i = \mathsf{var}(\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i), \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i), \mathsf{K}(s_i))$. This ensures that type information is invariant across the abstraction, in the following sense:

Proposition 1. *Let* $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ *be a type context. For all* $y \mapsto t \in B_C$, $\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(y) = \bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(t)$ *and* $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(y) = \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(t)$.

To properly define the abstraction in terms of the new variable y_i, a binding is added to B_C with the effect that $y_i \mapsto s_i$. Consequently, we may think that s_i is in the set of terms abstracted by y_i.

Rule (G3) simulates a transition of the automaton on reading a terminal symbol: if there is a terminal symbol-headed configuration $(a \ s_1 \ \cdots \ s_n, q)$ in the graph, then its children comprise all of the possible satisfying assignments to $\delta(q, a)$ expressed as sets of configurations. Recalling that each vertex that is a set of configurations should be thought of conjunctively, the children of $(a \ s_1 \ \cdots \ s_n, q)$, taken as a whole, should be thought of disjunctively – $a \ s_1 \ \cdots \ s_n$ generates a tree accepted from state q just if all the configurations contained in some child (satisfying assignment) are shown to be accepted. Rule (G4) simply decomposes set vertices into their constituent configurations. Therefore, the children of a set vertex should be thought of conjunctively.

Finally, rule (G5) ties the knot on the abstraction by considering the case when a typed variable is in head position in a configuration. In this case, the binding set is consulted and a node is added for each binding to the appropriate variable. We will think of the children of such a vertex conjunctively: for $y \ s_1 \ \cdots \ s_n$ to generate a tree accepted from state q, it had better be that every term that it abstracts generates a tree accepted from state q.

Due to the abstraction at the point of contraction in (G2) and the limited substitution (only in head position) in (G5), $\mathsf{ACG}(C)$ is necessarily a finite construction. In fact, we can go further:

Lemma 2. *Let C be a type context. Then the size of V_C is bounded by a polynomial function of the size of the scheme.*

Classification of leaves. Let us consider for a moment the leaves of $\mathsf{ACG}(C)$ for some type context C. It follows from the definition that the leaves all have a particular form. Every leaf is a configuration headed by a non-terminal symbol, i.e. a redex. Moreover, each such redex, if contracted using rule (G2), would yield a new configuration which is already known to be either C-accepting or C-rejecting. It is for this reason that such configurations are leaves: (G2) does not apply because the fourth premise would be violated.

Definition 6. *Given a type context $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$, the leaves (i.e. those vertices that have no children) of $\mathsf{ACG}(C)$ can be classified into two sets:*

(ACCEPTING LEAVES) *These leaves are configurations of the form* $(F \ s_1 \ \cdots \ s_n, q)$ *where* $\mathcal{R}(F) = \lambda x_1 \ \dots \ x_n. t$, *for each* $i \in [1..n]$, *there is a typed variable y_i such that* $\mathsf{A}(y_i) = \bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i)$ *and* $\Gamma_\exists \vdash_{\mathcal{A}} t[y_1/x_1, \dots, y_n/x_n] : q$.

(REJECTING LEAVES) *These leaves are configurations of the form* $(F \ s_1 \ \cdots \ s_n, q)$ *where* $\mathcal{R}(F) = \lambda x_1 \ \dots \ x_n. t$, *for each* $i \in [1..n]$, *there is a typed variable y_i such that* $\mathsf{R}(y_i) = \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i)$ *and* $\Gamma_\forall \vdash_{\mathcal{A}^c} t[y_1/x_1, \dots, y_n/x_n] : q$.

Note that a rejecting leaf is not itself C-rejecting, by Lemma 1 since it is in the graph it is necessarily C-unknown, but its contractum is C-rejecting. Similarly accepting leaves are not themselves C-accepting, but the contractum of an accepting leaf is C-accepting.

Lemma 3. *Let C be a context. Every leaf in $\mathsf{ACG}(C)$ is accepting or rejecting.*

4.2 The rejecting region

Region of rejection. The construction of an ACG from a given type context C is a method for analysing the type context. By constructing the graph it is possible to see where the information in the type context is deficient, and the main tools for identifying and correcting deficiencies are the regions and region type

extraction respectively. Consider a rejecting leaf v of the form $(F \ s_1 \cdots s_n, q)$. By definition, the contraction of this configuration using (G2) would yield a configuration $(t[y_1/x_1, \dots, y_n/x_n], q)$ which is already C-rejecting. In other words, the tree generated by *any* term of the form $t[t_1/y_1, \dots, t_n/y_n]$ such that $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(t_i) \leq \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(y_i)$ for each i is sure to be rejected from state q. Assuming that $\Gamma_\forall$ is co-consistent, this follows because necessarily $\Gamma_\forall \vdash_{\mathcal{A}^c} t[t_1/y_1, \dots, t_n/y_n] : q$. Recalling Proposition 1, one such sequence of t_i are the actual parameters of the term component of the rejecting leaf we started with: v. Hence, because we know that the contractum of the term part of v generates a tree that is rejected from state q_0, necessarily the term part of v itself generates a tree that is rejected from state q_0. So we have identified that v should be classified as C-rejecting (but is not currently). Through analogous reasoning (and remembering the conjunctive and disjunctive interpretations of the child relation in the graph), it is possible to identify other such vertices which are necessarily rejecting. The collection of all such is called the rejecting region.

Definition 7. *Given a context C, we define a subset $\mathsf{RR}(C) \subseteq V_C$ of the vertices of $\mathsf{ACG}(C)$, called the rejecting region, inductively:*

(R1) If c is a rejecting leaf then $c \in \mathsf{RR}(C)$.
(R2) If $\{(s_1, q_1), \dots, (s_n, q_n)\} \in V_C$ and there exists $j \in [1..n]$ and $(s_j, q_j) \in \mathsf{RR}(C)$ then $\{(s_1, q_1), \dots, (s_n, q_n)\} \in \mathsf{RR}(C)$.
(R3) If $\langle (F \ s_1 \ \cdots \ s_n, q), (t, q) \rangle \in E_C$ and $(t, q) \in \mathsf{RR}(C)$ then $(F \ s_1 \ \cdots \ s_n, q) \in \mathsf{RR}(C)$.
(R4) If $(a s_1 \cdots s_n, q) \in V_C$ and, for every v, $\langle (a s_1 \cdots s_n, q), v \rangle \in E_C$ implies $v \in \mathsf{RR}(C)$, then $(a \ s_1 \ \cdots \ s_n, q) \in \mathsf{RR}(C)$.
(R5) If $(y \ s_1 \ \cdots \ s_n, q) \in V_C$ and, for all $y \mapsto t \in B_C$, $(t \ s_1 \ \cdots \ s_n, q) \in \mathsf{RR}(C)$ then $(y \ s_1 \ \cdots \ s_n, q) \in \mathsf{RR}(C)$.

Unless it is the final iteration of the algorithm, the rejecting region will always be non-empty. The fact that an absence of rejecting leaves is an absence of counterexamples in the abstraction is formalised later, in Lemma 6.

Rejection type extraction. The vertices in the rejecting region are those configurations, that we have identified by constructing $\mathsf{ACG}(C)$, which *should* be classified by the context as rejecting, but *are not* – each is necessarily C-unknown, since it belongs to the graph. So the rejecting region represents a weakness in the context. To remedy it, from the region we will extract new type information to be added to the context ready for the next iteration.

Definition 8. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a typing context and $v \in \mathsf{RR}(C)$. A witness to the membership of v in $\mathsf{RR}(C)$ is a proof tree T rooted at the statement $v \in \mathsf{RR}(C)$ and constructed according to the rules (R1) – (R5). We describe an assignment of type environments $\mathcal{M}(T)$ to proof trees T, inductively on the shape of the proof.*

(M1) If the proof is by (R1) then v is a configuration of the form $(F \ s_1 \ \cdots \ s_n, q)$, and we set $\mathcal{M}(T)$ to be the single binding:

$$F : \bigwedge \mathbb{T}(\Gamma_\forall)(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall)(s_n) \to q$$

(M2) If the proof is by (R2) then v is a set $\{(s_1, q_1), \dots, (s_n, q_n)\}$ and for some $j \in [1..n]$ there is an immediate sub-proof T' of (s_j, q_j). We set $\mathcal{M}(T) = \mathcal{M}(T')$.

(M3) If the proof is by (R3) then v is a configuration of the form $(F \ s_1 \ \cdots \ s_n, q)$ and, necessarily, there is an immediate sub-proof T' of (t, q). We take for $\mathcal{M}(T)$ the environment $\mathcal{M}(T')$ augmented by the binding:

$$F : \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_n) \to q$$

(M4) If the proof is by (R4) then v is of the form $(a \ s_1 \ \cdots \ s_n, q)$ with a set W of children. For each $w \in W$, there is a subproof T_w. We set $\mathcal{M}(T) = \biguplus \{\mathcal{M}(T_w) \mid w \in W\}$.

(M5) If the proof is by (R5) then v is a configuration of the form $(y \ s_1 \ \cdots \ s_n, q)$ and, necessarily, for each $y \mapsto t \in B_C$ there is an immediate sub-proof T_t of $(t \ s_1 \ \cdots \ s_n, q)$. Let us write $\mathcal{M}(T_y)$ simply as notation for the environment given by $\biguplus \{\mathcal{M}(T_t) \mid y \mapsto t \in B_C\}$. We take for $\mathcal{M}(T)$ the environment:

$$y \colon \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T_y))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T_y))(s_n) \to q$$

Finally, we define a type environment, $\mathsf{env}_\mathsf{R}(C)$, whose domain is a subset of $\mathsf{dom}(\mathcal{N})$ and which is extracted from $\mathsf{RR}(C)$ by:

$$\mathsf{env}_\mathsf{R}(C)(F) = \bigwedge \{\mathcal{M}(T)(F) \mid \exists c \in \mathsf{RR}(C) \text{ with witness } T\}$$

So the types are extracted in an inductive fashion, starting from the leaves of each witness with the environment $\Gamma_\forall$ and working backwards, adding new types along with way. It is this well-foundedness that ensures that the types that are extracted are all "correct":

Lemma 4. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. If $\Gamma_\forall$ is $\mathcal{G}$-co-consistent in $\mathcal{A}^c$ then $\Gamma_\forall \uplus \mathsf{env}_\mathsf{R}(C)$ is $\mathcal{G}$-co-consistent in $\mathcal{A}^c$.*

Furthermore, whenever the rejecting region is non-empty, then genuinely new type information will be extracted. Taken together with Lemma 6, the following result is the key measure of progress in the algorithm.

Lemma 5. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and $\mathsf{ACG}(C)$ have some rejecting leaf. Then $\mathsf{env}_\mathsf{R}(C) \setminus \Gamma_\forall \neq \emptyset$.*

4.3 The accepting region

Region of acceptance. In a similar way, the accepting region serves to identify those configurations that *should* be classified as accepting by the type context, but which are not. The rules by which vertices can be inferred to be accepting are all complimentary to those that define the rejecting region (except for the case of variable-headed nodes, which are conjunctive in both regions) and, indeed, the construction is coinductive.

Definition 9. *Given a typing context C, we define a subset $\mathsf{RA}(C) \subseteq V_C$ of the vertices of $\mathsf{ACG}(C)$, called the accepting region, coinductively by:*

(A1) If $(F \ s_1 \ \cdots \ s_n, q) \in \mathsf{RA}(C)$ with successor $(t, q) \in V_C$, then $(t, q) \in \mathsf{RA}(C)$.

(A2) If $(a \ s_1 \ \cdots \ s_n, q) \in \mathsf{RA}(C)$, then there is some S such that $S \models \delta(q, a)$ and $\{(s_i, q') \mid (i, q') \in S\} \in \mathsf{RA}(C)$.

(A3) If $\{(s_1, q_1), \ldots, (s_m, q_m)\} \in \mathsf{RA}(C)$ then, for all $i \in [1..m]$, $(s_m, q_m) \in \mathsf{RA}(C)$.

(A4) If $(y \ s_1 \ \cdots \ s_n, q) \in \mathsf{RA}(C)$ then, for all $y \mapsto t \in B_C$, $(t \ s_1 \ \cdots \ s_n, q) \in \mathsf{RA}(C)$.

(A5) If $c \in \mathsf{RA}(C)$ is a leaf in V_C then c is an accepting leaf.

However, unlike the case for rejecting region there is no similar guarantee of non-emptiness on non-final iterations. It is perfectly possible that on any given iteration, the accepting region may be empty. In contrast, the absence of rejecting leaves, and hence emptiness of the rejecting region, leads to termination.

Lemma 6. *Let C be a type context and $\mathsf{ACG}(C)$ have no rejecting leaves. Then $\mathsf{ACG}(C) = \mathsf{RA}(C)$.*

Thus, in particular, $\mathsf{RA}(C)$ will contain the root and so $S : q_0$ will be added to the accepting environment, signalling termination.

Acceptance type extraction. To extract new type information from the accepting region we follow the approach of Kobayashi

and Ong [12, 14], in which types are assigned to prefixes of configurations recursively based on the kind of the prefix.

Definition 10. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. To each prefix (c, s) of each configuration $c \in \mathsf{RA}(C)$, we assign a strict type $\mathsf{extr}(c, s)$, which is defined inductively over the structure of the kind of s.*

(i) If s is of base kind, necessarily c is of the form (s, q) and set $\mathsf{extr}(c, s) = q$.

(ii) If s is of arrow kind, necessarily c is of the form $(s\,t_1 \cdots t_n, q)$. Let W be the set of accepting region configurations with prefix t_1. Set:

$$\mathsf{extr}(c, s) = \bigwedge \mathbb{T}_\mathcal{A}(\Gamma_\exists)(t_1) \wedge \bigwedge_{c' \in W} \mathsf{extr}(c', t_1) \to \mathsf{extr}(c, s\,t_1)$$

We define a type environment, $\mathsf{env}_\mathsf{A}(C)$, whose domain is a subset of $\mathsf{dom}(\mathcal{N})$ and which is extracted from $\mathsf{RA}(C)$ by:

$$\mathsf{env}_\mathsf{A}(C)(F) = \bigwedge \{\tau \mid \exists c \in \mathsf{RA}(C) \cdot \mathsf{extr}(c, F) = \tau\}$$

The acceptance types extracted in this way are all "correct":

Lemma 7. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context. If $\Gamma_\exists$ is $(\mathcal{G},\mathcal{A})$-consistent then also $\Gamma_\exists \uplus \mathsf{env}_\mathsf{A}(C)$ is $(\mathcal{G},\mathcal{A})$-consistent.*

4.4 Fixed point construction

Abstraction refinement. Finally, we are in a position to describe the overall abstraction refinement loop. Starting from a context C_0 that contains only the type assumptions on typed variables used by the abstraction, on each iteration the algorithm analyses the given context, say C_i, by constructing $\mathsf{ACG}(C_i)$; it then identifies deficiencies in C_i by constructing regions and attempts to repair those deficiencies by extracting new environments. Eventually, the type $S : q_0$ will be extracted from one of the regions and the algorithm will terminate.

Definition 11. *Recall A and R in Definition 1. The algorithm consists of constructing an eventually stable sequence of type contexts $(C_i)_{i \in \mathbb{N}}$ as follows:*

$$\begin{aligned} C_0 &= \langle \Gamma_\exists^0, \Gamma_\forall^0 \rangle &&= \langle \mathsf{A}, \mathsf{R} \rangle \\ C_{k+1} &= \langle \Gamma_\exists^{k+1}, \Gamma_\forall^{k+1} \rangle &&= \langle \Gamma_\exists^k \uplus \mathsf{env}_\mathsf{A}(C_k), \Gamma_\forall^k \uplus \mathsf{env}_\mathsf{R}(C_k) \rangle \end{aligned}$$

with limit, say $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$. Then if $q_0 \in \Gamma_\exists(S)$ answer YES and otherwise answer NO.

Since the initial environments $\Gamma_\exists^0$ and $\Gamma_\forall^0$ are trivially $\mathcal{G}$-consistent in $\mathcal{A}$ and $\mathcal{G}$-co-consistent in $\mathcal{A}^c$ respectively and since every extension of these environments by env_A and env_R preserves this property, it follows that the limit of the sequence also enjoys the property and hence can be relied upon to decide the model checking problem. Furthermore, since progress is guaranteed by Lemma 6 and Lemma 5, and the size of rejecting environment $\Gamma_\forall$ is bounded by the number of well-kinded types, we can state the following correctness theorem:

Theorem 2. *For any $\mathcal{G}$, $\mathcal{A}$, the algorithm terminates and:*

- *Answer YES implies $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$.*
- *Answer NO implies $\mathsf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^\perp)$.*

Furthermore, since each ACG is, in the worst case, polynomial in the size of the scheme (but in general, hyper-exponential in the order of the scheme) and the amount of work involved in computing the ACG, the regions and type extraction is polynomial in the size of the scheme, it follows that each iteration of the algorithm takes, in the worst case, an amount of time polynomial in the size of the scheme. Since the number of iterations is bounded by the number of well-kinded types, which is also polynomial in the size of the

Benchmark	Rules	Ord	Dec	PREFACE	TRECS
map_filter-e	64	5	R	0.53	0.01
fold_left	65	4	A	0.39	0.03
fold_right	65	4	A	0.39	0.03
forall_eq_pair	66	4	A	0.39	0.03
forall_leq	66	4	A	0.39	0.03
a-cppr	74	3	R	0.38	0.01
search-e	96	5	R	0.90	0.01
search	119	4	A	0.46	1.04
map_filter	143	5	A	0.51	0.13
risers	148	5	A	0.44	0.33
r-file	156	2	A	0.82	1.50
fold_fun_list	197	6	A	0.44	0.89
zip	210	3	A	0.58	15.10

Table 1. Benchmarks of category 1.

scheme, it follows that the algorithm as a whole is polynomial in the size of the scheme, assuming its order and arity and the size of the automaton are taken to be fixed.

5. Implementation and evaluation

We have implemented the algorithm in a prototype tool, called PREFACE, which is written in F# and available to download from http://mjolnir.cs.ox.ac.uk/web/preface.

5.1 Implementation

To ensure efficiency we have taken a number of decisions about how to code specific aspects of the algorithm which deviate from the presentation. Rather than constructing them as part of initialisation, we build the maps A and R lazily, adding bindings as they are needed by applications of rule (G2). Further, the implementation uses a flow analysis with increased accuracy, distinguishing between instances of arguments using not just the triple of acceptance type, rejection type and kind, but additionally the formal parameter and the state component of the calling configuration. Since intersection type checking is frequently invoked as part of the decision procedure, we aim to ensure it is done as efficiently as possible. Hence, we omit subtype checking[3] and hash cons the intersection types. Finally, rather than compute all possible witnessing trees for any given vertex $v \in RR(C)$, we take one representative, which is a function of the construction of the region.

5.2 Evaluation

We have evaluated the tool on the large collection of recursion scheme model checking instances found online and in the related literature. The full listing of results is available in the appendix of the long version of this paper, here we aim to present a very small representative sample in order to describe the general trends.

We have picked a number of benchmarks from three categories, which are displayed in Tables 1, 2 and 3 respectively. The benchmarks were run on an Intel Xeon machine with 12GB of RAM and 4 cores running at 2.4GHz, limiting the run-time of each tool on each benchmark to 2 minutes. In all cases the columns are, respectively, the name of the benchmark, the number of rules (equations), the order of the scheme, whether the tree is accepted (A) or rejected (R) by the property automaton. The remaining columns list the time taken by each tool from start to finish, which is either given in seconds, or marked "–" in case the tool ran out of resources.

Category 1. The first category consists of model checking instances that have arisen from OCaml verification problems via the

---

[3] Note that this does not affect the soundness or completeness of the decision procedure.

predicate abstraction tool MoCHi [26]. Although MoCHi can solve many complex examples, the scalability of a full blown predicate abstraction tool for higher-order programs is still an open topic of research. Consequently the problem instances derived from this tool are exclusively quite small and mostly less than 100 rules. MoCHi generates a mild extension of recursion schemes called RSFD [16], which are currently only supported by PREFACE and TRECS. Our tool PREFACE can typically solve each of these instances in less than 0.5 seconds, but this is already roughly an order of magnitude slower than TRECS. However, the overhead incurred by JIT compilation on Mono is a major factor; when compiled ahead of time on Windows, the time taken by PREFACE to solve these instances is typically less than 0.05 seconds, although usually still slower than TRECS. As the benchmarks become slightly larger, towards the bottom of the table, the time taken by TRECS starts to lag behind the time taken by PREFACE, which is the start of a general trend in the data to follow.

Category 2. The second category consists of instances arising from a tool for performing exact flow analysis [27]. These examples are significantly larger than those of Category 1 and, indeed, form some of the largest instances on which HORS model checkers have been evaluated in the literature as of the time of writing. Although they have fewer than 1000 rules each, due to the nature of the verification algorithm that produces them, they have high order and very high maximum arity, with `cfa-life2` being order 14 with arity 29 functions. Consequently, many of the tools have difficulty, but among those that are able to solve these instances, the trend observed in the Category 1 examples can be seen to continue.

Category 3. The final category consists of instances of a family of schemes due to Kobayashi [13]. This family of instances was designed to be deliberately difficult for bounded model checking style algorithms such as the hybrid algorithm of TRECS, whilst simultaneously being a good indicator of the scalability of Kobayashi's linear time algorithm as implemented in GTRECS. Although these schemes are not "real" in the sense of arising from program analysis problems they seem a good measure of scalability since they generate hyper-exponentially sized trees (and hence use the full power of higher-order schemes), their certificates are proportional to the number of rules and they can push the model checkers much further since the family contains much larger schemes than can be produced by current verification tools. The first half of the table shows instances which are generated with order-2 schemes and the second half shows instances with order-4 schemes. The size of the schemes roughly doubles within each half by row. As expected, GTRECS does a good job at solving even relatively large examples at order-2, though it has some difficulty at higher-orders. PREFACE does even better and, in contrast to the other tools, can be seen to solve these examples in time which is roughly linear in the number of rules.

5.3 Analysis

The good performance of our algorithm at scale must be attributed to the abstraction refinement approach that we have adopted. Since recursion schemes cannot destruct the trees that they create, their interesting behaviours are exclusively due to control flow arising from complex uses of higher-order functions. Hence, a CFA-style abstraction in combination with a property directed refinement works well, since this abstraction is particularly well suited to emphasising some of the essential structure of higher-order control flow and the refinement ensures that particular limitations of the CFA with respect to specific problem instances can be compensated for. A good example of this is in the Category 3 examples, whose very regular structure is determined by the analysis quickly and

Benchmark	Rules	Ord	Dec	PREFACE	HORSAT	HORSATT	C-SHORE	GTRECS2	TRAVMC	TRECS
cfa-psdes	237	7	A	0.51	0.28	1.81	3.44	–	–	–
cfa-matrix-1	383	8	A	0.61	0.73	6.30	18.58	–	–	–
cfa-life2	898	14	A	1.46	5.94	–	–	–	–	–

Table 2. Benchmarks of category 2.

Benchmark	Rules	Ord	Dec	PREFACE	HORSAT	HORSATT	C-SHORE	GTRECS2	TRAVMC	TRECS
exp2-1600	1606	2	A	8.39	–	–	–	10.47	–	–
exp2-3200	3206	2	A	17.51	–	–	–	59.13	–	–
exp2-6400	6406	2	A	39.58	–	–	–	–	–	–
exp2-12800	12806	2	A	92.19	–	–	–	–	–	–
exp4-400	408	4	A	14.12	–	106.53	–	–	–	–
exp4-800	808	4	A	30.55	–	–	–	–	–	–
exp4-1600	1608	4	A	71.06	–	–	–	–	–	–
exp4-3200	3208	4	A	–	–	–	–	–	–	–

Table 3. Benchmarks of category 3.

hence all are solved in exactly 3 iterations (independent of the number of rules or the order of the scheme).

However, although the refinement will eventually compensate for these particular limitations of the CFA, it is possible to construct instances in which the number of iterations required is unacceptably large with respect to the characteristics of the instance. Although such examples do not seem to occur in the corpus of instances drawn from the higher-order model checking literature and associated verification tools, we have been able to construct very small and simple schemes which exhibit this bad behaviour. The examples in Table 4 are based on a family of Boolean programs defined in [1]. Each of these instances is first order and consists of a few hundred rules, but the property automaton is strictly alternating. We record the number of iterations (Rnds) and the time (in seconds) taken by PREFACE and an extension, PREFACE$^+$, in the remaining columns.

		PREFACE		PREFACE$^+$	
Benchmark	Rules	Rnds	Time	Rnds	Time
t100	104	202	45.38	8	2.17
t200	204	402	178.81	8	3.99
t400	404	802	732.40	8	7.97
t800	804	1602	3074.03	9	18.50
t1600	1604	3202	13561.26	9	41.02

Table 4. Bad behaviour.

Each example tn consists of roughly n functions, which make exponentially many calls to each other in sequence, so that function $F1$ calls function $F2$ twice, function $F2$ calls function $F3$ twice and so on. However, what makes the examples expose bad behaviour in PREFACE is not the number of calls (which is hyper-exponentially smaller than the number of calls made by the exp$n$-m examples in Category 3), but the fact that each call is made once with a term that will eventually evaluate to true and once with a term that will eventually evaluate to false and that refutation of the property depends upon distinguishing between the two. On iteration $2i+2$, the flow analysis is only able to distinguish between the true and false variants of the calls made to functions Fj for $j \geq n - i$. Every 2 iterations, enough new information has been discovered in order to distinguish one more level of function calls, and hence each tn is solved after roughly $2n$ iterations.

This analysis suggests that not enough type information is being recovered from the ACG at each iteration and, indeed, by extending our implementation with heuristics for extracting more types, we have solved these examples more quickly. Our extension, labelled

PREFACE$^+$ in the table, exploits the *incremental* nature of the algorithm. By this we mean the following characteristics:

(i) The algorithm makes progress on each iteration, in the sense of extracting new types (even if the number of types extracted is perhaps smaller than one would like).

(ii) On each iteration, the provenance of the context is not important, only the fact that it comprises environments that are consistent and co-consistent respectively.

(iii) The larger the context for a given iteration, the more accurate the analysis of that iteration.

Our extension consists of, in a separate thread running in parallel with the main algorithm, taking the ACG that has most recently been computed and, based on the relationships between the vertices, making informed "guesses" at possible new types. In general the guesses may be incorrect, in the sense of leading to the creation of an environment which is unjustifiable, so the new environments are first type-checked according to the rules of (co-)consistency. If they type-check, the new types are then added into the context at the earliest opportunity and the guessing process can be repeated.

This extension appears to work well to solve the examples in Table 4, cutting the time taken to process t1600 down from almost 3 hours to under one minute! However, it seems unlikely to scale well to higher-orders, where the possible number of types from which to guess is much larger. Consequently, we do not consider this a satisfactory solution and leave to future work a proper treatment of this problem and that of the closely related area of how best to extract counter-example traces.

Note however that this particular class of examples is not difficult for all higher-order model checkers. Since all the examples are first order, the space of types is relatively small and we expect that algorithms that perform a search of this space more directly ought to perform very well. This is certainly the case for the tool HORSAT, which gives the following results reported by Kobayashi and Terao[4]:

Benchmark	t100	t300	t500	t700
HORSAT	0.26	1.00	2.05	3.77

6. Related work

Higher-order model checking algorithms. Exemplified by the tool TRECS [11], the first *practical* algorithms model check HORS

---

[4] Personal communication.

with respect to trivial automata, using intersection types as a finite representation of an infinite transition system. They start from the assignment of the automaton initial state to the start non-terminal S, and compute the "post-image" in the form of type information required to deduce unreachability of the error configurations. In TRECS, the types of non-terminals are first extracted from a finitely reachable part of the (infinite) configuration graph; the resultant type environment is then expanded and used as an over-approximation of a greatest fixpoint (of $shrink$ [12]). PREFACE is similar to TRECS in that it implements a forward algorithm based on intersection types. However, in contrast, PREFACE extracts types from a finite, 0CFA-like abstraction of the configuration graph. Another major difference is that each iteration of PREFACE refines two type environments, one is potentially a certificate of automaton acceptance, and the other of automaton rejection.

Very fast for HORS of up to a few hundred rules, the runtime of TRECS is nonetheless hyper-exponential in the size of the input HORS. The first fixed-parameter polytime (in the size of HORS) algorithm is GTRECS [13] which consists simply of two fixpoint constructions. The key innovation is a game-semantic reading of the intersection types *qua* a pair of expansion relations, modelling the legals moves of the two players of an arena game. The tool TRAVMC [20] is also based on game semantics. The algorithm harvests *variable profiles*, but represented as intersection types, from the *traversals* [21] over the HORS being analysed.

Recently Broadbent et al. [4] have introduced an algorithm based on *collapsible pushdown automata* (CPDA) [8]—which are equi-expressive with HORS—and implemented in the tool C-SHORe [5]. Given an input co-trivial tree automaton, the algorithm uses a generalisation of the saturation algorithm for pushdown automata to compute the "pre-image" of the final error configurations, and checks if it includes the start state. Thus information is propagated in the *backward* direction. Closely related are algorithms HORSAT and HORSATT [3]. Though based on saturation (of HORS rather than CPDA), they may be viewed as fixpoint computation of a function over type environments. To accelerate the pre-image computation, these algorithms benefit from a forward flow analysis, which excludes some irrelevant type bindings.

Abstraction refinement. Counterexample-guided abstraction refinement (CEGAR) was introduced by Clarke et al. [6] for symbolic model checking. The CEGAR loop was first applied to higher-order model checking by Ong and Ramsay [23] and by Kobayashi et al. [17]. The former addresses the undecidable problem of verifying safety properties of pattern-matching recursion schemes, using patterns to abstract properties. The latter is used in conjunction with predicate abstraction to verify simply-typed functional programs generated from infinite data domains such as integers. In contrast, PREFACE builds successively more accurate finite abstractions of the configuration graph of the HORS being analysed, from which potential certificates of acceptance and of rejection are derived.

Type-based flow analysis. Flow analyses were first applied to untyped languages. Jagannathan et al. [9] introduced a type-directed, polyvariant flow analysis for the predicative subset of System F, which can leverage types to analyse programs more precisely. Our algorithm uses an intersection type system for describing automaton definable properties and necessarily works in a situation in which not all type information is known; in contrast theirs uses more standard typing (System F) and starts from a situation in which all types are known. Thus their analysis is comparable to a single iteration of our algorithm where the associated context already contains all the possible correct type information. The property of *respecting types*, which is put forward by the authors as a measure of the appropriateness of a CFA for a typed language is,

for us, actually an essential technical requirement in order to extract new, valid type information.

Plevyak and Chien [24] considered a constraint-based type inference for object-oriented programs. Like our algorithm, not all type information is known at the start, and they compute types iteratively based on a flow analysis. However, within a single iteration they do not distinguish based on type information, instead they distinguish based on clashes discovered in the previous iteration. A problem with their approach of distinguishing calls unconditionally is that there may be infinitely many counterexamples which are being distinguished one at a time. Because of possible non-termination, their method cannot handle recursion in general.

Acknowledgments

References

[1] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Symposium on Model Checking of Software, SPIN'00*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[2] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[3] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *Computer Science Logic, CSL'13*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013.

[4] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *International Colloquium on Automata, Languages and Programming, ICALP'12*, volume 7392 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2012.

[5] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-SHORe: a collapsible approach to verifying higher-order programs. In *International Conference on Functional Programming, ICFP'13*, pages 13–24. ACM, 2013.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, CAV'00*, pages 154–169. Springer-Verlag, 2000.

[7] M. Coppo and M. Dezani. An extension of the basic functionality theory for the lambda-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[8] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Logic in Computer Science, LICS'08*, pages 452–461. IEEE Computer Society, 2008.

[9] S. Jagannathan, S. Weeks, and A. K. Wright. Type-directed flow analysis for typed intermediate languages. In *Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1997.

[10] N. D. Jones. Flow analysis of lambda expressions. In *International Colloquium on Automata, Logic and Programming, ICALP'81*, volume 115 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 1981.

[11] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Symposium on Principles of Programming Languages, POPL'09*, pages 416–428. ACM, 2009.

[12] N. Kobayashi. Model-checking higher-order functions. In *Principles and Practice of Declarative Programming, PPDP'09*, pages 25–36. ACM, 2009.

[13] N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *Foundations of Software Science and Computational Structures, FOSSACS 2011*, volume 6604 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2011.

[14] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Logic in Computer Science, LICS 2009*, pages 179–188. IEEE Computer Society, 2009.

[15] N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011.

[16] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Symposium on Principles of Programming Languages, POPL'10*, pages 495–508, 2010.

[17] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CE-GAR for higher-order model checking. In *Programming Languages Design and Implementation, PLDI'11*, pages 222–233. ACM, 2011.

[18] M. Might and O. Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(5–6):821–864, 2008.

[19] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computuer Science*, 54(2–3):267–276, 1987.

[20] R. P. Neatherway, C.-H. L. Ong, and S. J. Ramsay. A traversal-based algorithm for higher-order model checking. In *International Conference on Functional Programming, ICFP'12*, pages 353–364. ACM, 2012.

[21] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Logic In Computer Science, LICS'06*, pages 81–90. IEEE Computer Society, 2006.

[22] C.-H. L. Ong. Models of higher-order computation: Recursion schemes and collapsible pushdown automata. In *Logics and Languages for Reliability and Security, LLRS'10*, pages 263–299. 2010.

[23] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Symposium on Principles of Programming Languages, POPL'11*, pages 587–598. ACM, 2011.

[24] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Object Oriented Programming Systems, Language and Applications, OOPSLA'94*, pages 324–340. ACM, 1994.

[25] S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. Long version: http://mjolnir.cs.ox.ac.uk/papers/preface.pdf.

[26] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *Partial Evaluation and Semantic-Based Program Manipulation, PEPM'13*, pages 53–62. ACM, 2013.

[27] Y. Tobita, T. Tsukada, and N. Kobayashi. Exact flow analysis by higher-order model checking. In *Functional and Logic Programming, FLOPS'12*, volume 7294 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2012.

[28] S. van Bakel. Strict intersection types for the lambda calculus. *ACM Computing Surveys*, 43(3):20, 2011.

A. A guided run of the algorithm

By way of an example, we consider an instance of the HORS model checking problem due to Kobayashi [12] (though it is itself an encoding of a problem considered by Might and Shivers [18]).
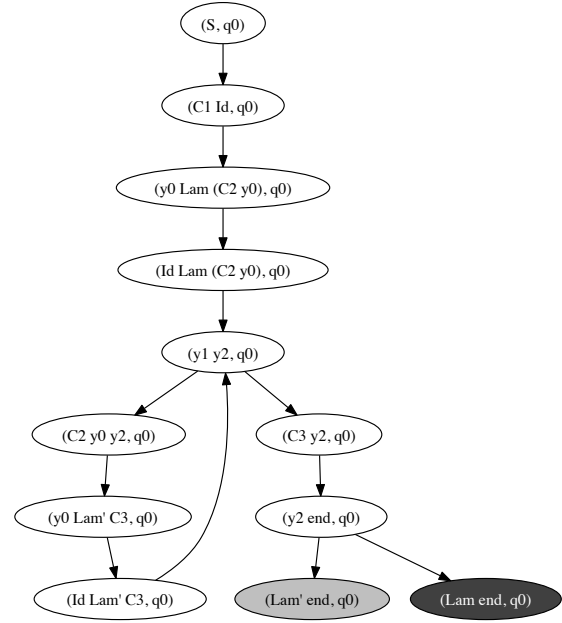
Figure 5. Abstract configuration graph for iteration 1.

A.1 Model checking instance

The instance consists of the following recursion scheme $\mathcal{G}$:

$$
\begin{aligned}
S &= \textit{C1 Id} \\
C1 &= \lambda id.\, id\ Lam\ (\textit{C2 id}) \\
C2 &= \lambda id\ unused.\, id\ Lam'\ C3 \\
C3 &= \lambda x.\, x\ \mathsf{end} \\
Lam &= \lambda x.\, \mathsf{flow}\ x \\
Lam' &= \lambda x.\, x \\
Id &= \lambda x\ k.\, k\ x
\end{aligned}
$$

over the terminal symbols $\mathsf{flow} : o \to o$ and $\mathsf{end} : o$. The property is specified by the following deterministic trivial automaton $\mathcal{A}$ which consists of a single state q_0 and a single transition $\delta(q_0, \mathsf{end}) = \mathsf{t}$. The idea is that the scheme will be rejected just if a use of Lam flows to the result, i.e. if flow appears in $\mathsf{Tree}(\mathcal{G})$. However, this is not the case, and hence $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A})$ is determined after three iterations.

The typed variables that will be used throughout this example are shown in Table 5. The reader may wish to periodically consult this listing whilst following the constructions described below.

A.2 Iteration 1

In the first iteration, the initial context C_0 just consists of $\langle \mathsf{A}, \mathsf{R} \rangle$. The graph $\mathsf{ACG}(C_0)$ is shown in Figure 5. Note that the accepting and rejecting regions are coloured light and dark grey respectively. The binding set for the graph is as shown below. Note that, as a small optimisation to aid readability, we will never add bindings of the form $y \mapsto y$ to the set during graph construction.

$$
\begin{array}{lll}
y2 \mapsto Lam' & y2 \mapsto Lam & y0 \mapsto Id \\
y1 \mapsto C3 & y1 \mapsto C2\ y_0 &
\end{array}
$$

Since there is no intersection type information for any of the non-terminals during the first iteration, there is no way to distinguish the two actual parameters Lam and Lam' which occur in two different calls to the non-terminal Id, once due to the contraction of

Symbol	Acceptance Type	Rejection Type	Kind
$y0$	$\top$	$\top$	$(o \to o) \to ((o \to o) \to o) \to o$
$y1$	$\top$	$\top$	$(o \to o) \to o$
$y2$	$\top$	$\top$	$o \to o$
$y3$	$q_0 \to q_0$	$\top$	$o \to o$
$y4$	$\top$	$\top \to q_0$	$o \to o$
$y5$	$(q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0$	$\top$	$(o \to o) \to ((o \to o) \to o) \to o$

Table 5. Typed variables listing.

$Id\ Lam\ (C2\ y0)$ and again due to the contraction of $Id\ Lam'\ C3$. Consequently, the typed variable $y2$, which is used to abstract arguments of type $(\top, \top, o \to o)$, confuses the two calls which leads to the undesirable (and spurious) effect of having the redex $Lam\ end$ occur in a configuration in the graph. This configuration is a rejecting leaf, because the configuration that would result from contracting the redex, which would be of the form $(\mathsf{flow}\ y, q_0)$, is C-rejecting for any C, in essence because the tree $\mathsf{flow}\ s$ is rejected by $\mathcal{A}$ for any tree s.

The rejecting region is grown out of the set of rejecting leaves, but in this case there is only one. Furthermore, because the parent of the rejecting leaf is a variable-headed configuration whose children are not all themselves in the rejecting region, the region is confined to just the rejecting leaf with which it started. From this region we extract a single type, which is $Lam : \top \to q_0$, representing the fact that when applied to an argument of no discernible rejecting type (namely end), Lam will construct a tree which is rejected from state q_0.

Similarly, although it is not "grown" out of the accepting leaves in an inductive construction, the accepting region is, in this case, nevertheless restricted to the single accepting leaf $(Lam'\ \mathsf{end}, q_0)$. Consequently, the type inferred from this leaf is $Lam' : q_0 \to q_0$, representing the fact that we now know that when Lam' is applied to a term of acceptance type q_0 (namely end), it will generate a tree which is accepted from state q_0.

Hence, the context for the next iteration will be:

$$C_1 = \langle \mathsf{A} \uplus \{Lam' : q_0 \to q_0\}, \mathsf{R} \uplus \{Lam : \top \to q_0\} \rangle$$

so that Lam and Lam', which were confused on this iteration, now have different types and hence will be distinguished by the next.

A.3 Iteration 2

The graph for the second iteration $\mathsf{ACG}(C_1)$ is shown in Figure 6. The binding set for the graph is as shown below.

$$y4 \mapsto Lam' \quad y1 \mapsto C3 \quad y0 \mapsto Id$$
$$y3 \mapsto Lam \quad y1 \mapsto C2\ y0$$

In this iteration, there are still spurious behaviours in the graph since, although the first parameter in the calls to Id are now distinguished by their type, the two terms $C3$ and $C2\ y0$, which appear as the second actual parameter in calls to Id are confused since neither has any associated intersection type information. In particular, as a result of this confusion the configuration $(C3\ y3, q_0)$ appears in the graph. This configuration is a rejecting leaf since, if contracted, it would yield a configuration $(Lam\ \mathsf{end})$ which is C_1-rejecting. From this leaf, the new rejection type $C3 : (\top \to q_0) \to q_0$ is added to the context. The accepting region dominates the bottom left corner of the graph as it is depicted in the figure. From the accepting region is extracted the set of acceptance types:

$C3 : (q_0 \to q_0) \to q_0$
$C2 : ((q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0) \to (q_0 \to q_0) \to q_0$
$C2 : ((q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0) \to \top \to q_0$
$Id : (q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0$
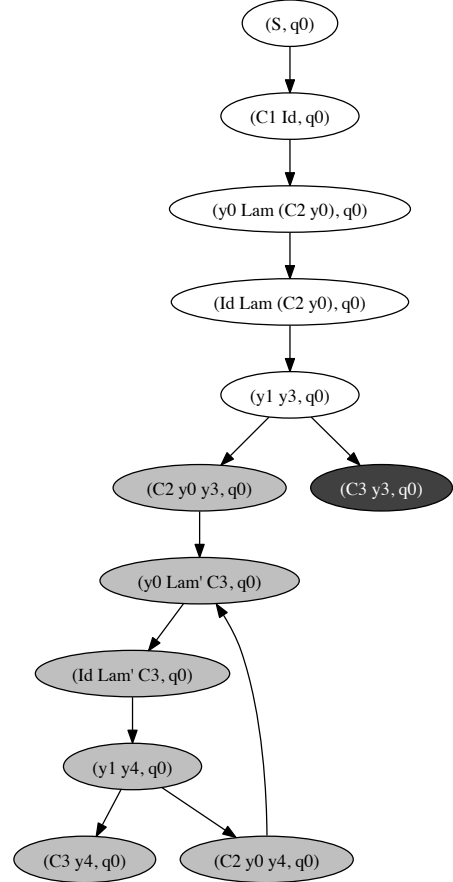
Figure 6. Abstract configuration graph for iteration 2.

This ensures that in the following iteration, the term $C3$ and the term $C2\ y0$ will no longer be confused, since in context C_2 the first has rejection type $(\top \to q_0) \to q_0$, whereas the second has no rejection type.

A.4 Iteration 3

The third iteration is the last (non-trivial) iteration of the algorithm when running on this instance. The graph $\mathsf{ACG}(C_2)$ is depicted in Figure 7. In this case, the associated binding set is simply $y5 \mapsto Id$. Here, the configuration $(Id\ Lam\ (C2\ y5), q_0)$ is an accepting leaf since, if it were to be contracted it would yield the configuration $(C2\ y5\ Lam)$ which is already known to be C_2-accepting. Hence the construction of the graph is halted at this point. Since there are
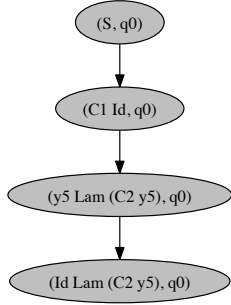
Figure 7. Abstract configuration graph for iteration 3.

no rejecting leaves, it follows that the accepting region is universal. Consequently, among the types extracted from the region is the typing $S : q_0$ which concludes the run of the algorithm with the answer YES.

B. Proofs of claims in the text

The structure of this appendix is as follows. In A.1 we have a number of results about the type assignment system, which will be used frequently in the proofs of claims of the text. In A.2 we have a result about the well-kindedness of the ACG which will be useful in showing the finiteness of all the constructions. Following A.2 we devote one subsection to each of the stated lemmas and theorems from the main text, in the order that they appear.

B.1 Results about the intersection type system.

We begin with a number of useful theorems about intersection type assignment. Although our intersection type assignment system is a novel variation (designed for type checking efficiency), the results are quite natural and so we will not typically refer to them explicitly in what follows.

Lemma 8. *Type assignment preserves refinement:*

$$\Gamma \vdash t : \tau \quad and \quad \Gamma :: \Delta \quad and \quad \Delta \vdash t : \kappa \quad implies \quad \tau :: \kappa$$

Proof. By induction on the intersection type derivation.

- When concluded by (T-VAR) or (T-FUN), the result follows from the second premise.
- When concluded by (T-CST), the result follows by definition.
- When concluded by (T-APP), necessarily t is of the form uv and $\Gamma \vdash u : \sigma \to \tau$ for some σ. It follows from the third premise that $\Delta \vdash u : \kappa_1 \to \kappa$ for some κ_1. Hence, it follows from the induction hypothesis that $\sigma \to \tau :: \kappa_1 \to \kappa$ and, by definition, $\tau :: \kappa$.

□

This allows us to give a very straightforward algorithm for computing the set of strict types assignable to a given term in a given environment. Let Γ be an intersection type environment and t a term. We define the set of strict types $\mathbb{T}(\Gamma)(t)$ by the following clauses:

$$
\begin{aligned}
\mathbb{T}(\Gamma)(x) &= \Gamma(x) \\
\mathbb{T}(\Gamma)(F) &= \Gamma(F) \\
\mathbb{T}(\Gamma)(c) &= \{\bigwedge(S|_1) \to \cdots \to \bigwedge(S|_n) \to q \mid S \models \delta(c, q)\} \\
\mathbb{T}(\Gamma)(s\,t) &= \{\tau \mid \exists \sigma \to \tau \in \mathbb{T}(\Gamma)(s) \cdot \bigwedge \mathbb{T}(\Gamma)(t) \le \sigma\}
\end{aligned}
$$

This algorithmic characterisation of type assignment makes the following proposition clear:

Proposition 2. *Let Γ be an environment, τ a strict type and u and v terms. Then $\Gamma \vdash u\,v : \tau$ iff there is some type σ such that $\Gamma \vdash u : \sigma \to \tau$ and $\bigwedge \mathbb{T}(\Gamma)(v) \le \sigma$.*

A disadvantage of the system is that it does not admit the usual subtyping rule, in general. However, the following weakening property suffices for most purposes.

Lemma 9 (Weakening). *Let Γ and Γ' be such that $\Gamma \le \Gamma'$. It follows that if $\Gamma' \vdash t : \tau$ then there is some τ' such that $\Gamma \vdash t : \tau'$ and $\tau' \le \tau$.*

Proof. By induction on the type derivation.

- When the derivation is concluded by (T-FUN) or (T-VAR) then t is some symbol ξ and it follows that $\tau \in \Gamma'(\xi)$. Since $\Gamma \le \Gamma'$, by definition, $\Gamma(\xi) \le \Gamma'(\xi)$ and hence the result follows.
- When the derivation is concluded by (T-CST), it is independent of the environment and so take witness τ' to be τ.
- When the derivation is concluded by (T-APP), t is of the form $u\,v$ and there are two intersection types σ_1 and σ_2 such that $\Gamma' \vdash u : \sigma_1 \to \tau$, $\Gamma' \vdash v : \tau''$ for each $\tau'' \in \sigma_2$ and $\sigma_2 \le \sigma_1$. It follows from the induction hypothesis that there is some type $\sigma_3 \to \tau'$ with, by (Q-ARR), $\sigma_2 \le \sigma_3$ and $\tau' \le \tau$ and also $\Gamma \vdash u : \sigma_3 \to \tau'$. It follows that $\Gamma \vdash u\,v : \tau'$ as required.

□

From this we obtain Proposition 3 as a straightforward corollary. It also follows that type assignment is monotone with respect to subtyping of environments:

Corollary 1. *Let Γ and Γ' be such that $\Gamma \le \Gamma'$. For all terms t:*

$$\bigwedge \mathbb{T}(\Gamma)(t) \le \bigwedge \mathbb{T}(\Gamma')(t)$$

Proposition 3. *if $\Gamma \vdash t : q$ and $\Gamma' \le \Gamma$ then $\Gamma' \vdash t : q$*

B.2 Well kindedness of the ACG

We begin with a useful result about the well kindedness of the ACG. It is the observation that the whole ACG construction preserves kindedness with respect to the kind environment $\mathcal{N} \cup \mathsf{K}$. This environment specifies the kind of all the non-terminal symbols through $\mathcal{N}$ and also of all the (disjoint set of) typed variables introduced as part of the abstraction through K.

Lemma 10. *Let C be a type context. For each mapping $y \mapsto t \in B_C$, $\mathcal{N} \cup \mathsf{K} \vdash t : \mathsf{K}(y)$ and, for each $v \in V_C$, exactly one of the following is true:*

(i) v is a configuration (t, q) and $\mathcal{N} \cup \mathsf{K} \vdash t : o$.
(ii) v is a set $\{(t_1, q_1), \ldots, (t_n, q_n)\}$ and, for each $i \in [1..n]$, $\mathcal{N} \cup \mathsf{K} \vdash t_i : o$.

Proof. By induction on the structure of $\mathsf{ACG}(C)$:

(ACG1) By definition, $S : o \in \mathcal{N}$.
(ACG2) It follows from the induction hypothesis that

$$\mathcal{N} \cup \mathsf{K} \vdash F\,s_1 \cdots s_n : o$$

and thus that F has some type $\kappa_1 \to \cdots \to \kappa_n \to o$ in $\mathcal{N}$. Since the scheme is well kinded, also:

$$\mathcal{N} \cup \{x_i : \kappa_i \mid i \in [1..n]\} \vdash t : o$$

and, for each $i \in [1..n]$, $\mathsf{K}(y_i) = \kappa_i$. Hence (i) follows from the substitution lemma for simple typing. To show the property of the mapping, notice that for each $i \in [1..n]$, necessarily $\mathcal{N} \cup \mathsf{K} \vdash s_i : \kappa_i$ and $\mathsf{K}(y_i) = \kappa_i$.

(ACG3) It follows from the induction hypothesis that:

$$\mathcal{N} \cup \mathsf{K} \vdash a \, s_1 \, \cdots \, s_n : o$$

Since then necessarily a has arity n, it follows that, for each $i \in [1..n]$: $\mathcal{N} \cup \mathsf{K} \vdash s_i : o$ and (ii) follows.

(ACG4) Here (i) follows directly from the induction hypothesis.

(ACG5) It follows from the induction hypothesis that

$$\mathcal{N} \cup \mathsf{K} \vdash y \, s_1 \, \cdots \, s_n : o$$

and $\mathcal{N} \cup \mathsf{K} \vdash t : \mathsf{K}(y)$. Hence, (i) follows by the substitution lemma for simple typing. $\square$

B.3 Proof of Lemma 1

We prove the following stronger statement:

Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. For each $v \in V_C$, exactly one of the following is true:

(i) v is a configuration and v is C-unknown.

(ii) v is a set of configurations and both of the following are true:

(a) there exists $c \in v$ such that c is not C-accepted

(b) for all $c \in v$, c is not C-rejected

Proof. By induction on the structure of $\mathsf{ACG}(C)$.

(ACG1) By the precondition (S, q_0) is trivially C-unknown.

(ACG2) By the precondition $(t[y_1/x_1, \ldots, y_n/x_n], q)$ is trivially C-unknown.

(ACG3) By the precondition, for each $(i, q') \in S$, (s_i, q') is not C-rejected. Assume for contradiction that, for all $(i, q') \in S$, (s_i, q') is C-accepted. It follows that then $\Gamma_\exists \vdash_{\mathcal{A}} a \, s_1 \, \cdots \, s_n : q$ which contradictions the induction hypothesis.

(ACG4) It follows from the induction hypothesis that, for each $j \in [1..n]$, (s_j, q_j) is not C-rejected. By the precondition also (s_i, q_i) is not C-accepted. Hence (s_i, q_i) is C-unknown.

(ACG5) It follows from the induction hypothesis that $(y s_1 \cdots s_n, q)$ is C-unknown. Since, additionally, $\mathbb{T}(\Gamma_\forall)(y) = \mathbb{T}(\Gamma_\forall)(t)$ and $\mathbb{T}(\Gamma_\exists)(y) = \mathbb{T}(\Gamma_\exists)(t)$ it follows that also $(t s_1 \cdots t_n, q)$ is C-unknown. $\square$

B.4 Proof of Lemma 2

Simple and compound terms. Call a term s *simple* just if there is a non-terminal F such that $F = \lambda x_1 \cdots x_n . t$ and s is a subterm of t (possibly by a renaming of variables). Call a term *compound* if it is of the form $\xi s_1 \cdots s_n$ with ξ a terminal, non-terminal or typed variable and, for each $i \in [1..n]$, s_i is simple.

Lemma 11. *Let C be a type context. Then all of the following are true:*

(i) *For each $y \mapsto t \in B_C$, t is simple.*

(ii) *For each configuration $(t, q) \in V_C$, t is compound.*

(iii) *For each set $\{(s_1, q_1), \ldots, (s_n, q_n)\}$ and each $i \in [1..n]$, s_i is simple.*

Proof. By induction on the structure of $\mathsf{ACG}(C)$.

(ACG1) Clearly, S is compound.

(ACG2) It follows from the induction hypothesis that $F s_1 \cdots s_n$ is compound, so each s_i is simple. Clearly $t[y_1/x_1, \ldots, y_n/x_n]$ is simple (and hence compound).

(ACG3) It follows from the induction hypothesis that $a \, s_1 \, \cdots \, s_n$ is compound and hence each s_i is simple.

(ACG4) It follows from the induction hypothesis that, for each $j \in [1..n]$, s_j is simple and hence s_i is simple (and hence compound).

(ACG5) It follows from the induction hypothesis that $y \, s_1 \, \cdots \, s_n$ is compound and hence each s_i is simple and also that t is simple. Hence $t \, s_1 \, \cdots \, s_n$ is compound.

$\square$

We can now prove the statement in the lemma:

Let C be a type context. Then the size of $\mathsf{ACG}(C)$ is bounded by a polynomial function of the size of the scheme.

Proof. We assume that the rules of the input HORS are normalised to the form $F = \lambda \overline{x}.\xi \, (F_1 \, \overline{x}_1) \cdots (F_m \, \overline{x}_m)$, where ξ may be a terminal, non-terminal or variable, and m may be 0. We shall call the sequence of terms $(F_1 \, \overline{x}_1) \cdots (F_m \, \overline{x}_m)$ the *tail* of F. To obtain such a normalised HORS, it suffices to replace each rule of the form $F = \lambda \overline{x}.\xi \, t_1 \cdots t_i \cdots t_n$, where t_i is not of the form $F_i \, \overline{x}_i$, with $F = \lambda \overline{x}.\xi \, t_1 \cdots (H \, \overline{x}') \cdots t_n$ and add the new rule: $H = \lambda \overline{x}'.\lambda \overline{y}.t_i \, \overline{y}$, where H is a fresh non-terminal, $\{\overline{x}'\}$ is the subset of variables of $\{\overline{x}\}$ that occur in t_i, and $\overline{y}$ is a sequence of variables added to ensure that $t_i \, \overline{y}$ has kind o. Let $|\mathcal{G}_0|$ and A_0 be the size and the largest arity of the original recursion scheme. Since at most $|\mathcal{G}_0|$ replacements are required to normalise the scheme, the size and the largest arity of the normalised scheme is $O(|\mathcal{G}_0|A_0)$ and $2A_0$ respectively (the increase of the arity by A_0 is due to the extra parameters $\overline{y}$ above). Thus the complexity result obtained below is not affected by the normalisation.

Assume that the set of kinds that appear in the input normalised HORS $\mathcal{G}$ is contained in a fixed set $\mathcal{S}$. Let $|\mathcal{G}|$, A, N and P respectively be the size, largest arity, order and number of rules of $\mathcal{G}$. Let $|Q|$ be the number of states of the ATT. An upper-bound of the number of strict types of order n, written K_n, can be given inductively: $K_0 = |Q|$ and $K_{n+1} = |Q| \, 2^{A \, K_n}$. Thus $K_n = \mathbf{exp}_n(O(A \, |Q|))$.

It follows from the definition of $\mathcal{Y}$ that $|\mathcal{Y}| = O(K_{N-1}^2 \, |\mathcal{S}|)$. Take an element $y \mapsto t$ of B_C. Because $\mathcal{G}$ is normalised, the (simple) term t is either a variable or has the form $F \, \overline{x}$, where the variables are elements of $\mathcal{Y}$. Thus there are $O(P \, |\mathcal{Y}|^A)$ such simple terms t. It follows from Lemma 11(i) that $|B_C| = O(P \, |\mathcal{Y}|^{A+1})$.

Take an element $(t, q) \in V_C$. Because $\mathcal{G}$ is normalised, t has the form $\zeta \, (F_1 \, \overline{x}_1) \cdots (F_m \, \overline{x}_m)$ where the sequence $(F_1 \, \overline{x}_1) \cdots (F_m \, \overline{x}_m)$ is the tail of some non-terminal, and ζ is either a variable or of the form $F \, \overline{x}$. There are $O(|\mathcal{Y}| \, P \, |\mathcal{Y}|^{A^2})$ and $O(P^2 \, |\mathcal{Y}|^{A^2})$ compound terms t of the respective forms. Thus, there are $O(P^2 \, |\mathcal{Y}|^{A^2+1})$ such compound terms t. Next we compute an upper bound of the number of elements of V_C that are sets of abstract configurations. Since $\mathcal{G}$ is normalised, terminal-headed terms that arise in the construction of $\mathsf{ACG}(C)$ are necessarily the righthand side of a rule $F \, \overline{x} = a \, (F_1 \, \overline{x}_1) \cdots (F_m \, \overline{x}_m)$, with variables taken from $\mathcal{Y}$. There are $O(P \, |\mathcal{Y}|^{A^2})$ such terms. Since there are $O(2^{A \, |Q|})$ satisfying assignments for each positive boolean expression of the ATT, there are $O(2^{A \, |Q|} \, P \, |\mathcal{Y}|^{A^2})$ sets of abstract configurations in V_C. It follows that $|V_C| = O((P \, |\mathcal{Y}| + 2^{A \, |Q|}) \, P \, |\mathcal{Y}|^{A^2})$. $\square$

B.5 Proof of Lemma 3

We shall need a well-known fact about the complement automaton:

Lemma 12. *Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ be an ATT, then:*

$$S \models \delta^c(q, a) \qquad \textit{iff} \qquad \forall S' \cdot S' \models \delta(q, a) \textit{ implies } S' \cap S \neq \emptyset$$

We can now prove the statement of the lemma:

> Let C be a type context. Every leaf $v \in V_C$ is either accepting or rejecting.

Proof. Let v be a leaf in $\mathsf{ACG}(C)$. We distinguish cases on the form of v.

- If v is of the form $(F\ s_1 \cdots s_n,\ q)$ then, by (ACG2), it must be that v is either C-accepted or C-rejected and hence v is either an accepting or rejecting leaf (respectively).
- If v is of the form $(a\ s_1 \cdots s_n,\ q)$ then, by (ACG3), it must be that for every satisfying assignment S to $\delta(q, a)$ there is some $(i\ q') \in S$ such that (s_i, q') is C-rejected. However, this is inconsistent with $v \in V_C$ since, it follows from Lemma 12 that then $\bigcup_{i=1}^{n}\{(i\ q') \mid \Gamma_\forall \vdash_{\mathcal{A}^c} s_i : q'\}$ is a satisfying assignment to $\delta^c(q, a)$. Consequently $\Gamma_\forall \vdash_{\mathcal{A}^c} a\, s_1 \cdots s_n : q$ which contradicts Lemma 1 (i).
- If v is of the form $\{(s_1, q_1), \ldots, (s_n, q_n)\}$ then, by (ACG3)) it must be that, for all $i \in [1..n]$, (s_i, q_i) is C-accepted. But this contradicts Lemma 1 part (ii)(b).
- If v is of the form $(y\ s_1 \cdots s_n,\ q)$ then, by construction, there is some $y \mapsto t \in B_C$ and consequently, v being a leaf is inconsistent with the closedness of $\mathsf{ACG}(C)$ under (ACG5).

$\square$

B.6 Proof of Lemma 6

The statement is as follows:

> Let C be a type context and $\mathsf{ACG}(C)$ have no rejecting leaves. Then:
> $$\mathsf{ACG}(C) = \mathsf{RA}(C)$$

Proof. Since, by definition, $\mathsf{RA}(C)$ is a subgraph of $\mathsf{ACG}(C)$, the result follows by coinduction on the structure of $\mathsf{RA}(C)$.

(A1) Assume $(F\ s_1 \cdots s_n,\ q)$ is not an accepting leaf, since, by assumption, there are no rejecting leaves, it is unknown. It follows from the coinduction hypothesis that $(F\ s_1 \cdots s_n,\ q) \in V_C$, hence the result follows by (ACG2).

(A2) It follows from the coinduction hypothesis that $(a\,s_1 \cdots s_n,\ q) \in V_C$, it follows by Lemma 3 that the only leaves in $\mathsf{ACG}(C)$ are accepting or rejecting so this vertex has a successor. It follows by (ACG3) that necessarily this successor is a set $\{(s_i, q') \mid (i, q) \in S\}$ for some satisfying assignment S to $\delta(q, a)$.

(A3) It follows from the induction hypothesis that, in this case, $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in V_C$. It follows from ((G)) that if $i \in [1..n]$ then $(s_i, q_i) \in V_C$ is a child.

(A4) It follows from the coinduction hypothesis that $(y s_1 \cdots s_n,\ q) \in V_C^{\mathsf{A}}$. Since $y \mapsto t \in B_C$, the result follows from (ACG5).

$\square$

B.7 Proof of Lemma 4

In preparation for the proof of Lemma 4, we first show that the environment extracted from a witnessing tree types the configuration that it witnesses. More formally:

Lemma 13. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context. For all $v \in \mathsf{RR}(C)$ with witnessing proof tree T, either:*

(i) v is a configuration (t, q) and $\Gamma_\forall \uplus \mathcal{M}(T) \vdash_{\mathcal{A}^c} t : q$ or

(ii) v is a set $\{(t_1, q_1), \ldots, (t_n, q_n)\}$ and there is some $i \in [1..n]$ such that $\Gamma_\forall \uplus \mathcal{M}(T) \vdash_{\mathcal{A}^c} t_i : q_i$.

Proof. By induction on T. In cases (R1), (R3) and (R5), the result follows by definition. In case (R2), the result follows from the induction hypothesis. It remains to check case (R4). In this case v is of the form $(a\ s_1 \cdots s_n,\ q)$. Let the children be in the set W. Necessarily, for each child $w \in W$ there is a sub-proof T_w and, by (M4), $\mathcal{M}(T) = \biguplus\{\mathcal{M}(T_w) \mid w \in W\}$. It follows that, for each satisfying assignment $S \models \delta(q, a)$, either the corresponding set of configurations $\{(s_i, q') \mid (i, q') \in S\}$ is in W or is already known to be rejecting, in the sense that there is some j such that $\Gamma_\forall \vdash s_j : q'$. It follows from the induction hypothesis that, for each such set w in W, it is also the case that there is some j such that $\Gamma_\forall \uplus \mathcal{M}(T_w) \vdash s_j : q'$. Consequently, for each satisfying assignment S, there is some $(j, q') \in S$ such that $\Gamma_\forall \uplus \biguplus\{\mathcal{M}(T_w) \mid w \in W\} \vdash s_j : q'$. It follows from Lemma 12 that the set of all such pairs (j, q') is a satisfying assignment to $\delta^c(q, a)$ and hence $\Gamma_\forall \uplus \mathcal{M}(T) \vdash a\ s_1 \cdots s_n : q$ as required. $\square$

It is also helpful to point out a consequence of the definition of rejection type extraction in a local situation, with respect to variable, term bindings. Any occurrence of a variable headed vertex in the rejection region must, by rule (R5), be followed by vertices that are prefixed by each of its abstract descendents t such that $y \mapsto t \in B_C$. It follows that, when types are assigned, since each abstract descendent t occurs closer to the root, the environment $\mathcal{M}(T_t)$ associated with it's witness T_t is smaller than the environment $\mathcal{M}(T_y)$ associated with the witness for the variable headed vertex, since $\mathcal{M}(T_y)$ subsumes all of the environments of its descendents. Consequently, any type extracted for y will be extracted in a larger environment than the corresponding type for t. Hence, we have the following:

Lemma 14. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context and let T be the tree witnessing the membership of some vertex $v \in V_C$ in $\mathsf{RR}(C)$, then for all $y \mapsto t \in B_C$:*

$$\bigwedge \mathbb{T}(\mathcal{M}(T))(t) \leq \mathcal{M}(T)(y)$$

Proof. Let t be of the form $\xi t_1 \cdots t_k$. First observe that, for all types $\tau \in \mathcal{M}(T)(y)$, there is some vertex v' of the form $(y\ s_1 \cdots s_n,\ q)$ with witness T' a sub-proof of T and τ is of the form:

$$\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_1) \to q$$

However, by rule (R5), in every case there must also be a vertex v'' of the form $(\xi t_1 \cdots t_k\ s_1 \cdots s_n,\ q)$ with witness T'' a sub-proof of T'. So let τ be such a type. We shall show that, for all choices of ξ, there is some type τ_ξ of the form:

$$\bigwedge T_1 \to \cdots \to \bigwedge T_k \to \bigwedge S_1 \to \cdots \to \bigwedge S_n \to q$$

such that the following conditions hold:

(i) $\Gamma_\forall \uplus \mathcal{M}(T) \vdash_{\mathcal{A}^c} \xi : \tau_\xi$

(ii) for all $i \in [1..k]$, $\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T''))(t_i) \leq \bigwedge T_i$

(iii) for all $i \in [1..n]$, $\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_i) \leq \bigwedge S_i$

It then follows that $\Gamma_\forall \uplus \mathcal{M}(T) \vdash t : \bigwedge S_1 \to \cdots \bigwedge S_n \to q$ and $\bigwedge S_1 \to \cdots \to \bigwedge S_n \to q \leq \tau$.

To this end, distinguish cases on the form of ξ. If ξ is a variable or non-terminal symbol then let τ_ξ be the type extracted from the proof of T''. Then (i) and (ii) hold by definition and (iii) is true since, in this case, S_i is exactly $\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T''))(s_i)$ and $\mathcal{M}(T') \leq \mathcal{M}(T'')$. Otherwise ξ is a terminal symbol c and we let τ_ξ be the type induced by $\delta(q, c)$ according to rule (T-CST). It follows from Lemma 13 that, therefore (i) and (ii) hold and furthermore that $\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T''))(s_i) \leq \bigwedge S_i$. Since $\mathcal{M}(T') \leq \mathcal{M}(T'')$, the result follows transitively. $\square$

We can now prove the following statement, of which Lemma 4 is a corollary, since it should be clear from the definition of co-consistency that the multiplication of a finite number of co-consistent environments is again co-consistent.

> Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and $\Gamma_\forall$ be $(\mathcal{G}, \mathcal{A}^c)$-co-consistent. Then, for all $v \in \mathsf{RR}(C)$ with proof tree T, $\mathcal{M}(T) \uplus \Gamma_\forall$ is $(\mathcal{G}, \mathcal{A}^c)$ co-consistent.

Proof. The proof is by induction on the structure of T.

(R1) In this case v is a configuration $(F\ s_1\ \cdots\ s_n,\ q)$ and $\mathcal{M}(T)$ contains only

$$F : \bigwedge \mathbb{T}(\Gamma_\forall)(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall)(s_n) \to q$$

Let this type be τ. Since $\Gamma_\forall$ is already co-consistent, it remains just to check that there is some $\Gamma \subseteq \mathcal{M}(T) \uplus \Gamma_\forall$ such that $\Gamma \vartriangleright F : \tau$. Take as witness $\Gamma_\forall$ since, by definition, v is a rejecting leaf and hence its successor $(\mathcal{R}(F)[y_1/x_1, \ldots, y_n/x_n],\ q)$ is typable in the environment $\Gamma_\forall$. Due to 1, for each $i \in [1..n]$, $\Gamma_\forall(y_i) = \bigwedge \mathbb{T}(\Gamma_\forall)(s_i)$ and so the result follows by a renaming of variables.

(R2) In this case v is a set $\{(s_1, q_1), \ldots, (s_n, q_n)\}$ and there is an immediate sub-proof T' of (s_j, q_j). Since, by (M2) $\mathcal{M}(T) = \mathcal{M}(T')$, the result follows from the induction hypothesis.

(R3) In this case v is a configuration of the form $(F\ s_1\ \cdots\ s_n,\ q)$ and there is an immediate sub-proof T' for the configuration $(\mathcal{R}(F)[y_1/x_1, \ldots, y_n/x_n],\ q)$. By (M3) the environment $\mathcal{M}(T)$ consists of $\mathcal{M}(T')$ augmented with the binding:

$$F : \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_n) \to q$$

Let this type be τ. It follows from the induction hypothesis that $\Gamma_\forall \uplus \mathcal{M}(T')$ is already co-consistent and, from the above Lemma 13, that $\Gamma_\forall \uplus \mathcal{M}(T')$ is sufficient to prove $\mathcal{R}(F)[y_1/x_1, \ldots, y_n/x_n] : q$ (*). Hence, to see that there is some $\Gamma \subseteq \Gamma_\forall \uplus \mathcal{M}(T)$ such that $\Gamma \vartriangleright F : \tau$ take as witness $\Gamma_W := \Gamma_\forall \uplus \mathcal{M}(T')$. It remains to show that:

$$\Gamma_W \uplus \{x_i : \bigwedge \mathbb{T}(\Gamma_W)(s_i) \mid i \in [1..n]\} \vdash_{\mathcal{A}^c} \mathcal{R}(F) : q$$

Due to (*) it suffices to show only that, for all $i \in [1..n]$, $\bigwedge \mathbb{T}(\Gamma_W)(s_i) \leq \Gamma_W(y_i)$. To this end, consider some $\tau \in \Gamma_W(y_i)$, it must be that either (i) $\tau \in \Gamma_\forall(y_i)$ or (ii) $\tau \in \mathcal{M}(T')(y_i)$. In the former case, also $\tau \in \mathbb{T}(\Gamma_\forall)(s_i)$ by Proposition 1. In the latter case, observe that, by Lemma 14, $\bigwedge \mathbb{T}(\mathcal{M}(T'))(s_i) \leq \mathcal{M}(T')(y_i)$ and $\Gamma_W \leq \mathcal{M}(T')$ and the result follows transitively.

(R4) In this case v is of the form $(a\ s_1\ \cdots\ s_n,\ q)$. Let the children be in the set W. Necessarily, for each child $w \in W$ there is a sub-proof T_w and by (M4) $\mathcal{M}(T) = \biguplus\{\mathcal{M}(T_w) \mid w \in W\}$. This environment is co-consistent since by the induction hypothesis, for all $w \in W$, $\Gamma_\forall \uplus \mathcal{M}(T_w)$ is co-consistent.

(R5) In this case v is of the form $(y\ s_1\ \cdots\ s_n,\ q)$ and for each $y \mapsto t$ in B_C, there is a sub-proof T_t of $(t\ s_1\ \cdots\ s_n,\ q)$. The environment $\mathcal{M}(T)$ is given by $\mathcal{M}(T_y)$ augmented with:

$$y : \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T_y))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T_y))(s_n) \to q$$

It follows from the induction hypothesis that $\Gamma_\forall \uplus \mathcal{M}(T_y)$ is co-consistent whence $\Gamma_\forall \uplus \mathcal{M}(T)$ is co-consistent. $\qquad\square$

B.8   Proof of Lemma 5

The statement of the Lemma is as follows:

> Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and $\mathsf{ACG}(C)$ have some rejecting leaf. Then:
> $$\mathsf{env}_\mathsf{R}(C) \setminus \Gamma_\forall \neq \emptyset$$

Proof. Let $v = (F\ s_1\ \cdots\ s_n,\ q)$ be the rejecting leaf, by Lemma 1, v is itself not C-rejected. By (RR1) v is also a leaf in $\mathsf{RR}(C)$, let its witnessing proof tree be T. By the strengthened Lemma 4 proved above, $\mathcal{M}(T) \uplus \Gamma_\forall \vdash_{\mathcal{A}^c} F\ s_1\ \cdots\ s_n : q$ and by definition $\mathcal{M}(T)$ is the single binding

$$F : \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T'))(s_n) \to q$$

which we shall write simply as $F : \tau$ for brevity. Since v is not itself C-rejected, necessarily $\tau \notin \Gamma_\forall(F)$ and so the result follows. $\qquad\square$

B.9   Proof of Lemma 7

In preparation for the proof of Lemma 7, it is helpful to point out some simple consequences of the definition of acceptance type extraction in certain local situations.

The situation for variables.

Proposition 4. *Let C be a context and $v = (y\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RA}(C)$ and $v' = (t\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RA}(C)$, then:*

$$\mathsf{extr}(v',\ t) = \mathsf{extr}(v,\ y)$$

Proof. By definition, since in each case the prefix is applied to the same arguments. $\qquad\square$

The situation for non-terminals.

Proposition 5. *Let C be a context with $(F\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RA}(C)$ and $(t[y_1/x_1,\ \ldots,\ y_n/x_n],\ q) \in \mathsf{RA}(C)$. For all $i \in [1..n]$:*

$$\{\mathsf{extr}(v'',\ y_i) \mid v'' \in W_{y_i}\} \subseteq \{\mathsf{extr}(v'',\ s_i) \mid v'' \in W_{s_i}\}$$

where W_t is the set of configurations in $\mathsf{RA}(C)$ that have t as a prefix.

Proof. By rule (A4), since necessarily $y_i \mapsto s_i \in B_C$ and further by Proposition 4. $\qquad\square$

Since we have carefully chosen the closure conditions that define the acceptance region, the proof of Lemma 7 can follow the same format as the proof of Kobayashi for type extraction from the (infinite) concrete configuration graph. We first prove that the environment extracted is sufficient to type all the prefixes of the vertices that it is extracted from. First let us define an auxilliary environment which extends $\mathsf{env}_\mathsf{A}(C)$ by retaining the variable typings which were disregarded according to the definition in the text:

$$\mathsf{env}_\mathsf{A}^+(C)(x) = \bigwedge\{\tau \mid \exists c \in \mathsf{RA}(C) \cdot \mathsf{extr}(c,\ x) = \tau\}$$

letting x range over variables or non-terminals.

Lemma 15. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context. For each term s and vertex $v \in \mathsf{RA}(C)$ such that (v, s) is a prefix:*

$$\Gamma_\exists \uplus \mathsf{env}_\mathsf{A}^+(C) \vdash_\mathcal{A} s : \mathsf{extr}(v,\ s)$$

Proof. By induction on s.

- When s is a terminal a, then v is of the form $(a\ s_1 \cdots s_n,\ q)$ and, by (A2), there is some satisfying assignment S to $\delta(q,\ a)$ and a child $v' = \{(s_i,\ q') \mid (i,\ q') \in S\}$. But then by (A3), for each $(i,\ q') \in S$, $(s_i,\ q') \in \mathsf{RA}(C)$ (*). By definition $\mathsf{extr}(v,\ a)$ is of the form:

$$\bigwedge S_1 \wedge \bigwedge T_1 \to \cdots \to \bigwedge S_n \wedge \bigwedge T_n \to q$$

where, for each $i \in [1..n]$, S_i is the set $\mathbb{T}(\Gamma_\exists)(s_i)$ and T_i is the set of all types $\mathsf{extr}(v'', s_i)$ such that (v'', s_i) is prefix in $\mathsf{RA}(C)$. Notice that, by (*), for each $(i, q') \in S$, $q' \in T_i$. Consequently:

$$\bigcup_{i=1}^{n} \{(i, q) \mid q \in S_i \cup T_i\} \models \delta(q, a)$$

and the result follows from (T-Cst).

- When s is a non-terminal F, then v is of the form $(F\, s_1 \cdots s_n,\, q)$. By definition $\mathsf{extr}(v, F) \in \mathsf{env}_\mathsf{A}^+(C)(F)$ and the result follows from (T-Fun).
- When s is a variable y, then v is of the form $(y\, s_1 \cdots s_n,\, q)$. By definition $\mathsf{extr}(v, y) \in \mathsf{env}_\mathsf{A}^+(C)(y)$ and the result follows from (T-Var).
- When s is an application $t_1\ t_2$ then (v, t_1) is also a prefix and so it follows from the induction hypothesis that $\mathsf{env}_\mathsf{A}^+(C) \vdash_\mathcal{A} t_1 : \mathsf{extr}(v, t_1)$. By definition $\mathsf{extr}(v, t_1)$ is of the form:

$$\bigwedge \mathbb{T}(\Gamma_\exists)(t_2) \ \wedge\ \bigwedge_{v' \in W} \mathsf{extr}(v', t_2) \quad \rightarrow \quad \mathsf{extr}(v, t_1\ t_2)$$

with W the set of all configurations that have t_2 as a prefix. For each such v', since (v', t_2) is a prefix, it follows from the induction hypothesis that $\mathsf{env}_\mathsf{A}^+(C) \vdash_\mathcal{A} t_2 : \mathsf{extr}(v', t_2)$. The result follows from (T-App).

$\square$

We can now conclude with the proof of Lemma 7. The lemma is stated as follows:

Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context. If $\Gamma_\exists$ is $(\mathcal{G}, \mathcal{A})$-consistent then also $\Gamma_\exists \uplus \mathsf{env}_\mathsf{A}(C)$ is $(\mathcal{G}, \mathcal{A})$-consistent.

Proof. Let $\tau \in (\Gamma_\exists \uplus \mathsf{env}_\mathsf{A}(C))(F)$ and $\mathcal{R}(F) = \lambda x_1 \cdots x_n.\, t$. If $\tau \in \Gamma_\exists(F)$ then the result follows by assumption. Otherwise, there is a vertex $v \in \mathsf{RA}(C)$ of the form $(F\, s_1 \cdots s_n,\, q)$ and $\tau = \mathsf{extr}(v, F)$. We distinguish two cases.

(i) Either, by (A5), v is an accepting leaf so that by definition there is some choice of typed variables $\{y_i \mid i \in [1..n]\}$ such that, for all $i \in [1..n]$, $\Gamma_\exists(y_i) = \bigwedge \mathbb{T}(\Gamma_\exists)(s_i)$ and also $\Gamma_\exists \vdash t[y_1/x_1, \ldots, y_n/x_n] : q$. By definition then, $\mathsf{extr}(v, F)$ is exactly:

$$\bigwedge \mathbb{T}(\Gamma_\exists)(s_1) \rightarrow \cdots \rightarrow \bigwedge \mathbb{T}(\Gamma_\exists)(s_n) \rightarrow q$$

and the result follows by a renaming of variables.

(ii) Otherwise, by (A1), v has a child v' that is of the form $(t[y_1/x_1, \ldots, y_n/x_n],\, q)$ and it follows from Lemma 15 that $\mathsf{env}_\mathsf{A}^+(C) \vdash_\mathcal{A} t[y_1/x_1, \ldots, y_n/x_n] : q$. Now, for each $i \in [1..n]$, it follows from Proposition 5 that in this situation,

$$\mathsf{env}_\mathsf{A}^+(C)(y_i) \subseteq \{\mathsf{extr}(v'', s_i) \mid v'' \in W_{s_i}\}$$

Since the required environment assumes for each $i \in [1..n]$, $x_i : \bigwedge \mathbb{T}(\Gamma_\exists)(s_i) \wedge \bigwedge \{\mathsf{extr}(v'', s_i) \mid v'' \in W_{s_i}\}$ these assumptions on each x_i are stronger than the assumptions on y_i required to prove the statement $t[y_1/x_1, \ldots, y_n/x_n] : q$. Hence the result follows from weakening and a renaming of variables.

$\square$

B.10 Proof of Theorem 2

The proof is broken into three parts. The first is in showing that the sequence of type contexts stabilizes after a finite number of iterations — in other words, that the algorithm terminates. The second part is in showing that the limit of the sequence $\langle \Gamma_\exists, \Gamma_\forall \rangle$ is a pair comprising a consistent and co-consistent environment respectively. The third part shows that eventually one of the contexts must contain an environment in which S is assigned q_0.

The sequence must eventually stabilize. The stabilization of the sequence of type contexts is guaranteed by the fact that every such context is a pair containing two environments that are both refinements of the kind environment $\mathcal{N} \cup \mathcal{K}$. Since there are only finitely many such refinements, and since the sequence is clearly monotone, it must come to an end after a finite number of iterations. So in the following we show that every environment in the sequence is a refinement. By definition, the initial environments A and R are refinements of K. We proceed by showing that any typings that are added to these environments are also refinements.

Lemma 16. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a context with $\Gamma_\exists :: \mathcal{N} \cup \mathsf{K}$. If (v, s) is a prefix in $\mathsf{RA}(C)$ with $\Delta \vdash s : \kappa$, then $\mathsf{extr}(v, s) :: \kappa$.*

Proof. By induction on the kind κ.

- If $\kappa = o$, then v is of the form (t, q) and $\mathsf{extr}(v, s) = q$. By definition $q :: o$.
- If κ is an arrow kind $\kappa_1 \rightarrow \kappa_2$, then v is a configuration $(s\, t_1 \cdots t_n,\, q)$ and $\Delta \vdash t_1 : \kappa_1$ and $\Delta \vdash s\, t_1 : \kappa_2$. By definition $\mathsf{extr}(v, s)$ is of the form $\bigwedge T \rightarrow \mathsf{extr}(v, s\, t_1)$. It follows from the induction hypothesis that $\mathsf{extr}(v, s\, t_1) :: \kappa_2$ (*). Now let $\tau \in T$ and consider cases. If $\tau \in \mathbb{T}(\Gamma_\exists)(t_1)$ then, by assumption and Lemma 8, $\tau :: \kappa_1$. Otherwise, there is a prefix (v', t_1) such that $\tau = \mathsf{extr}(v', t_1)$ and it follows from the induction hypothesis that $\tau :: \kappa_1$. Hence $\bigwedge T :: \kappa_1$ and with (*) the result follows.

$\square$

It follows from Lemma 10 that every such prefix will have a kind consistent with $\mathcal{N} \cup \mathsf{K}$. Hence one can show by induction that every $\Gamma_\exists$ featured in a context is a refinement of $\mathcal{N} \cup \mathsf{K}$. The same can be said of each $\Gamma_\forall$, since:

Lemma 17. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ with $\Gamma_\forall :: \mathcal{N} \cup \mathsf{K}$. Let $v \in \mathsf{RR}(C)$ with witness T. It follows that $\mathcal{M}(T) :: \mathcal{N} \cup \mathsf{K}$.*

Proof. By induction on T. If T is concluded by any of (M1), (M3) or (M5) then the new type is a refinement of the kind of F or y as a result of Lemma 8 and Lemma 10 and the well kindedness of the rest of the environment follows from the induction hypothesis. If T is concluded by (R4), then observe that the new type is well-kinded by definition and the rest of the environment is well kinded as a result of the induction hypothesis. Otherwise T is concluded by (M2), there is no new type extracted and so the result simply follows from the induction hypothesis. $\square$

Lemma 18. *The sequence of type contexts $(C_i)_{i \in \mathbb{N}}$ is eventually stable.*

Proof. Since every type extracted from the acceptance region is well kinded according to $\mathcal{N} \cup \mathsf{K}$ (Lemma 16) and every type extracted from the rejection region is well kinded according to $\mathcal{N} \cup \mathsf{K}$ (Lemma 17), it follows that for each $i \in \mathbb{N}$, $\Gamma_{\forall_i} :: \mathcal{N} \cup \mathsf{K}$ and $\Gamma_{\exists_i} :: \mathcal{N} \cup \mathsf{K}$. Since there are only finitely many type environments that refine $\mathcal{N} \cup \mathsf{K}$, there is some $i \in \mathbb{N}$ such that $C_i = C_{i+1}$. $\square$

Every context is (co)-consistent. To see that every context $\langle \Gamma_\exists, \Gamma_\forall \rangle$ in the sequence is such that $\Gamma_\exists$ is consistent and $\Gamma_\forall$ is co-consistent, observe that the inital context is trivially so and that since type extraction preserves this property and that since this property is preseved by type environment multiplication, so it holds of every context.

Lemma 19. *For each $i \in \mathbb{N}$, if $C_i = \langle \Gamma_\exists, \Gamma_\forall \rangle$ then $\Gamma_\exists$ is consistent and $\Gamma_\forall$ is sound.*

Proof. By induction on i.

- If $i = 0$, then $\Gamma_\exists^0$ is trivially consistent and $\Gamma_\forall^0$ is trivially sound.
- If $i = k + 1$, then is follows from the induction hypothesis that $\Gamma_\exists^k$ is $(\mathcal{G}, \mathcal{A})$-consistent and $\Gamma_\forall^k$ is $(\mathcal{G}, \mathcal{A}^c)$-co-consistent. The result follows from Lemma 7 and Lemma 4 respectively.

$\square$

Consequenty, every context can be relied upon to give accurate information about the acceptance and rejection behaviour of the non-terminal symbols in the scheme.

The limit must decide the property. It remains only to show that in the limit $\langle \Gamma_\exists, \Gamma_\forall \rangle$, exactly one of the environments will assign q_0 to S. For this we require one more fact, it is that, in the limit the associated ACG is empty.

Lemma 20. *If C is the stable limit of the sequence $(C_i)_{i \in \mathbb{N}}$ then $\mathsf{ACG}(C) = \emptyset$.*

Proof. Assume for contradiction that $\mathsf{ACG}(C) \neq \emptyset$. Then, by construction, necessarily $(S, q_0) \in V_C$ and hence (S, q_0) is C-unknown whence $q_0 \notin \Gamma_\exists(S)$. We distinguish two cases. In case $\mathsf{ACG}(C)$ contains some rejecting leaf then, by Lemma 5, there is some new type introduced into $\mathsf{env}_\mathsf{R}(C)$ which contradicts the stability of C. Otherwise $\mathsf{ACG}(C)$ contains no rejecting leaves, but then, by Lemma 6, $\mathsf{RA}(C) = \mathsf{ACG}(C)$ and so $(S, q_0) \in \mathsf{RA}(C)$. But, by Lemma 15, $\mathsf{env}_\mathsf{A}(C) \vdash_\mathcal{A} S : q_0$, so $q_o \in \mathsf{env}_\mathsf{A}(C)$ contradicting stability of C. $\square$

Hence, in the limit $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$, it follows from (C1) that root vertex (S, q_0) is either C-accepting or C-rejecting. Necessarily then either $q_0 \in \Gamma_\exists(S)$ or $q_0 \in \Gamma_\forall(S)$.

Proof of the theorem. The statement of the theorem is as follows:

For any $\mathcal{G}$, $\mathcal{A}$, the algorithm terminates and:

- Answer YES implies $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A})$.
- Answer NO implies $\mathsf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A})$.

Proof. Since every type context C yields an ACG which is finite (Lemma 2), and, by definition, the corresponding rejecting and accepting regions are subgraphs of the ACG, it follows that $\mathsf{env}_\mathsf{R}(C)$ and $\mathsf{env}_\mathsf{A}(C)$ can be computed effectively. Since the sequence of type contexts must eventually stabilize to some context $\langle \Gamma_\exists, \Gamma_\forall \rangle$ (Lemma 18), it follows that the algorithm terminates. Notice that, a consequence of Lemma 20 and (ACG1) is that necessarily either $q_0 \in \Gamma_\exists(S)$ or $q_0 \in \Gamma_\forall(S)$. If the answer is yes, then necessarily $q_0 \in \Gamma_\exists(S)$ and, by Lemma 19, $\Gamma_\exists$ is consistent and so it follows from Theorem 1 that therefore $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$. Otherwise $q_0 \in \Gamma_\forall(S)$ and by Lemma 19, $\Gamma_\forall$ is $(\mathcal{G}, \mathcal{A}^c)$-co-consistent and so it follows from Theorem 1 that $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}((\mathcal{A}^\perp)^c)$. $\square$

C. Evaluation in full.

The benchmarking results presented in the main body of the paper are just a small, but representative sample of the large number of instances with which we have compared the current crop of higher-order model checkers. This data is presented in full in this appendix.

The tools we compared were PREFACE 1.2, HORSAT and HORSATT 1.0, GTRECS2 3.2, C-SHORE 1.1, TRAVMC 2.0 and TRECS 1.35. Since GTRECS2 has two modes of which one or the other is better depending on whether the instance is positive or negative, we ran just the right one on each example in order to give the best possible results. We ran PREFACE and TRAVMC on Mono 3.0.10, HORSAT, HORSATT, GTRECS2 and TRECS were compiled under OCaml 3.11.2 and C-SHORE was run on the Oracle Java SE Runtime 1.7.0_25.

The general form of the examples listed in Table 4 is as follows. The scheme $\mathtt{t}n$ is defined schematically as follows:

```
S =
  br (if (Not (L1 (L1 true))) err ok)
     (if (Not (L1 (L1 false))) err ok)
Not b = if b false true
Or b1 b2 = if b1 true b2
Bits a b c g =
  if (Or (Not a) (Or (Not b) (Not c)))
    (
      if (Not a) (Bits true b c g)
                (
                  if (Not b)
                     (Bits false true c g)
                     (Bits false false true g)
                )
    )
    g
L1 g =
  Not (
    if g (Bits false false false g) (L2 (L2 g))
  )
L2 g =
  Not (
    if g (Bits false false false g) (L3 (L3 g))
  )

  ...

L<n> g =
  Not (
    if g (Bits false false false g) g
  )
```

and the property automaton, which also serves to give an interpretation of the `if` constant, is the alternating trivial automaton with the following transition function:

$$
\begin{aligned}
\delta(q0, ok) &= \epsilon \\
\delta(q0, if) &= (1, q_t) \wedge (2, q_0) \\
\delta(q0, if) &= (1, q_f) \wedge (3, q_0) \\
\delta(qt, if) &= (1, q_t) \wedge (2, q_t) \\
\delta(qt, if) &= (1, q_f) \wedge (3, q_t) \\
\delta(qf, if) &= (1, q_t) \wedge (2, q_f) \\
\delta(qf, if) &= (1, q_f) \wedge (3, q_f) \\
\delta(qt, true) &= \mathtt{t} \\
\delta(qf, false) &= \mathtt{t} \\
\delta(q0, br) &= (1, q0) \wedge (2, q0)
\end{aligned}
$$

Benchmark	Rules	Order	Decision	Preface	TRecS
accfile	2	1	A	0.37	0.00
app_fo	7	2	A	0.38	0.02
appx_fo	6	2	A	0.39	0.02
flatten	3	1	A	0.37	0.01
gapid	11	3	A	0.50	0.02
homrep	10	4	A	0.54	0.01
isort_coerce1	13	1	A	0.38	0.01
isort_main	5	1	A	0.38	0.00
jwig-cal2_coerce2	35	1	A	0.40	0.02
jwig-cal2_main	49	1	A	0.70	0.05
jwig-cal_coerce1	35	1	A	0.40	0.02
jwig-cal_coerce2	35	1	A	0.40	0.02
jwig-cal_main	49	1	A	0.71	0.05
jwig-guess2_main	26	2	A	0.69	0.04
jwig-guess_coerce1	25	2	A	0.40	0.01
jwig-guess_main	26	2	A	0.69	0.04
mapfilt_coerce1			A	–	–
mapfilt_coerce2	34	5	A	0.41	0.01
merge	5	1	A	0.37	0.00
merge_addr	3	1	A	0.37	0.00
minicaml-k_coerce1	79	2	A	0.51	0.12
minicaml-k_coerce10	79	2	A	0.52	0.12
minicaml-k_coerce11	79	2	A	0.52	0.12
minicaml-k_coerce12	79	2	A	0.52	0.12
minicaml-k_coerce13	79	2	A	0.52	0.12
minicaml-k_coerce14	79	2	A	0.52	0.12
minicaml-k_coerce15	79	2	A	0.52	0.12
minicaml-k_coerce16	79	2	A	0.52	0.12
minicaml-k_coerce17	79	2	A	0.52	0.12
minicaml-k_coerce18	79	2	A	0.52	0.11
minicaml-k_coerce19	79	2	A	0.52	0.11
minicaml-k_coerce2	79	2	A	0.51	0.12
minicaml-k_coerce20	79	2	A	0.52	0.11
minicaml-k_coerce21	79	2	A	0.51	0.12
minicaml-k_coerce22	79	2	A	0.52	0.11
minicaml-k_coerce23	79	2	A	0.52	0.11
minicaml-k_coerce24	79	2	A	0.51	0.11
minicaml-k_coerce25	79	2	A	0.52	0.11
minicaml-k_coerce26	79	2	A	0.52	0.11
minicaml-k_coerce27	79	2	A	0.52	0.11
minicaml-k_coerce28	79	2	A	0.52	0.11
minicaml-k_coerce29	79	2	A	0.51	0.11
minicaml-k_coerce3	79	2	A	0.52	0.12
minicaml-k_coerce30	79	2	A	0.54	0.16
minicaml-k_coerce4	79	2	A	0.52	0.12
minicaml-k_coerce5	79	2	A	0.52	0.12
minicaml-k_coerce6	79	2	A	0.52	0.12
minicaml-k_coerce7	79	2	A	0.52	0.12
minicaml-k_coerce8	79	2	A	0.51	0.12
minicaml-k_coerce9	79	2	A	0.51	0.11
minicaml-k_main	25	2	A	0.41	0.04

Table 7. Benchmarks arising from HMTT verification (1/2).

Benchmark	Rules	Order	Decision	Preface	TRecS
mult	3	1	A	0.39	0.00
remove_b	4	2	A	0.37	0.00
replace	8	3	A	0.44	0.01
rev	2	1	A	0.37	0.01
split2_coerce1	28	2	A	0.42	0.03
split2_coerce2	28	2	A	0.43	0.03
split2_main	5	2	A	0.39	0.01
split_coerce1	28	2	A	0.43	0.03
split_coerce2	28	2	A	0.43	0.03
split_coerce3	28	2	A	0.42	0.03
split_coerce4	28	2	A	0.42	0.03
split_main	5	2	A	0.39	0.01
xhtmlf_drop-a	15	1	A	1.07	0.33
xhtmlf_id	2	1	A	1.10	0.58
xhtmlf_remove-meta	14	1	A	0.87	0.20
xhtmlm_drop-a	2	1	A	0.56	0.15
xhtmlm_id	2	1	A	0.52	0.07
xhtmlm_remove-meta	2	1	A	0.52	0.06
xhtmls_drop-a	2	1	A	0.42	0.02
xhtmls_id	2	1	A	0.40	0.11
xhtmls_remove-meta	2	1	A	0.40	0.01
xmarkq1_coerce1	100	2	A	0.66	0.20
xmarkq1_main	13	2	A	0.55	0.07
xmarkq2_coerce1	95	1	A	0.47	0.05
xmarkq2_coerce2	95	1	A	0.48	0.05
xmarkq2_main	10	1	A	0.42	0.02
xml_rep1	7	3	A	0.38	0.01
jwig-cal2_coerce1	35	1	R	0.43	0.01
jwig-guess2_coerce1	25	2	R	0.42	0.01
mapfilt_main	6	2	R	0.38	0.00
split2_coerce3	28	2	R	0.44	0.01
split2_coerce4	28	2	R	0.44	0.01
xhtmlf_drop-div	15	1	R	1.43	0.03
xhtmlf_remove-div	14	1	R	0.89	0.18
xhtmlm_drop-div	2	1	R	0.62	0.02
xhtmlm_remove-div	2	1	R	0.52	0.01
xhtmls_drop-div	2	1	R	0.42	0.01
xhtmls_remove-div	2	1	R	0.41	0.01
xml_rep2	7	3	R	0.39	–

Table 8. Benchmarks arising from HMTT verification (2/2).

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
cfa-life2	898	14	A	1.46	5.94	–	–	–	–	–
cfa-matrix-1	383	8	A	0.61	0.73	6.30	18.58	–	–	–
cfa-psdes	237	7	A	0.51	0.28	1.81	3.44	–	–	–

Table 9. Benchmarks arising from exact flow analysis.

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
fibstring	5	4	A	0.38	–	–	–	0.24	3.60	–
filepath	90	2	A	0.81	0.84	6.76	–	–	18.53	–
filter-nonzero-1	60	5	A	3.08	0.37	1.17	–	–	88.30	0.24
fold_fun_list	199	7	A	0.44	0.26	0.81	–	–	–	0.70
fold_right	105	5	A	0.45	–	12.27	–	–	–	–
jwig-cal_main	109	2	A	0.90	30.98	5.79	–	0.17	0.85	0.08
l	5	3	A	0.38	0.01	0.01	–	0.12	1.77	–
lock2	11	4	A	0.43	0.02	0.02	–	0.44	0.32	0.02
map-plusone-2	57	5	A	0.89	0.19	0.64	–	–	2.60	2.14
order5	11	5	A	0.41	0.10	0.02	–	–	0.31	0.01
specialize_cps_coerce1-c	83	3	A	0.63	2.06	0.93	–	–	4.45	–
xhtmlf-m-church	64	2	A	1.46	18.82	3.10	–	7.52	3.93	0.31
dna	71	2	A	18.67	0.16	0.33	–	0.06	0.45	0.04
zip	212	4	A	1.17	9.53	–	–	–	–	–
filewrong	11	4	R	0.39	0.02	0.02	–	0.04	0.32	0.00
filter-nonzero	49	5	R	0.79	0.10	0.13	–	0.27	0.37	0.01
search-e-church	98	6	R	6.09	8.98	0.86	–	–	0.52	0.01
tak	79	8	R	5.35	4.20	0.39	–	15.55	4.82	–
xhtmlf-div-2	64	2	R	2.65	21.86	2.38	–	20.22	0.68	0.30

Table 10. Benchmarks associated with HorSat(T).

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
checknz	23	2	A	0.37	0.01	0.02	0.73	0.02	0.31	0.01
filepath	90	2	A	0.81	0.85	6.70	1.75	–	18.22	–
filter-nonzero-1	60	5	A	3.09	0.39	1.18	7.33	–	88.27	0.25
last	34	2	A	0.38	0.02	0.03	0.85	0.03	0.32	0.01
map-head-filter-1	86	3	A	0.50	0.23	2.71	1.39	–	1.04	0.18
map-plusone	43	5	A	0.55	0.06	0.12	1.86	0.18	0.36	0.02
map-plusone-1	47	5	A	0.68	0.10	0.20	2.36	–	0.53	0.03
map-plusone-2	55	5	A	0.89	0.22	0.65	4.27	–	2.63	2.05
mkgroundterm	55	2	A	0.40	0.09	0.10	1.07	0.06	0.36	0.05
risers	76	2	A	0.56	0.07	0.13	1.22	0.08	0.39	0.07
safe-head	66	3	A	0.48	0.06	0.08	1.47	0.08	0.35	0.03
safe-init	76	3	A	0.58	0.15	0.19	2.20	0.40	0.43	0.04
safe-tail	70	3	A	0.53	0.06	0.10	1.58	0.12	0.36	0.03
tails	46	3	A	0.39	0.03	0.04	0.94	0.61	0.33	0.02
checkpairs	49	2	R	0.43	0.03	0.03	1.08	0.05	0.34	0.01
filter-nonzero	49	5	R	0.80	0.11	0.13	2.88	0.27	0.37	0.01
map-head-filter	62	3	R	0.50	0.12	0.12	1.44	0.20	0.36	0.01

Table 11. Benchmarks arising from PMRS verification.

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
example2-1	2	1	A	0.37	0.00	0.00	0.59	0.01	0.30	0.01
example2-3	6	1	A	0.36	0.00	0.01	0.60	0.01	0.39	0.00
exception	5	1	A	0.36	0.00	0.00	0.61	0.01	0.30	0.00
fib	5	3	A	0.49	4.96	2.09	0.93	–	58.10	0.02
file	2	1	A	0.36	0.00	0.00	0.59	0.01	0.30	0.00
fileocamlc	23	4	A	0.38	0.03	0.05	1.17	0.09	0.32	0.02
fileocamlc-old	21	4	A	0.38	0.03	0.04	1.18	0.06	0.32	0.01
fileocamlc-simple	12	4	A	0.37	0.02	0.02	0.88	0.03	0.31	0.01
fileocamlc2	22	4	A	0.38	0.03	0.04	1.15	0.06	0.32	0.01
fileorder5-2	30	5	A	0.42	0.06	0.06	1.30	–	0.45	0.17
flow	7	4	A	0.37	0.01	0.00	0.65	0.01	0.29	0.00
foo	5	2	A	0.36	0.01	0.01	0.75	0.11	0.30	0.00
guess	13	1	A	0.47	0.01	0.02	0.72	0.01	0.31	0.01
lock1	12	4	A	0.37	0.01	0.02	0.71	0.02	0.30	0.01
lock2	11	4	A	0.43	0.02	0.02	1.18	0.41	0.32	0.02
m91	25	5	A	0.88	0.88	0.09	1.43	–	0.39	0.05
merge	19	2	A	0.62	0.05	0.06	1.48	0.06	0.39	0.34
merge2	9	2	A	0.55	0.03	0.04	1.47	0.06	0.40	0.01
merge3	24	2	A	0.63	0.07	0.09	1.51	0.97	0.42	0.08
merge4	22	2	A	0.63	0.06	0.08	1.59	0.73	0.40	0.08
order4	11	5	A	0.41	0.02	0.02	1.14	–	0.31	0.01
order5	11	5	A	0.41	0.02	0.02	1.26	–	0.31	0.01
order5-2	9	5	A	0.40	0.01	0.02	0.96	–	0.34	0.02
order5-2-variant	9	5	A	0.40	0.01	0.02	0.96	–	0.35	0.02
order5-variant	12	5	A	0.51	0.03	0.02	1.21	2.60	0.31	0.02
rev	5	2	A	0.39	0.01	0.01	0.76	0.02	0.31	0.01
stress	13	1	A	0.36	0.01	0.01	0.63	0.01	0.38	0.12
twofiles	11	4	A	0.41	0.02	0.02	0.96	0.13	0.31	0.01
twofilesexn	12	4	A	0.40	0.02	0.03	0.97	0.06	0.31	0.01
ex2ib	1	0	R	0.35	0.00	0.00	0.58	0.01	0.30	0.00
example3-1	2	1	R	0.37	0.01	0.00	0.59	0.01	0.31	0.00
file2	2	1	R	0.36	0.01	0.00	0.62	0.01	0.30	0.00
filewrong	11	4	R	0.38	0.02	0.02	0.91	0.14	0.32	0.00

Table 12. Benchmarks associated with TRecS.

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
exp2-100	106	2	A	0.81	3.96	7.93	9.68	0.10	–	–
exp2-200	206	2	A	1.24	23.99	64.13	32.98	0.23	–	–
exp2-400	406	2	A	2.15	–	–	–	0.64	–	–
exp2-800	806	2	A	4.19	–	–	–	2.38	–	–
exp2-1000	1006	2	A	5.08	–	–	–	3.88	–	–
exp2-1600	1606	2	A	8.39	–	–	–	10.47	–	–
exp2-3200	3206	2	A	17.51	–	–	–	59.13	–	–
exp2-6400	6406	2	A	39.58	–	–	–	–	–	–
exp2-10000	10006	2	A	67.31	–	–	–	–	–	–
exp2-12800	12806	2	A	92.19	–	–	–	–	–	–

Table 13. Benchmarks associated with GTRecS at order 2.

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
exp3-100	107	3	A	1.60	4.70	1.25	–	0.95	–	–
exp3-200	207	3	A	2.97	36.71	7.27	–	2.82	–	–
exp3-400	407	3	A	6.19	–	53.33	–	10.28	–	–
exp3-800	807	3	A	11.36	–	–	–	39.71	–	–
exp3-1600	1607	3	A	24.14	–	–	–	–	–	–
exp3-3200	3207	3	A	56.14	–	–	–	–	–	–
exp3-6400	6407	3	A	–	–	–	–	–	–	–

Table 14. Benchmarks associated with GTRecS at order 3.

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
exp4-100	108	4	A	3.67	8.31	2.80	–	–	–	–
exp4-200	208	4	A	7.06	71.69	14.97	–	–	–	–
exp4-400	408	4	A	14.12	–	106.53	–	–	–	–
exp4-800	808	4	A	30.55	–	–	–	–	–	–
exp4-1000	1008	4	A	40.57	–	–	–	–	–	–
exp4-1600	1608	4	A	71.06	–	–	–	–	–	–
exp4-3200	3208	4	A	–	–	–	–	–	–	–
exp4-6400	6408	4	A	–	–	–	–	–	–	–

Table 15. Benchmarks associated with GTRecS at order 4.

Benchmark	Rules	Order	Decision	Preface	HorSat	HorSatT	C-SHORe	GTRecS2	TravMC	TRecS
exp5-100	109	5	A	8.67	14.02	1.81	–	–	–	–
exp5-200	209	5	A	17.80	–	10.16	–	–	–	–
exp5-400	409	5	A	37.45	–	70.53	–	–	–	–
exp5-800	809	5	A	87.17	–	–	–	–	–	–
exp5-1600	1609	5	A	–	–	–	–	–	–	–
exp5-3200	3209	5	A	–	–	–	–	–	–	–
exp5-6400	6409	5	A	–	–	–	–	–	–	–

Table 16. Benchmarks associated with GTRecS at order 5.