An Abstraction Refinement Approach to Higher-Order Model Checking

Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong

University of Oxford

Abstract. The model checking problem for higher-order recursion schemes has become an important object of study in connection with automated, higher-order program verification. However, decision procedures that have been demonstrated to scale effectively beyond a few hundred rules seem elusive. We present a new algorithm, based on abstraction refinement, which is designed to scale well to instances that consist of thousands of rules and more. In common with previous approaches, our algorithm uses an intersection type approach to the problem, but is novel in its ability to reason both using types that characterise automaton acceptance and types that characterise automaton rejection simultaneously.

1 Introduction

Higher-order model checking, or the model checking problem for trees generated by higher-order recursion schemes (HORS), holds considerable promise for research in higher-order program verification. Since HORS are simultaneously very expressive, algorithmically well behaved [1], and accurately model higher-order control flow, they are an appealing target for automatic verification procedures for functional programs. However, although there are many ingenious algorithms [2,3,4,5] that aim to solve the higher-order model checking problem on "practical" examples, none have been demonstrated to be effective on any instance whose scheme consists of over a few hundred rewrite rules. In contrast, we present an algorithm whose implementation has been shown to scale up to 10,000 rewrite rules and beyond.[1]

Our algorithm, which decides the HORS model checking problem with respect to alternating trivial tree automata, has been designed from the outset to be scalable. Since the inherent worst-case complexity of HORS model checking is extreme (hyper-exponential in the order of the scheme), 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 which is *relevant* to deciding the particular instance at hand is actually computed. To help achieve this goal, our algorithm is based on an abstraction refinement loop. In this way, initially only a relatively cheap but coarse-grained approximation to the problem is processed and, as much as possible, detail is

[1] For example, the order-2 benchmark $\mathcal{G}_{2,10000}$ of Kobayashi [2], which consists of 10006 rules, can be processed by our prototype implementation in around 40 seconds.

only added by successive iterations where the problem instance necessitates it. Moreover, it can be shown that, assuming the order and arity of the scheme are fixed, our algorithm runs in time which is bounded by a polynomial function of the size of the scheme.

In common with many of the existing algorithms for higher-order model checking, we adopt the intersection type approach which was first pioneered by Kobayashi [6]. However, unlike all other approaches, we use intersection types to characterise both property automaton acceptance *and* rejection, and our algorithm reasons about *both* kinds of judgement simultaneously. Intersection types are used as a representation of the current state of knowledge about the behaviours of the scheme. When two behaviours of the scheme are known to have different types, then they will be distinguished by the abstraction. As more types become known during successive iterations, more behaviours can potentially be separated.

The rest of the article is structured as follows. In section 2 we fix notation and preliminary definitions. In section 3 the algorithm is motivated and defined. In an appendix we present a guided run through of the algorithm on an example instance. Due to constraints we have been unable to discuss our prototype implementation, PREFACE, but a web-based interface to the tool, the opportunity to download and to view benchmarking data is available at http://mjolnir.cs.ox.ac.uk/web/preface. Proofs have been omitted but they will appear, along with a more detailed exposition, in the forthcoming doctoral dissertation of the first author.

2 Preliminaries

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

Simple sorts. The *simple sorts* over the sort 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 sort are natural numbers defined as usual. If a simple sort 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 $(s, t, u, v \in) T_\Sigma(\mathcal{F}, \mathcal{X})$, is defined by the grammar:

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

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,

$$\frac{\Delta \vdash_{\mathcal{S}} s : \kappa_1 \to \kappa_2 \qquad \Delta \vdash_{\mathcal{S}} t : \kappa_1}{\Delta \vdash_{\mathcal{S}} s\ t : \kappa_2}\ (\text{S-App})$$

$$\frac{}{\Delta,\, F : \kappa \vdash_{\mathcal{S}} F : \kappa}\ (\text{S-Fun}) \qquad \frac{\mathsf{sort}(c) = \kappa}{\Delta \vdash_{\mathcal{S}} c : \kappa}\ (\text{S-Cst}) \qquad \frac{}{\Delta,\, x : \kappa \vdash_{\mathcal{S}} x : \kappa}\ (\text{S-Var})$$

SYSTEM OF SIMPLE SORT ASSIGNMENT

contain no occurrences of function symbols by T_Σ. In case the atomic constants are said to be *sorted* we assert that there is an associated sorting function sort which maps each constant $c \in \Sigma$ to a first-order sort in $\mathbb{S}$.

Sorted terms. A sort environment Δ, is a finite, partial function from $\mathcal{X} \cup \mathcal{F}$ to $\mathbb{S}$. A *sort judgement* is an expression of the form $\Delta \vdash_{\mathcal{S}} t : \kappa$ which is provable in the system for simple sort assignment. A *well-sorted term* over $\mathcal{S}$ is just a derivable judgement $\Delta \vdash_{\mathcal{S}} t : \kappa$. Note that derivations of a given judgement are unique.

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* is finite set of first-order, sorted constants.
- The alphabet of *non-terminal symbols*, $(F, G, H \in) \mathcal{N}$, is a finite set of sorted constants, disjoint from Σ.
- The set *rewrite rules*, $\mathcal{R}$, is a function mapping each non-terminal symbol F of sort $\kappa_1 \to \cdots \to \kappa_n \to o$ to an expression $\lambda x_1 \cdots x_n.\, t$, such that:

  $$x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash_{\mathcal{S}} t : o$$

  is a provable judgement in the simple sort system.
- The *start symbol*, S, is a distinguished non-terminal symbol.

Each recursion scheme is assigned an *order* which is given by the maximum order of any of its non-terminal symbols. The *value tree* of a scheme $\mathcal{G}$, denoted $\mathsf{Tree}(\mathcal{G})$ is the (possibly infinite) term tree obtained by fair rewriting of the start symbol, *ad infinitum*.

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 there is a corresponding notion of ranked tree.

Positive Boolean formulae. Given a finite set X, the *positive Boolean formulas over* X, denoted $(\phi \in)\ \mathsf{B}^+(X)$, are defined by the following 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 $X \backslash S$ makes ϕ true.

Alternating trivial tree automata. An alternating trivial tree automaton (ATT) $\mathcal{A}$ is a tuple $\langle \Sigma, Q, \delta, q_0, F \rangle$ in which:

- The *signature*, $(a, b, c \in)\ \Sigma$, is a finite set of ranked constants.
- The *state space*, $(q \in)\ Q$, is a finite set.
- 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*, q_0, is a distinguished element of Q.
- 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 $F = \varnothing$ and we say that it has a *co-trivial* acceptance condition.

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

(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 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 $\mathcal{A}^c$, as usual for alternating tree automata. Note that the complement of an ATT with a trivial acceptance conditition is an ATT with a co-trivial acceptance condition. We define $\mathcal{A}^\perp$ as the automaton $\mathcal{A}$ augmented with additional transitions so as to accept the distinguished symbol $\perp$ from every state.

Intersection types. In what follows fix an ATT $\mathcal{A}$. 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:

$$\begin{aligned} (\textsc{Strict Types})\ \tau, \theta &::= q \mid \sigma \to \theta \\ (\textsc{Intersection Types})\ \sigma &::= \textstyle\bigwedge_{i=1}^{n} \theta_i \end{aligned}$$

in which $q \in Q$ and $n \geqslant 0$. By way of short-hand, we will typically write $\top$ for the empty intersection $\bigwedge \varnothing$, an intersection $\bigwedge_{i=1}^{2} \theta_i$ containing two elements infix as $\theta_1 \wedge \theta_2$ and an intersection $\bigwedge \{\theta\}$ of the singleton set containing θ simply as θ. By this abuse we will typically consider the strict types as a subset of the intersection types.

$$\frac{\Gamma \vdash s : \bigwedge_{i=1}^{n} \tau_i \to \tau \qquad \Gamma \vdash t : \tau_i \ (i \in [1..n])}{\Gamma \vdash s\ t : \tau} \text{ (T-App)}$$

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

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

SYSTEM OF INTERSECTION TYPE ASSIGNMENT

Intersection type environment. An *intersection type environment* Γ is a finite, partial function from $\mathcal{F} \cup \mathcal{X}$ 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:

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

Finally, will write $\Gamma{\restriction}X$ for the restriction of Γ to only those typings which whose subject lies in X.

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 for intersection type assignment above. Note, that in that system, we use the notation $S|_i$ to denote the restriction of the set of pairs S to just those pairs whose first component is exactly i. Given an intersection type environment Γ and a term t, we define the set of all strict types assignable to t under Γ by:

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

Intersection refinement types. The *intersection refinement types* over Q are those judgements $\sigma :: \kappa$, pronounced "σ refines κ" which are 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 $\sigma :: \kappa$.

Intersection type 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 $F \in \mathsf{dom}(\mathcal{N})$, there is a *possibly infinite* witness, rooted at $\Gamma \rhd F : \sigma$, and built according to the following system:

$$\frac{}{q :: o}\ (\text{K-Bas}) \qquad \frac{\sigma :: \kappa_1 \qquad \theta :: \kappa_2}{\sigma \to \theta :: \kappa_1 \to \kappa_2}\ (\text{K-Arr}) \qquad \frac{\tau_i :: \kappa\ (i \in [1..n])}{\bigwedge_{i=1}^{n} \tau_i :: \kappa}\ (\text{K-Int})$$

SYSTEM OF KIND ASSIGNMENT

$$\frac{\begin{array}{c}\mathcal{R}(F) = \lambda x_1 \ldots x_n.\, t \\ \Gamma,\, x_1 : \sigma_1, \ldots, x_1 : \sigma_n \vdash t : q \\ \Gamma \rhd G : \sigma\ (\text{for each } G : \sigma \in \Gamma)\end{array}}{\Gamma \rhd F : \sigma_1 \to \cdots \to \sigma_n \to q} \qquad \frac{\begin{array}{c}\Gamma' \subseteq \Gamma \\ \Gamma' \rhd F : \tau_i\ (\text{for each } i \in [1..n])\end{array}}{\Gamma \rhd F : \bigwedge_{i=1}^{n} \tau_i}$$

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 [7].

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

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

3 An abstraction refinement algorithm

In the following we present an algorithm that attempts to prove both of $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ and $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}((\mathcal{A}^{\perp})^c)$ simultaneously using the characterisation given in Theorem 1. To this end, the algorithm constructs an eventually stable sequence of pairs of intersection type environments, called contexts, of the form $\langle \Gamma_\exists, \Gamma_\forall \rangle$ with $\Gamma_\exists$ $(\mathcal{G},\mathcal{A})$-consistent and $\Gamma_\forall$ $(\mathcal{G},\mathcal{A}^c)$-co-consistent. In the limit, exactly one of the environments is guaranteed to contain the typing $S : q_0$, which will decide the model checking problem one way or the other. In iteration $i+1$, an abstraction of the behaviour of the scheme is constructed which is a function of the type information contained in context $\langle \Gamma^i_\exists, \Gamma^i_\forall \rangle$. This type information determines when the abstraction should or should not distinguish different instances calls to the same non-terminal symbol. Once the abstraction is constructed, it is divided up into three distinct regions. The *accepting* region contains those behaviours which result in trees that are accepted by the property automaton. The *rejecting* region contains those behaviours which result in trees that are rejected by the property automaton (i.e. accepted by the complement of the property automaton). Finally, the *unknown* region contains the remaining behaviours of the abstraction, about which nothing is yet certain. Now, from these regions can be extracted new type information. Naturally, the types extracted from the accepting region contribute to $\Gamma^{i+1}_\exists$ and the types extracted from the

rejecting region contribute to $\Gamma_\forall^{i+1}$. It is a fact that, on every non-final iteration i, some genuinely new, rejection type information is contributed to $\Gamma_\forall^{i+1}$, though the same is not true of $\Gamma_\exists^{i+1}$ and acceptance type information. If either of the successor environments contain the typing $S : q_0$ then the algorithm terminates or else otherwise proceeds onto the next iteration. Since new type information is contributed to $\Gamma_\forall$ on every iteration, and there is only a finite amount of type information which can be contributed in total, this process cannot continue forever.

3.1 Construction of abstraction

Typed variables. The main mechanism for abstraction will be a set of typed variables. By means of an additional set of variable-term bindings (to be described in the sequel) each variable exists as an abstraction for the set of terms that can be obtained from it by repeated substitution according to the bindings. Each variable has three pieces of associated type information: an acceptance type, a rejection type and a sort. An important part of the algorithm lies 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 sort as the variable. The association of a variable with its type information is made precise in the following.

Definition 1. *Let* var *be a bijection, mapping the finite set of triples of the form* $(\sigma_\mathsf{A}, \sigma_\mathsf{R}, \kappa)$ *consisting of kinded types* $\sigma_\mathsf{A} :: \kappa$ *and* $\sigma_\mathsf{R} :: \kappa$ *to a finite set of term variables* $\mathcal{Y} \subseteq \mathcal{X}$*. 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. Hence, 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. that 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 all of 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. The abstraction itself is a finite representation of the possibly infinite *configuration graph*, as defined in Kobayashi and Ong [7]. This graph can be viewed as the state space of a kind of product construction between

the reduction graph of the scheme and the transition graph of the property automaton. Hence, in the *concrete* configuration graph, one thinks of configurations as 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. The configurations (t, q) should be viewed as a kind of primitive assertion, that term t generates a tree which is accepted from state q. The abstraction is always rooted at (S, q_0) and the exploration of the state space from this point corresponds to computing the necessary requirements, phrased in terms of configurations, that must be satisfied in order that S generate a tree accepted from state q_0. 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 finite 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 3.1.*

We consider motivation of each of the rules of the inductive definition in turn. First, the rule (ACG1) 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 such ACG:

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 (ACG2) 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, 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 $y_i = \mathsf{var}(\bigwedge \mathbb{T}(\Gamma_\exists)(s_i), \bigwedge \mathbb{T}(\Gamma_\forall)(s_i), \mathsf{K}(s_i))$. 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* $y \mapsto t \in B_C$*,* $\mathbb{T}(\Gamma_\exists)(y) = \mathbb{T}(\Gamma_\exists)(t)$ *and* $\mathbb{T}(\Gamma_\forall)(y) = \mathbb{T}(\Gamma_\forall)(t)$*.*

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 among the set of terms abstracted by typed variable y_i. Rule (ACG3) 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 $\delta(a, q)$ 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 (ACG4) simply decomposes set vertices into their constituent configurations. Therefore, the children of a set vertex should be thought of conjunctively. Finally, rule (ACG5) 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

(ACG1) Whenever all the following are true:
(i) (S, q_0) is C-unknown
then all the following are also true:
– $(S, q_0) \in V_C$

(ACG2) Whenever all the following are true:
(i) $(F\ s_1\ \cdots\ s_n, q) \in V_C$
(ii) $\mathcal{R}(F) = \lambda x_1\ \cdots\ x_n.\, t$ and $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_n \to o$
(iii) $(t[y_1/x_1, \cdots, y_n/x_n], q)$ is C-unknown
(iv) for each $i \in [1..n]$, $y_i = \mathsf{var}(\bigwedge \mathbb{T}(\Gamma_\exists)(s_i), \bigwedge \mathbb{T}(\Gamma_\forall)(s_i), \kappa_i)$
then all the following are also true:
– $(t[y_1/x_1, \cdots, y_n/x_n], q) \in V_C$
– $(F\ s_1\ \cdots\ s_n, q) \longrightarrow (t[y_1/x_1, \cdots, y_n/x_n], q) \in E_C$
– for each $i \in [1..n]$: $y_i \mapsto s_i \in B_C$

(ACG3) 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:
– $\{(s_i, q') \mid (i, q') \in S\} \in V_C$
– $(a\ s_1\ \cdots\ s_n, q) \longrightarrow \{(s_i, q') \mid q' \in \sigma_i \wedge i \in [1..n]\} \in E_C$.

(ACG4) 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:
– $(s_i, q_i) \in V_C$
– $\{(s_1, q_1), \ldots, (s_n, q_n)\} \longrightarrow (s_i, q_i) \in E_C$

(ACG5) 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:
– $(t\ s_1\ \cdots\ s_n, q) \in V_C$
– $(y\ s_1\ \cdots\ s_n, q) \longrightarrow (t\ s_1\ \cdots\ s_n, q) \in E_C$

Fig. 1. Abstract configuration graph construction.

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 (ACG2) and the limited substitution (only in head position) in (ACG5), it follows that $\mathsf{ACG}(C)$ is necessarily a finite construction. In fact, it is possible to 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 an ACG of a 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 (ACG2), 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: (ACG2) does not apply because the third 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 $\mathcal{R}(F) = \lambda x_1 \ldots x_n.\, t$, for each $i \in [1..n]$, there is a typed variable y_i such that $\mathsf{A}(y_i) = \bigwedge \mathbb{T}(\Gamma_\exists)(s_i)$ and $\Gamma_\exists \vdash_{\mathcal{A}} t[y_1/x_1, \ldots, y_n/x_n] : q$.*

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

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

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

3.2 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 $(F\ s_1 \cdots s_n, q)$. By definition, the contraction of this configuration using (ACG2) would yield a configuration $(t[y_1/x_1, \ldots, y_n/x_n], q)$ which is already C-rejecting. In other words, the tree generated by *any* term of the form $t[t_1/y_1, \ldots, t_n/y_n]$ such that $\mathbb{T}(\Gamma_\forall)(y_i) \subseteq \mathbb{T}(\Gamma_\forall)(t_i)$ for each i is sure to be rejected from state q. This follows, assuming that $\Gamma_\forall$ is co-consistent, since in such a case $\Gamma_\forall \vdash_{\mathcal{A}^c} t[t_1/y_1, \ldots, t_n/y_n] : q$. Recalling Proposition 1, one such term is the first component of the rejecting leaf we started with $(F\, s_1 \cdots s_n, q)$. Hence, because we know that the contractum

of this redex generates a tree that is rejected from state q_0, necessarily the redex itself generates a tree that is rejected from state q_0. 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 by:*

(R1) If c is a rejecting leaf then $c \in \mathsf{RR}(C)$.
(R2) If $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in V_C$ and there exists $j \in [1..n]$ such that $(s_j, q_j) \in \mathsf{RR}(C)$ then $\{(s_1, q_1), \ldots, (s_n, q_n)\} \in \mathsf{RR}(C)$.
(R3) If $(F\ s_1 \cdots s_n, q) \longrightarrow (t, q) \in E_C$ and $(t, q) \in \mathsf{RR}(C)$ then $(F\ s_1 \cdots s_n, q) \in \mathsf{RR}(C)$.
(R4) If $(a\, s_1 \cdots s_n, q) \in V_C$, let W be all v such that $(a\, s_1 \cdots s_n, q) \longrightarrow v \in E_C$. If, for all $v \in W$, $v \in \mathsf{RR}(C)$, 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. An absence of rejecting vertices is an absence of counterexamples in the abstraction.

Lemma 4. *Let C be a type context and $\mathsf{ACG}(C)$ have no rejecting leaves. Then:*

$$\mathsf{ACG}(C) = \mathsf{RA}(C)$$

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

Rejection type extraction. The vertices in the rejecting region are those configurations that we have identified, as a result of constructing $\mathsf{ACG}(C)$, that *should* be classified by our type information as rejecting, but *are not* – each is necessarily C-unknown, since it exists in 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 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.*

(i) If the proof is by (R1) then v is a configuration of the form $(F\, s_1 \cdots s_n, q)$, and we set:

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

(ii) If the proof is by (R2) then 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). We set $\mathcal{M}(T) = \mathcal{M}(T')$.

(iii) *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 $\mathcal{M}(v)$ the environment:*

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

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

(v) *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 $(t\, s_1 \cdots t_n, q)$. 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:*

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

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

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

So the types are extracted in an inductive fashion, starting from the leaves of each witness and working backwards. It is this well-foundedness that ensures that the types that are extracted are all "correct":

Lemma 5. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context. If $\Gamma_\forall$ is $(\mathcal{G},\mathcal{A}^c)$-co-consistent then $\Gamma_\forall \uplus \mathsf{env}_\mathsf{R}(C)$ is $(\mathcal{G},\mathcal{A}^c)$-co-consistent.*

Furthermore, whenever the rejecting region is non-empty, then genuinely new type information will be extracted. Taken together with Lemma 4, the following result is the key measure of progress in the algorithm.

Lemma 6. *Let $C = \langle \Gamma_\exists, \Gamma_\forall \rangle$ be a type context and $\mathsf{ACG}(C)$ have some rejecting leaf. Then:*

$$\mathsf{env}_\mathsf{R}(C) \backslash \Gamma_\forall \neq \varnothing$$

3.3 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* $(F\ s_1 \cdots s_n, q) \in \mathsf{RA}(C)$ *and* $(F\ s_1 \cdots s_n, q) \longrightarrow (t, q) \in V_C$, *then* $(t, q) \in \mathsf{RA}(C)$.
(A2) *If* $(a\ s_1 \cdots s_n, q) \in \mathsf{RA}(C)$, *then there is some satisfying assignment* S *to* $\delta(q, a)$ *such that* $\{(s_i, q') \mid (i, q') \in S\} \in \mathsf{RA}(C)$.
(A3) *If* $\{(s_1, q_1), \ldots, (s_m, q_m)\} \in \mathsf{RA}(C)$ *then,* $\forall i \in [1..m]$, $(s_m, q_m) \in \mathsf{RA}(C)$.
(A4) *If* $(y\, s_1 \cdots s_n, q) \in \mathsf{RA}(C)$ *then, for all* $y \mapsto t \in B_C$, $(t\, s_1 \cdots s_n, q) \in \mathsf{RA}(C)$.

However, unlike the case for accepting 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.

Acceptance type extraction. To extract new type information from the accepting region we follow the approach of Kobayashi in [2] (a simplification of work in Kobayashi and Ong [7]), in which types are assigned to prefixes of configurations recursively based on the sort 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 sort of* s.

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

$$\mathsf{extr}(c, s) \;=\; \bigwedge \mathbb{T}(\Gamma_\exists)(t_1) \wedge \bigwedge_{c' \in W} \mathsf{extr}(c', t_1) \;\rightarrow\; \mathsf{extr}(c, s\ t_1)$$

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

3.4 Fixed point construction

Abstraction refinement. Finally, we are in a position to describe the overall iterative 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 $(\mathcal{G},\mathcal{A})$-consistent and $(\mathcal{G},\mathcal{A}^c)$-co-consistent 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 4 and Lemma 6, 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 scheme, it follows that the algorithm as a whole is polynomial in the size of the scheme, assuming its order and arity are taken to be fixed.

References

1. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, Los Alamitos, CA, USA, IEEE Computer Society (2006) 81–90
2. Kobayashi, N.: Model-checking higher-order functions. In: PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2009) 25–36
3. Kobayashi, N.: A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In: Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, Springer (2011) 260–274
4. Neatherway, R.P., Ramsay, S.J., Ong, C.H.L.: A traversal-based algorithm for higher-order model checking. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012, ACM (2012) 353–364

5. Broadbent, C.H., Carayol, A., Hague, M., Serre, O.: A saturation method for collapsible pushdown systems. In: Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II, Springer (2012) 165–176
6. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2009) 416–428
7. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA, IEEE Computer Society (2009) 179–188
8. Might, M., Shivers, O.: Exploiting reachability and cardinality in higher-order flow analysis. Journal of Functional Programming **18**(5-6) (2008) 821–864

A A guided run of the algorithm

By way of an example, we consider an instance of the HORS model checking problem which is due to Kobayashi in [2] (though it is itself an encoding of a problem considered by Might and Shivers in [8]).

A.1 Model checking instance

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

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

over the terminal symbols $\mathtt{flow} : o \to o$ and $\mathtt{end} : o$. The property is specified by the following deterministic trivial automaton $\mathcal{A}$ which consists of a single state q_0 and a single transition $\delta(q_0, \mathtt{end}) = \mathtt{t}$. The idea is that the scheme will be rejected just if a use of Lam flows to the result, i.e. if $\mathtt{flow}$ 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 as follows:

$$\begin{aligned} &id67 : (\top, \top, (o \to o) \to ((o \to o) \to o) \to o) \\ &k69 : (\top, \top, (o \to o) \to o) \\ &x68 : (\top, \top, o \to o) \\ &x76 : (q_0 \to q_0, \top, o \to o) \\ &x80 : (\top \to q_0, \top, o \to o) \\ &id84 : ((q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0, \top, (o \to o) \to ((o \to o) \to o) \to o) \end{aligned}$$

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 A.1. If you have the luxury of viewing this document in colour, you will notice that variable-headed configurations are outlined in blue and vertices in the rejecting region and accepting region are coloured red and green respectively. The binding set for the graph is as shown below. Note that,

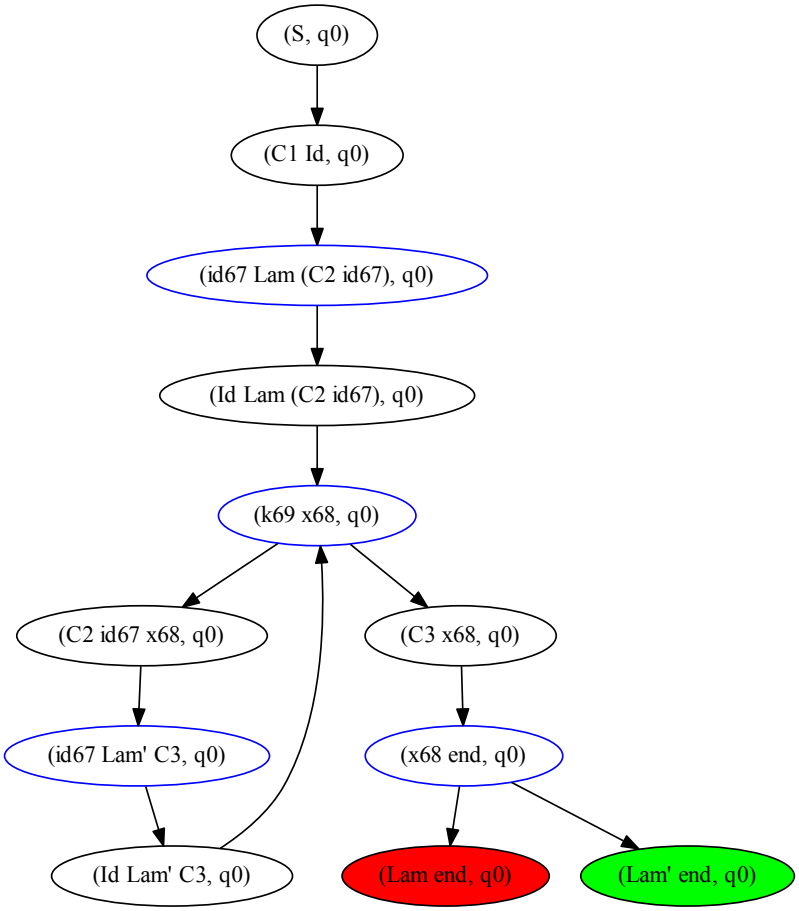

Fig. 2. Abstract configuration graph for iteration 1.

as a small optimisation to aid readability, we will never add bindings of the form $y \mapsto y$ to the set during graph construction.

$$
\begin{aligned}
x68 &\mapsto \mathit{Lam}' \\
x68 &\mapsto \mathit{Lam} \\
k69 &\mapsto \mathit{C3} \\
k69 &\mapsto \mathit{C2} \\
\mathit{id}67 &\mapsto \mathit{Id}
\end{aligned}
$$

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 calls to the non-terminal Id, once due to the contraction of $\mathit{Id}\,\mathit{Lam}\,(\mathit{C2}\ \mathit{id67})$ and again due to the contraction of $\mathit{Id}\,\mathit{Lam}'\,\mathit{C3}$. Consequently, the typed variable $x68$, 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 $\mathit{Lam}\,\mathit{end}$ 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 $(\texttt{flow}\ y, q_0)$, is C-rejecting for any C, in essence because $\texttt{flow}\ s$ is rejected from $\mathcal{A}$ 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 $\mathit{Lam} : \top \to q_0$, representing the fact that when applied to an argument of no discernable 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 rejecting region is, in this case, nevertheless restricted the single accepting leaf $(\mathit{Lam}'\ \texttt{end}, q_0)$. Consequently, the type inferred from this leaf is $\mathit{Lam}' : q_0 \to q_0$, 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 $C_1 = \langle \mathsf{A} \uplus \{\mathit{Lam}' : q_0 \to q_0\}, \mathsf{R} \uplus \{\mathit{Lam} : \top \to q_0\}\rangle$, so that Lam and Lam', which were confused on this iteration, now have different types and hence will be distinguished by the next.

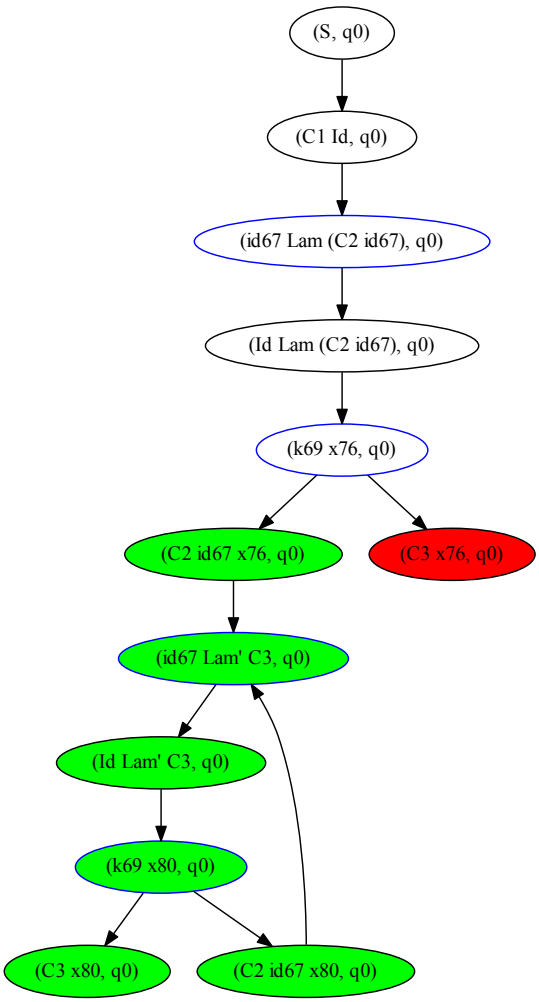

Fig. 3. Abstract configuration graph for iteration 2.

A.3 Iteration 2

The graph for the second iteration $\mathsf{ACG}(C_1)$ is shown in Figure A.2. The binding set for the graph is as shown below.

$$\begin{aligned} x80 &\mapsto \mathit{Lam}' \\ x76 &\mapsto \mathit{Lam} \\ k69 &\mapsto \mathit{C3} \\ k69 &\mapsto \mathit{C2}\ \mathit{id67} \\ \mathit{id}67 &\mapsto \mathit{Id} \end{aligned}$$

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 id67*, 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 $(\mathit{C3}\ \mathit{x76},\ q_0)$ appears in the graph. This configuration is a rejecting leaf since, if contracted, it would yield a configuration (*Lam* `end`) which is C_1-rejecting. From this leaf, the new rejection type $\mathit{C3} : (\top \to q_0) \to q_0$ 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:

$$\begin{aligned} &\mathit{C3} : (q_0 \to q_0) \to q_0 \\ &\mathit{C2} : ((q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0) \to (q_0 \to q_0) \to q_0 \\ &\mathit{C2} : ((q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0) \to \top \to q_0 \\ &\mathit{Id}\ \ : (q_0 \to q_0) \to ((q_0 \to q_0) \to q_0) \to q_0 \end{aligned}$$

This ensures that in the following iteration, the term $C3$ and the term *C2 id67* 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 A.3. In this case, the associated binding set is simply:

$$\mathit{id}84 \mapsto \mathit{Id}$$

Here, the configuration $(\mathit{Id}\ \mathit{Lam}\ (\mathit{C2}\ \mathit{id84}),\ q_0)$ is an accepting leaf since, if it were to be contracted it would yeild the configuration (*C2 id84 Lam*) which is already known to be C_2-accepting. Hence the construction of the graph is halted prematurely. Since there are 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.

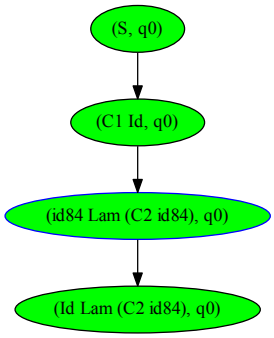

Fig. 4. Abstract configuration graph for iteration 3.