A Traversal-based Algorithm for Higher-Order Model Checking

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

Steven J. Ramsay
University of Oxford
steven.ramsay@cs.ox.ac.uk

Abstract

Higher-order model checking—the model checking of trees generated by higher-order recursion schemes (HORS)—is a natural generalisation of finite-state and pushdown model checking. Recent work has shown that it can serve as a basis for software model checking for functional languages such as ML and Haskell. In this paper, we introduce *higher-order recursion schemes with cases* (HORSC), which extend HORS with a definition-by-cases construct (to express program branching based on data) and non-determinism (to express abstractions of behaviours). This paper is a study of the *universal HORSC model checking problem for deterministic trivial automata*: does the automaton accept every tree in the tree language generated by the given HORSC? We first characterise the model checking problem by an intersection type system extended with a carefully restricted form of union types. We then present an algorithm for deciding the model checking problem, which is based on the notion of *traversals* induced by the fully abstract game semantics of these schemes, but presented as a goal-directed construction of derivations in the intersection and union type system. We view HORSC model checking as a suitable backend engine for an approach to verifying functional programs. We have implemented the algorithm in a tool called TRAVMC, and demonstrated its effectiveness on a test suite of programs, including abstract models of functional programs obtained via an abstraction-refinement procedure from pattern-matching recursion schemes.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Verification

Keywords Model-checking, Higher-order Programs

1. Introduction

Over the past decade, model checking and its allied methods have been applied to program verification with great effect. For first-order, imperative programs, highly optimised finite-state and pushdown model checkers (such as SLAM [2] and BLAST [3]) have been successfully applied to bug-finding, property checking and test case generation. Building on theoretical results on the model checking of *higher-order recursion schemes* (HORS) [7, 16], Kobayashi [9] has sparked a growing interest in the development of an analogous model checking framework for higher-order, functional programs.

A HORS is a kind of higher-order grammar, which can be viewed as a mechanism for generating a possibly-infinite, ranked tree. HORS model checking is concerned with the problem of deciding whether the tree generated by a given HORS satisfies a given property and, when the property is expressed by a formula of the modal mu-calculus (equivalently, an alternating parity tree automaton), then the problem is known to be decidable [16]. Since they can equally well be viewed as a closed, ground-type term of the simply-typed lambda calculus with recursion and uninterpreted first-order constants, HORS are a natural home for models of higher-order computation. Indeed, HORS model checking is a smooth generalisation of finite-state and pushdown model checking (finite-state programs and pushdown systems/Boolean programs are captured by order-0 and order-1 HORS respectively).

HORS model checking is, inherently, an extremely complex problem. Ong [16] has shown that the modal mu-calculus model checking problem for order-n recursion schemes is n-EXPTIME (i.e. tower of exponentials of height n) complete. Even for the purposes of safety verification (model checking against properties expressible as *deterministic trivial tree automata* (DTT)), the problem is $(n-1)$-EXPTIME complete [12], which is still formidably complex. Hence, the feasibility of HORS model checking as a verification technology is predicated upon the ability to design decision procedures that hit the worst-case complexity only in pathological cases.

That such algorithms are possible was demonstrated by Kobayashi's *hybrid algorithm*, presented in [8], which solves the safety verification problem. In an attempt to avoid the hyper-exponential bottleneck, the algorithm closely analyses the actual behaviour of the HORS as it is evaluated, generating the ranked tree. The hybrid algorithm builds a graph to record the trace of this computational behaviour and from the graph derives guesses at proofs which witness the satisfaction of the property. The algorithm is implemented in the TRECS tool [10], which has been shown to perform remarkably well in a variety of applications.

However, whilst HORS allow for the expression of higher-order behaviour very naturally, they lack two important features which, we believe, are highly desirable in a convenient abstract model of *functional programs*. The first is a case analysis construct, with which one can express program branching based on data; the second is non-determinism[1], with which one can express abstractions of behaviour. In this paper, we present a class of structures called *higher-order recursion schemes with cases* (HORSC) which extend HORS in both these directions, allowing grammar rules to be non-deterministic and incorporating a finitary case analysis construct.

Example 1. The *Risers* program from Mitchell and Runciman [15] provides an interesting example of a program with partial pattern matching that cannot crash:

```
risers [] = []
risers [x] = [[x]]
risers (x : y : etc) = if x ≤ y then (x : s) : ss else [x] : (s : ss)
    where (s : ss) = risers (y : etc)
```

A natural abstraction that might be selected by an automated approach is to the finite domain $\{\mathsf{Nil}, \mathsf{Cons}_1, \mathsf{Cons}_2\}$ (for lists of length 0, 1 or more and 2 or more respectively). Using non-deterministic choice for the **if** statement and a case construct operating on the finite domain yields:

```
risers xs → case(xs, Nil, Cons₁, ifthenelse)
ifthenelse → cons (destruct (risers Cons₁))
ifthenelse → cons (cons (destruct (risers Cons₁)))
destruct xs → case(xs, error, Nil, Cons₁)
destruct xs → case(xs, error, Nil, Cons₂)
cons xs → case(xs, Cons₁, Cons₂, Cons₂)
```

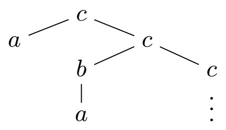
The pattern match error is preserved – it occurs in the case where an empty list is destructed under the assumption that it has length at least one. Furthermore, the safety of the original program has been preserved in the abstraction to HORSC-like syntax.

Our central contribution is an algorithm to decide the model checking problem of HORSC against DTT. Our algorithm is inspired by the game-semantic analysis (in particular, the notion of *traversals*) behind the original decidability proof of Ong [16] for the model checking problem for HORS. The technical machinery of game semantics is not required for this algorithm, but here we offer a brief overview for the interested reader. Game semantics [6] is a way of giving meanings to programs by viewing computation as a game between Proponent (whose point of view is the program) and Opponent (whose point of view is the program context). The type of a program $M : \theta$ is interpreted as an arena $[\![\theta]\!]$, and the program is interpreted as a Proponent strategy, $[\![M]\!]$, for playing in the arena $[\![\theta]\!]$. Inspired by the success of the hybrid algorithm, we aim to search for proofs in a way which is guided by an analysis of the behaviour of the HORS but, rather than evaluating the HORS and analysing its traces, we analyse the *traversal* induced by its game semantics.

The standard method to evaluate such a λ-term is by β-reduction but, because of the nature of substitution, β-reduction *deforms* the syntactic structure of the term and information about the computation that took place can be lost in the reduct. The game semantics of the simply-typed λ-calculus gives rise to a method of evaluating a term M by *traversing* its computation tree, $\lambda(M)$, which is a slightly souped-up version of its abstract syntax tree. In contrast, evaluation by traversal leaves the structure of the term in question intact.

Example 2 (Traversals over recursion scheme $\mathcal{G}_1$)**.** Let $a : o$, $b : o \to o$ and $c : o \to o \to o$ be terminal symbols. Consider the recursion scheme $\mathcal{G}_1$ given by the following recursive definition of functions, $S : o$ and $F : o \to o$, viewed as rewrite rules:

$$\begin{array}{rcl} S & \to & F\,a \\ F\,x & \to & c\,x\,(F\,(b\,x)) \end{array}$$



Unfolding from S, we have

$$S \to F\,a \to c\,a\,(F\,(b\,a)) \to c\,a\,(c\,(b\,a)\,(F\,(b\,(b\,a)))) \to \cdots$$

thus generating the infinite term $c\;a\;(c\;(b\;a)\;(c\;(b\;(b\;a)))\;(\cdots))$. Define the tree generated by $\mathcal{G}_1$, $[\![\mathcal{G}_1]\!]$, to be the abstract syntax tree of the infinite term, as shown above (on the right).

The computation tree $\lambda(\mathcal{G}_1)$ is the underlying tree in Figure 1 whose nodes are labelled by symbols λ, λx, @, a, b and c. We will not give the rules that define the traversals over a computation tree. Instead, we illustrate how traversals compute the paths in $[\![\mathcal{G}_1]\!]$ that are labelled $c \cdot a$ and $c \cdot c \cdot b \cdot a$ respectively. The path $c \cdot a$ in $[\![\mathcal{G}_1]\!]$ corresponding to the traversal over $\lambda(\mathcal{G}_1)$ from the root to (1), jumping to the segment that starts from (1), namely, $\lambda \cdot @ \cdot \lambda x \cdot c \cdot \lambda \cdot x \cdot \lambda \cdot a$. The path $c \cdot c \cdot b \cdot a$ in $[\![\mathcal{G}_1]\!]$ corresponding to the traversal over $\lambda(\mathcal{G}_1)$ from the root to (2), jumping to the segment that starts from (2) and which ends at (3), and jumping to the segment that starts from (3); namely, $\lambda \cdot @ \cdot \lambda x \cdot c \cdot \lambda \cdot @ \cdot \lambda x \cdot c \cdot \lambda \cdot x \cdot \lambda \cdot b \cdot \lambda \cdot x \cdot \lambda \cdot a$. Let $\Sigma = \{\, a, b, c \,\}$. Note that the Σ-projection of the two traversals are the two paths in $[\![\mathcal{G}_1]\!]$.

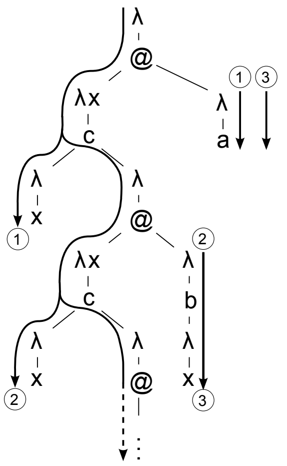


Figure 1. Traversals in $\lambda(\mathcal{G}_1)$ (Example 2)

An insight of Kobayashi, which has been instrumental in the design of practical model checking algorithms, is that the HORS model checking problem can be characterised as a problem of type inference in a certain intersection type system. By this characterisation, searching for a proof that a given HORS satisfies a given property is reduced to searching for a typing for the given HORS in the type system induced by the given property. We show that the HORSC model checking problem also has an elegant, type-theoretic characterisation, but that the combination of higher-order functions, case-analysis and non-determinism lead one to consider a system of intersection *and union* types. Since we want to minimize any increase to the size of the search space of typings (which, by the characterisation, act as potential witnesses to property satisfaction), we have carefully constructed a type system in which union types can occur only in a restricted fashion. In particular, unions are only ever allowed over a subset of the ground types.

[1] In fact, there is no requirement for HORS to be purely deterministic by definition, but the type theory on which the model checking tools are built has only been properly developed for deterministic HORS.

In light of this type-theoretic characterisation, we present our model checking algorithm as a goal-directed construction of a typing derivation. (For reasons of exposition, we suppress the game-semantic origin and interpretation of the algorithm, but present a formal account of the correspondence in the appendices) The ultimate aim is to show that the start symbol S of the HORS is typable by a type representing the initial state of the property automaton q_0, so the initial goal is to find a typing environment Γ such that $\Gamma \vdash S : q_0$. In our type system, we are allowed to take for Γ the environment that consists of the single typing $S : q_0$, but only if we are able to show that the definition of S (by a production rule in the HORS) respects this typing. Hence, following the type system, the algorithm is obliged to spawn a subgoal (itself a typing judgement) according to the definition of S. In general, to solve a goal the algorithm simply attempts to construct a typing derivation according to the rules of the type system, but, where this construction involves making additional assumptions (such as in the typing derivation for $S : q_0$ as above) an obligation is incurred to justify these assumptions. Since discharging such obligations can sometimes require "jumping back" to refine previously completed typing derivations, the construction is not a straightforward bottom-up exercise in tree building. In fact, the pattern of construction (precisely, the sequence of calls to the *Close*−-procedures of Algorithm 1) follows exactly the game-semantic traversals over the corresponding computation tree.

Based on an empirical evaluation, the traversal algorithm is several orders of magnitude faster than Kobayashi's linear-time algorithm GTRecS [11]. Although it does not quite match Kobayashi's hybrid algorithm (which is generally up to an order-of-magnitude faster), the traversal algorithm is still remarkably fast and practical, in view of the worst-case asymptotic complexity of the problem, which is $(n-1)$-EXPTIME complete [12].

Outline The rest of the paper is organised as follows. We introduce higher-order recursion schemes with cases in Section 2, and recall some standard definitions from the literature. In Section 3 we describe an intersection and union type system used to characterise the model checking problem for HORSC, before going on to describe a type inference algorithm in Section 4. Section 5 presents the empirical evaluation of our methods and algorithms, with a discussion of related work in Section 6, followed by our conclusion and further directions in Section 7.

2. Higher-Order Recursion Schemes with Cases

We introduce a new class of structures, *higher-order recursion schemes with cases* and their model checking problem, and agree on familiar definitions of Σ-labelled trees and deterministic trivial tree automata.

Recursion Schemes with Cases

Let D be a set of *directions* (e.g. $D = \{1, 2, \cdots, m\}$). A *D-tree* (or simply *tree*) is a prefix-closed subset T of D^*. Let Σ be a ranked alphabet. A *Σ-labelled tree* is a function $t : dom(t) \to \Sigma$ such that $dom(t)$ is a tree. We refer to elements of $dom(t)$ as *nodes* of t.

In what follows, we refer to simple types as *kinds* (reserving the word *type* for intersection types, to be introduced shortly) and define the set of kinds by $\kappa ::= d \mid o \mid \kappa \to \kappa$ where o is the kind of Σ-labelled trees, and d is the kind of a finite domain for definition by cases. As usual, the *order* of a kind is the maximum nesting of an arrow on the left, that is: $ord(o) = 0$ and $ord(\kappa_1 \to \kappa_2) = \max(ord(\kappa_1)+1, ord(\kappa_2))$. We use β and β_i to range over ground (i.e. order-0) kinds.

Definition 1. A *(non-deterministic) higher-order recursion scheme with cases* (HORSC) is a quadruple $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where

(i) Σ is an alphabet of well-kinded terminal symbols (ranged over by f, g, a, b, etc.) with kinds drawn from those of order at most one. Further Σ contains a distinguished subset of d-kinded symbols, $\mathcal{B} = \{ \mathsf{b}_1, \ldots, \mathsf{b}_n \}$; and if $f \in (\Sigma \setminus \mathcal{B})$ then f has return kind o i.e. $f :: \beta_1 \to \cdots \to \beta_m \to o$ where $m \geq 0$.

(ii) $\mathcal{N}$ is an alphabet of kinded non-terminal symbols (ranged over by F, G etc.).

(iii) $\mathcal{R}$ is a set of rewrite rules of the form $F x_1 \cdots x_m \to e$ where $F :: \kappa_1 \to \cdots \to \kappa_m \to \beta$ with $\beta \in \{ d, o \}$, each $x_i :: \kappa_i$ is drawn from a countably infinite set of variables and $e :: \beta$ is a (well-kinded) applicative *term* generated from the following grammar

$$e \ ::= \ x \mid f \mid F \mid e_1\, e_2 \mid \mathsf{case}(e, e_1, \ldots, e_n)$$

where n is the cardinality of $\mathcal{B}$, $x \in \{ x_1, \ldots, x_m \}$, $f \in \Sigma$ and $F \in \mathcal{N}$. When a term contains no occurrence of a variable x, we say that it is *closed*. The kinding rule for the case construct is: if $s :: d$ and each $t_i :: \beta$ (base kind) then $\mathsf{case}(s, t_1, \ldots, t_n) :: \beta$; the other kinding rules are standard. We consider $\mathcal{R}$ to be a function defined by:

$$\mathcal{R}(F) := \{ \lambda x_1 \cdots x_m.e \mid F\, x_1 \cdots x_m \to e \in \mathcal{R} \}$$

When $\mathcal{G}$ is deterministic, that is, for each $F \in \mathcal{N}$, $\mathcal{R}(F)$ is a singleton, we abuse notation and identify $\mathcal{R}(F)$ with its only member.

(iv) $S \in \mathcal{N}$ is a distinguished 'start' symbol of kind o, and $\mathcal{R}(S)$ is a singleton set. By abuse of notation we write $S \to \mathcal{R}(S)$ for the unique rule for S.

The (call-by-name) reduction relation of the HORSC $\mathcal{G}$, written $\to_{\mathcal{G}}$ (or simply $\to$ whenever $\mathcal{G}$ is understood), is a binary relation over closed, ground-kinded terms, defined by induction over the following rules.

$$\frac{\lambda x_1 \ldots x_m.t \in \mathcal{R}(F)}{F\, s_1 \ldots s_m \to t[\overline{s}/\overline{x}]} \qquad \frac{1 \leq i \leq n}{\mathsf{case}(\mathsf{b}_i, s_1, \ldots, s_n) \to s_i}$$

$$\frac{s \to s'}{C[s] \to C[s']}$$

where the (one-holed) contexts are defined as follows:

$$\begin{array}{rcl} C & ::= & [\,] \mid C\, s \mid s\, C \mid \mathsf{case}(C, t_1, \ldots, t_n) \\ & \mid & \mathsf{case}(s, t_1, \ldots, t_i, C, t_{i+2}, \ldots t_n). \end{array}$$

We refer to (closed, ground-kinded) terms of the shape $F\, s_1 \ldots s_m$ or $\mathsf{case}(\mathsf{b}_i, s_1, \ldots, s_n)$ as *redexes*. Note that whenever $s \to s'$, there are unique C and Δ such that $s = C[\Delta]$ and Δ is the redex contracted (i.e. $\Delta \to \Delta$ and $s' = C[\Delta]$).

Write $\Sigma^\perp$ for the alphabet Σ extended with symbol $\perp$ of arity 0. Given a term t, we define $t^\perp$ for the (finite) $\Sigma^\perp$-labelled tree defined inductively by (i) $(f\, s_1 \ldots s_n)^\perp := f\, s_1^\perp \ldots s_n^\perp$ (ii) $t^\perp := \perp$ if t is of the form $F\, s_1 \ldots s_n$ or $\mathsf{case}(s, t_1, \ldots, t_n)$. With respect to the standard approximation ordering $\sqsubseteq$ (defined by the compatible closure of $\perp \sqsubseteq t$ for all t), the set of $\Sigma^\perp$-labelled trees is a complete partial order. The *tree language generated by* $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is defined to be the set of $\Sigma^\perp$-labelled trees of the form $\bigsqcup_{i \in I} t_i^\perp$ where I is a prefix of ω, and $\langle t_i \rangle_{i \in I}$ is a maximal (possibly infinite) sequence of closed, ground-kinded terms satisfying:

(outermost) The term $t_0 = S$ and for each $i \in I$, $t_i \to t_{i+1}$ is an *outermost* reduction (i.e. the redex contracted is not a subterm of another redex in t_i)

(fairness) Every outermost redex is eventually contracted i.e. for each $i \in I$ and each outermost redex Δ in t_i, there exists $i' \geq i$ such that Δ is contracted in $t_{i'} \to t_{i'+1}$.[2]

Example 3. The HORSC $\mathcal{G}_2$ is specified by terminal symbols $\mathsf{b}_1 :: d$, $\mathsf{b}_2 :: d$, $\mathit{zero} :: o$, $\mathit{succ} :: o \to o$ and $\mathit{pred} :: o \to o$; non-terminal symbols $S :: o$, $H :: d$ and $G :: (o \to o) \to o$, start symbol S and rules:

$$\begin{array}{lcl} S & \to & \mathsf{case}(H, G\,\mathit{succ}, G\,\mathit{pred}) \\ H & \to & \mathsf{b}_1 \\ H & \to & H \\ G\,g & \to & g\,\mathit{zero} \end{array}$$

It computes the single, finite tree which, when written as a term, is denoted $\mathit{succ}\,\mathit{zero}$ i.e. $[\![\mathcal{G}_2]\!] = \{\, \mathit{succ}\,\mathit{zero} \,\}$.

Remark 1. HORSC extends Kobayashi's *recursion schemes with finite data domains* (RSFD) [14] in several ways: (i) The b_is of HORSC are terminals, but the d_is of RSFD are *data* (distinct from variables, terminals and non-terminals). (ii) In RSFD the return kind of both non-terminals and the case construct must be o. There is no such restriction in HORSC. (iii) RSFD does not handle non-determinism.

A consequence of (i) and (ii) is that in RSFD, the first argument of the case construct must be an atomic datum d_i or a variable. In contrast, the first argument of a case construct in HORSC is an arbitrary term of kind d i.e. any term which may reduce to an element of $\mathcal{B}$ or otherwise diverge. For example, the HORSC $\mathcal{G}_2$ is not a RSFD, since it is non-deterministic, and contains a case construct that has a non-terminal as the first argument.

Deterministic Trivial Tree Automata

We use a simple form of automata over infinite trees to specify properties of the tree languages of HORSC.

Definition 2. A *deterministic trivial tree automaton* (DTT) is a quadruple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ where

(i) Σ is a ranked alphabet;

(ii) Q is a finite set of states containing an initial state q_0;

(iii) $\delta : Q \times \Sigma \rightharpoonup Q^*$ is a (partial) transition function such that if $\delta(q, a) = q_1 \ldots q_n$ then n is the arity of a.

A Σ-labelled tree t is *accepted* by a DTT $\mathcal{A}$ just if there is a Q-labelled tree r, called a *run-tree* of $\mathcal{A}$ over t, satisfying:

(i) $\mathit{dom}(r) = \mathit{dom}(t)$;

(ii) $r(\epsilon) = q_0$;

(iii) for every $\alpha \in \mathit{dom}(r)$, $(r(\alpha), t(\alpha), r(\alpha\,1) \cdots r(\alpha\,m)) \in \delta$ where m is the arity of $t(\alpha)$.

Thus a run tree of $\mathcal{A}$ over t is an annotation of the nodes of t with states that respects δ such that the root is annotated q_0.

Example 4 (A DTT $\mathcal{A}_1$)**.** Take the ranked alphabet Σ of Example 2; $[\![\mathcal{G}_1]\!]$ is accepted by $\mathcal{A}_1 = \langle \Sigma, \{q_0, q_1\}, \delta, q_0 \rangle$, where $\delta : (q_0, c) \mapsto q_1 q_0, (q_1, b) \mapsto q_1, (q_1, a) \mapsto \epsilon$. Thus $\mathcal{A}_1$ accepts a Σ-labelled tree t if, and only if, a and b are seen only on the left of a c.

$$\begin{tikzcd} & c, q_0 \arrow[dl, dash] \arrow[dr, dash] & & & \\ a, q_1 & & c, q_0 \arrow[dl, dash] \arrow[dr, dash] & & \\ & b, q_1 \arrow[d, dash] & & c, q_0 & \\ & a, q_1 & & \vdots & \end{tikzcd}$$

Universal HORSC Model Checking Problem

Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ be a DTT. Define the DTT $\mathcal{A}^{\perp} := \langle \Sigma^{\perp}, Q, \delta', q_0 \rangle$ by $\delta' := \delta \cup \{\, (q, \perp, \epsilon) \mid q \in Q \,\}$ (so that $\mathcal{A}^{\perp}$ will accept any subtree labelled $\perp$ from any state).

Given a HORSC $\mathcal{G}$ and a DTT $\mathcal{A}$, we say that the tree language $[\![\mathcal{G}]\!]$ is *universally accepted* (respectively *existentially*) by the DTT $\mathcal{A}$ just if every (respectively some) element of the tree language $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}^{\perp}$. The *Universal HORSC Model Checking Problem for DTT* is to check whether the language $[\![\mathcal{G}]\!]$ is universally accepted by $\mathcal{A}^{\perp}$. Henceforth, we will refer to this problem simply as the *HORSC Model Checking Problem.*

3. An Intersection and Union Type System

We wish to characterise the HORSC model checking problem as a kind of type inference problem in an intersection type system. In doing so, we not only establish decidability, but also rephrase the question of acceptance as one of bounded search – which is much better understood algorithmically.

Well-Kinded Types

We introduce an intersection and union type system parameterised by a DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ with $\mathcal{B} \subseteq \Sigma$. First we define the set of *well-kinded types* simultaneously with a kinding relation on types, which is defined by induction over the following rules:

$$\frac{q \in Q}{q \ :: \ o} \qquad \frac{B \subseteq \mathcal{B}}{\bigvee B :: d} \qquad \frac{\theta_i :: \kappa_1 \text{ (for all } i \in I) \quad \theta \ :: \ \kappa_2}{(\bigwedge_{i \in I} \theta_i) \to \theta \ :: \ \kappa_1 \to \kappa_2}$$

Any expression σ such that $\sigma :: \kappa$ is derivable in the above system is a well-kinded type. For example, given $Q = \{q_0, q_1\}$, the expressions $q_1 \to q_0$ and $((q_1 \to q_1) \wedge (q_0 \to q_0)) \to q_0$ are *well-kinded types* while $(q_0 \wedge (q_0 \to q_1)) \to q_1$ is not. Note that there are only finitely many well-kinded types of each kind. We write *Type* for the collection of well-kinded types. Henceforth, we will say type to mean well-kinded type.

We write $\bigwedge_{i=1}^{k} \theta_i$ for $\bigwedge\{\theta_1, \cdots, \theta_k\}$, and $\top$ for $\bigwedge \emptyset$; similarly we write $\bigvee_{j=1}^{l} \mathsf{b}_{i_j}$ for $\bigvee\{\mathsf{b}_{i_1}, \cdots, \mathsf{b}_{i_l}\}$and $\perp$ for $\bigvee \emptyset$; further we write $\bigvee\{\mathsf{b}_i\}$ simply as b_i. Note that intersection is only allowed on the left of an arrow; and union is only defined on a subset of $\mathcal{B}$.

Type System

We now present the type system itself. Intuitively, a typing for a term t describes the tree generated by t. For example, the typing $a : q_0$ indicates that the trivial tree a is accepted from state q_0. Intuitively a term has an intersection type if it generates a tree that is acceptable from *every* state in the intersection; a term has a union type if it generates a singleton tree b_i for some i. For example, the typing $\lambda x.s : (q_0 \wedge q_1) \to (\mathsf{b}_0 \vee \mathsf{b}_1)$ says that we have a function

[2] Note that if $s = C[\Delta] \to C[\Delta]$ and Δ is outermost, and Δ' is a different outermost redex in s, then Δ' occurs in C i.e. Δ has a unique residual in $C[\Delta]$.

that takes a tree accepted from both q_0 and q_1 as an argument and returns a tree $s[t/x]$ that is either b_0 or b_1.

A *type environment* (typically Γ) is a finite set of *type bindings*, which are pairs $\xi : \tau$ where ξ is a non-terminal symbol or a variable, and τ is a type. Note that non-terminal symbols and variables are treated in the same way by the system; and different types may be bound to the same symbol in an environment.

A *judgement* is a triple, written $\Gamma \vdash_{\mathcal{A}} t : \theta$, in which Γ is a type environment, θ is a type and t is a λ-term with case construct. A judgement is valid just if it can be derived in the following system:

$$\frac{\theta \text{ is well-kinded}}{\Gamma, x : \theta \vdash_{\mathcal{A}} x : \theta}\ \textsc{Var}$$

$$\frac{\delta(q, a) = q_1 \cdots q_n}{\Gamma \vdash_{\mathcal{A}} a : q_1 \to \cdots \to q_n \to q}\ \textsc{Term}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} s : (\bigwedge_{i \in I} \theta_i) \to \theta \qquad \Gamma \vdash_{\mathcal{A}} t : \theta_i \quad (i \in I)}{\Gamma \vdash_{\mathcal{A}} s\, t : \theta}\ \textsc{App}$$

$$\frac{\Gamma, x : \theta_1, \ldots, x : \theta_n \vdash_{\mathcal{A}} t : \theta \qquad x \notin \Gamma}{\Gamma \vdash_{\mathcal{A}} \lambda x.t : (\bigwedge_{i \in \{1,\ldots,n\}} \theta_i) \to \theta}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t : \bigvee_{i \in I} \mathsf{b}_i \qquad \Gamma \vdash_{\mathcal{A}} t_i : \theta \quad (i \in I)}{\Gamma \vdash_{\mathcal{A}} \mathsf{case}(t, t_1, \ldots, t_n) : \theta}\ \vee\text{-}\textsc{Elim} / \textsc{Case}$$

$$\frac{\exists i \in I \cdot \Gamma \vdash_{\mathcal{A}} t : \mathsf{b}_i}{\Gamma \vdash_{\mathcal{A}} t : \bigvee_{i \in I} \mathsf{b}_i}\ \vee\text{-}\textsc{Intro} / \textsc{Union}$$

$$\frac{}{\Gamma \vdash_{\mathcal{A}} \mathsf{b}_i : \mathsf{b}_i}\ \textsc{Base}$$

Note in particular the final three rules, which cover the addition of case to the term language. In $\vee$-ELIM each possible typing for t requires a proof of typability of the corresponding t_i. The disjunction can only be eliminated here, ensuring that disjunction types cannot be used in other contexts. A canonical typing derivation will reserve the $\vee$-INTRO rule to be used immediately before BASE, delaying the choice of which member type of the disjunction to choose as late as possible. Each $b \in \mathcal{B}$ can be typed by a singleton disjunction of a type of the same name.

Characterisation

Following Kobayashi [9, 13], we characterise the HORSC model checking problem in terms of the existence of certain type environments that are appropriate to the scheme that we are checking.

Definition 3. Fix a HORSC $\mathcal{G}$ and a DTT $\mathcal{A}$. We say that a type environment Γ is $\vdash_{\mathcal{G},\mathcal{A}}$*-complete*, written $\vdash_{\mathcal{G},\mathcal{A}} \Gamma$, just if

(i) $dom(\Gamma) \subseteq \mathcal{N}$

(ii) $\Gamma \vdash_{\mathcal{A}} S : q_0$

(iii) for each $(F : \theta) \in \Gamma$ and for each $\lambda\overline{x}.t \in \mathcal{R}(F)$ we have $\Gamma \vdash_{\mathcal{A}} \lambda\overline{x}.t : \theta$.

Intuitively, a type environment Γ is $\vdash_{\mathcal{G},\mathcal{A}}$*-complete* whenever it contains enough well-kinded typings for the non-terminal symbols in $\mathcal{G}$ so that S can be typed with q_0, but not so many that some are inconsistent with the behaviour of their defining rules.

Theorem 1 (Characterisation)**.** *Given a HORSC $\mathcal{G}$ and a DTT $\mathcal{A}$, $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}^{\perp}$ if, and only if, there exists a $\vdash_{\mathcal{G},\mathcal{A}}$-complete type environment.*

Proof. See Appendix D. □

Given a HORSC $\mathcal{G}$ and a DTT $\mathcal{A}$, the number of non-terminal symbols in $\mathcal{G}$ and the number of well-kinded types is finite. It follows that the problem of the existence of a $\vdash_{\mathcal{G},\mathcal{A}}$-complete type environment is decidable. However, the size of the search space is hyper-exponential in the largest order of the kind of any non-terminal symbol. Thus, in the following section we describe an algorithm which is able to explore this vast expanse in a goal-directed way, which, we will argue in Section 5, gives good performance in practice.

Remark 2. In fact, since the data types in HORSC are finite, the model checking problem can be shown to be decidable by reduction, via determinisation and a Church-style encoding of constants as projection functions, to an instance of the HORS model checking problem. However, such a transformation is known to increase the order and arity of the non-terminal symbols and so is not palatable from a practical point of view.

Example 5 (A typing for $\mathcal{G}_1$)**.** We can see that $\Gamma_1 = \{S : q_0, F : q_1 \to q_0\}$ is $\vdash_{\mathcal{G}_1,\mathcal{A}_1}$-complete, hence, thanks to Theorem 1, $[\![\mathcal{G}_1]\!]$ is accepted by $\mathcal{A}_1$.

4. The HORSC Model Checking Algorithm

Our approach to deciding the HORSC model checking problem exploits the characterisation by the intersection and union type system as stated in Theorem 1. Given a HORSC $\mathcal{G}$ and a DTT $\mathcal{A}$, the decision procedure seeks to construct a $\vdash_{\mathcal{G},\mathcal{A}}$-complete type environment.

Fix $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$. Consider a term $t_0\, t_1 \ldots t_n$ (where t_0 is atomic) which is expected to produce a tree of type q, the canonical example being the term $\mathcal{R}(S)$ and the type q_0. This can be viewed as a typing judgement $\vdash t_0\, t_1 \ldots t_n : q$. Our goal is to construct a derivation for it. After n (bottom-up) applications of the APP rule, a subgoal $\vdash t_0 : \theta$ is generated where $\theta = \alpha_1 \to \cdots \to \alpha_n \to q$ and the α_i are type variables that are as yet undetermined. The values they take on will depend on how t_0 uses its arguments, and we can explore this in a syntax-directed manner.

- Suppose t_0 is a terminal symbol. Since $\delta(q, a)$ is unique, all α_i will be fully determined, yielding n further subgoals, which are judgements of the form $t_i : \alpha_i$ to prove.
- Encountering a non-terminal, say $t_0 = F$, requires us to assume that $F : \theta$ and to build new derivations showing that $s : \theta$ for all $s \in \mathcal{R}(F)$. Bear in mind the characterisation of the problem by $\vdash_{\mathcal{G},\mathcal{A}}$-completeness (Theorem 1).
- In case the symbol t_0 is a variable (i.e. a formal parameter), we must ensure that the corresponding *actual* parameter has the necessary return type.

The use of type variables (such as α_i above) captures the connection made by term variables between typing derivations and enables us to detect the situation where a typing derivation is redundant. A type variable is instantiated to a set of type expressions (called *open types*) which may themselves contain type variables. A derivation need not be explored further when two derivations both aim to show $\mathcal{R}(F) : \theta$, one of which is already complete. We use a restricted system of union types to represent non-deterministic choices in the argument to a case term, which is illustrated in the following example.

$$\dfrac{\dfrac{}{\vdash_{\mathcal{A}} \mathsf{b}_1 : \mathsf{b}_1}\ \textsc{Base}}{\vdash_{\mathcal{A}} \mathsf{b}_1 : \beta}\ \vee\text{-}\textsc{Intro} \qquad \dfrac{\dfrac{\dfrac{}{\{g : \alpha_1\} \vdash_{\mathcal{A}} g : \alpha_2 \to q_0}\ \textsc{Var}}{\{g : \alpha_1\} \vdash_{\mathcal{A}} g\, zero : q_0}\ \textsc{App}}{\vdash_{\mathcal{A}} \lambda g.g\, zero : \alpha_1 \to q_0}\ \textsc{Abs}$$

$$\dfrac{\dfrac{}{\Gamma^o \vdash_{\mathcal{A}} H : \beta}\ \textsc{Var} \qquad \dfrac{\dfrac{}{\Gamma^o \vdash_{\mathcal{A}} G : \alpha_1 \to q_0}\ \textsc{Var} \quad \Gamma^o \vdash_{\mathcal{A}} succ : \alpha_2 \to q_0}{\Gamma^o \vdash_{\mathcal{A}} G\, succ : q_0}\ \textsc{App}}{\Gamma^o \vdash_{\mathcal{A}} \mathsf{case}(H, G\, succ, G\, pred) : q_0}\ \vee\text{-}\textsc{Elim}$$

Table 1. Examples of pre-derivations ($\Gamma^o = \{H : \beta\}$)

Example 6 (Building a derivation for $\mathcal{G}_2 \vDash \mathcal{A}_2$)**.** We consider a simple DTT $\mathcal{A}_2 = \langle\{succ, pred, zero\}, \{q_0, q_1\}, \delta, q_0\rangle$ where δ is the map: $(q_0, succ) \mapsto q_1, (q_1, zero) \mapsto \epsilon$. Starting with the initial goal of showing $S : q_0$, it immediately becomes necessary to build a derivation rooted at $\vdash_{\mathcal{A}} \mathcal{R}(S) : q_0$. Since the right-hand side of S

$$\mathcal{R}(S) = \mathsf{case}(H, G\, succ, G\, pred)$$

is a case construct, we assume that $H : \beta$ where β is a fresh type variable and proceed to explore $\mathcal{R}(H)$ to find which members of the finite domain $\mathcal{B}$ it can reduce to. This leaves us with the derivation

$$\dfrac{\dfrac{}{\{H : \beta\} \vdash_{\mathcal{A}} H : \beta}\ \textsc{Var}}{\{H : \beta\} \vdash_{\mathcal{A}} \mathsf{case}(H, G\, succ, G\, pred) : q_0}\ \vee\text{-}\textsc{Elim}$$

and two further derivations to build, which are rooted at the following, corresponding to the respective right-hand sides of H:

$$\emptyset \vdash_{\mathcal{A}} H : \beta \qquad\qquad \emptyset \vdash_{\mathcal{A}} \mathsf{b}_1 : \beta$$

where β is instantiated to $\bigvee \emptyset = \bot$ initially, avoiding the need to show any typings for the choice terms in the case construct. As usual, $\bot$ represents nontermination, which is exactly the situation that would prevent the case from reducing to any choice term. If the exploration of the scrutinee (here H) ever reduces to a b_i then β will be updated accordingly. Notice that taking the type environment to be $\Gamma = \{S : q_0, H : \beta\}$ in the sense of Theorem 1, the derivations to ensure that the right-hand sides match the typings are already in place, although as yet incomplete. To build a derivation rooted at the right-hand judgement we use the $\vee$-Intro and Base rules (see the derivation on the left in the top row of Table 1). This requires β to contain b_1, causing an additional obligation to type the first choice term ($G\, succ$) of the case construct.

To complete this example, we aim to build a derivation rooted at this new judgement

$$\{H : \beta\} \vdash_{\mathcal{A}} G\, succ : q_0.$$

In order to apply the App rule (in a bottom-up fashion), we introduce another type variable, α_1. Dually to the use of $\vee$-Elim, α_1 is initially instantiated to $\bigwedge \emptyset = \top$, again avoiding the need to prove any typing for $succ$ at this time. Exploring the right-hand side of G (top-right in Table 1), as for H, we find a use of the variable g. Looking at the typing rules, we find that this typing must be justified by the Var rule, which requires "enlarging" α_1, and just as before, after adding the new type to α_1, the use of the App rule to "close" the judgement $\{H : \beta\} \vdash_{\mathcal{A}} G\, succ : q_0$ is no longer valid. We must add an extra judgement for the operand (see the lower derivation in Table 1), which in turn can be justified by the Term rule. This captures informally how we build up the typing derivations. Notice that if we take Γ to be the union of all non-terminal type bindings in the various derivations then (i) $dom(\Gamma) \subseteq \mathcal{N}$; (ii) $\Gamma \vdash_{\mathcal{A}} S : q_0$; (iii) If all judgements are closed then for each $F : \theta$ in Γ, each $t \in \mathcal{R}(F)$, we have $\Gamma \vdash_{\mathcal{A}} t : \theta$. Clearly if the tree language generated by the HORSC is finite, then all judgements will eventually be closed following this approach. However in general we require a more complex termination condition.

Open Types, Instantiation and Reification Maps

We now formalise the method introduced in Example 6. First we introduce *open types*, which represent intersection types using type variables. An open type has the form $\alpha_1 \to \cdots \to \alpha_n \to \beta$ where each variable α_i ranges over finite sets of open types, and $\beta \in Q \cup \mathcal{P}(\mathcal{B})$. Given an instantiation map (to be defined shortly), open types are a representation of types. Assume, for each kind κ, a countably infinite set A_κ of type variables. The set $\mathbb{P}_\kappa$ of *open types* of kind κ is defined by recursion over κ as follows (we use $\theta^o, \theta^o_1, \cdots$ to range over $\mathbb{P}_\kappa$).

$$\mathbb{P}_o := Q \qquad \mathbb{P}_d := \mathcal{P}(\mathcal{B})$$
$$\mathbb{P}_{\kappa_1 \to \kappa_2} := \{\, \alpha \to \theta^o \mid \alpha \in \mathsf{A}_{\kappa_1}, \theta^o \in \mathbb{P}_{\kappa_2} \,\}$$

Let $\mathsf{A} := \bigcup_{\kappa \in Kind} \mathsf{A}_\kappa$ and $\mathbb{P} := \bigcup_{\kappa \in Kind} \mathbb{P}_\kappa$. We say that a function $\Theta : \mathsf{A} \to \mathcal{P}(\mathbb{P})$ is an *instantiation map* if it is (i) *finite*: there exists a finite subset C of A such that Θ maps every element of $(\mathsf{A} \setminus C)$ to $\emptyset$, and (ii) *kind-respecting*: for each kind κ, Θ restricts to a function from A_κ to the set $\mathcal{P}_{\mathrm{fin}}(\mathbb{P}_\kappa)$ of finite subsets of $\mathbb{P}_\kappa$.

Instantiation maps $\Theta : \mathsf{A} \to \mathcal{P}(\mathbb{P})$ are used to reify open types. Given such a map, we derive from it a kind-indexed family of maps on open types, $\widehat{\Theta}_\kappa : \mathbb{P}_\kappa \to Type_\kappa$ with $\kappa \in Kind$, as follows:

$$\widehat{\Theta}_o(q) := q, \qquad \widehat{\Theta}_d(B) := \bigvee B$$
$$\widehat{\Theta}_{\kappa_1 \to \kappa_2}(\alpha \to \theta^o) := \left(\bigwedge\nolimits_{\theta^o_1 \in \Theta(\alpha)} \widehat{\Theta}_{\kappa_1}(\theta^o_1)\right) \to \widehat{\Theta}_{\kappa_2}(\theta^o)$$

Note that for each $\alpha \in \mathsf{A}_{\kappa_1}$, $\Theta(\alpha)$ is a finite subset of $\mathbb{P}_{\kappa_1}$. The map $\widehat{\Theta}_\kappa$ is well-defined by structural induction on κ. We define $\widehat{\Theta} : \mathbb{P} \to Type$ by $\theta^o \mapsto \widehat{\Theta}_\kappa(\theta^o)$ for $\theta^o \in \mathbb{P}_\kappa$, and call it the *reification map*.

Example 7. Let $\kappa = ((o \to o) \to o) \to o \to o$, and take $\theta^o = \alpha_1 \to \alpha_2 \to q_1$, an element of $\mathbb{P}_\kappa$. Let Θ be the instantiation map: $\alpha_1 \mapsto \{\, \alpha_3 \to q_2, \alpha_4 \to q_1 \,\}$, $\alpha_2 \mapsto \{\, q_1 \,\}$, $\alpha_3 \mapsto \emptyset$, $\alpha_4 \mapsto \{\, \alpha_5 \to q_0 \,\}$, $\alpha_5 \mapsto \{\, q_0 \,\}$. Then

$$\widehat{\Theta}(\theta^o) = \bigwedge \{\, \top \to q_2, (q_0 \to q_0) \to q_1 \,\} \to q_1 \to q_1.$$

Open types are used to build up intermediate information about the necessary typings of non-terminal symbols while keeping the relation between these different types explicit in the mapping. This relationship would be lost using concrete types.

For notational convenience we use some further conventions. We use the superscript 'o' to mean *open* (in the sense of containing variables). Thus open types are ranged over by $\theta^o, \theta_1^o, \cdots$; similarly, *open-type environments* are ranged over by $\Gamma^o, \Gamma_1^o, \cdots$. The reification map $\widehat{\Theta}$ is extended to open-type environments Γ^o where it proceeds point-wise. Let $J = \Gamma^o \vdash t : \theta^o$ be an *open-type judgement*. We write $\widehat{\Theta}(J)$ to mean the judgement $\widehat{\Theta}(\Gamma^o) \vdash_{\mathcal{A}} t : \widehat{\Theta}(\theta^o)$. Further, let Δ be a finite tree whose nodes are labelled by open-type judgements (such as typing derivations). We write $\widehat{\Theta}(\Delta)$ to mean the tree that is obtained from Δ by replacing each judgement J by $\widehat{\Theta}(J)$.

Recall that a typing derivation is a tree whose nodes are labelled by judgements; each such judgement is justified by a rule if it labels an internal node, or by an axiom if it labels a leaf node. Informally a *pre-derivation* is a finite tree whose nodes are labelled with open-type judgements. In a pre-derivation, a judgement that occurs at a leaf-node is said to be *closed* if there is a line above it; otherwise it is said to be *open*. A pre-derivation that has no open judgements is said to be *closed*; otherwise it is *open*. We write D for the set of pre-derivations.

The Model Checking Algorithm

The algorithm proceeds by growing a tree $\mathcal{D}$ and an accompanying instantiation map Θ. Each node n of $\mathcal{D}$ is associated with a type binding of the form $(F, s : \theta^o)$ where F is a non-terminal, $s \in \mathcal{R}(F)$ and θ^o is an open type; and n represents the subgoal of building a derivation for the judgement $\Gamma^o \vdash s : \theta^o$ for some open-type environment Γ^o. In the process of constructing such a derivation (in a bottom-up fashion), new derivation subgoals may be created, which are represented by the spawning of new nodes (corresponding to the subgoals); and Θ is updated. The root node is associated with the binding $(S, \mathcal{R}(S) : q_0)$ (recall that we write $S \to \mathcal{R}(S)$ for the unique rule for S), and it represents the original goal, namely, to build a derivation for $\cdots \vdash \mathcal{R}(S) : q_0$.

Formally, a *state* of the algorithm is a pair $(\mathcal{D}, \Theta)$ where $\mathcal{D}$ is a $((\mathcal{R} \times \mathbb{P}) \times \mathsf{D})$-labelled tree, and Θ is an instantiation map. Each node n of $\mathcal{D}$ is labelled by a quadruple, $\mathcal{D}(n) = (F, s : \theta^o, \Delta)$, such that the judgement at the root of the pre-derivation Δ has the form $\Gamma^o \vdash s : \theta^o$ for some term $s \in \mathcal{R}(F)$ and open-type environment Γ^o. Observe that (F, s) uniquely identifies a rule from $\mathcal{R}$. Henceforth we shall refer to Δ as the *pre-derivation* of n, and the triples $(F, s : \theta^o)$ and $(F, s : \widehat{\Theta}(\theta^o))$ respectively as the *open-type binding* and *reified-type binding* of n.

Given a state $(\mathcal{D}, \Theta)$, a node of $\mathcal{D}$ is said to be *closed* if its pre-derivation Δ is closed (and we shall see—Lemma 1—that it follows that $\widehat{\Theta}(\Delta)$ is a valid type derivation of $\vdash_{\mathcal{A}}$); otherwise, the node is *open*. The function open, when applied to $\mathcal{D}$, returns the set of judgements J that is currently open (in some open pre-derivation of $\mathcal{D}$).

The top loop of the algorithm is shown in Algorithm 1 and follows the ideas outlined in Example 6. As mentioned earlier, we must start with the open judgement $\emptyset \vdash \mathcal{R}(S) : q_0$ and this informs the initialisation. (W.l.o.g. we assume that $\mathcal{R}(S)$ is a singleton set.) The *open* judgements, $\Gamma^o \vdash s : \theta^o$, are then closed in turn by application of the appropriate rule (as implemented by one of the six *Close*– procedures), depending on the shape of s.

Termination of the loop depends on the existence of a *complete cut* of a certain initial subtree of $\mathcal{D}$. Fix a state $(\mathcal{D}, \Theta)$. Define

$$\mathcal{D}^{\mathrm{cl}} := \{\, n \in dom(\mathcal{D}) \mid n \text{ and all its } \mathcal{D}\text{-ancestors are closed} \,\}.$$

Thus $\mathcal{D}^{\mathrm{cl}}$ is the largest initial subtree of $\mathcal{D}$ consisting only of closed nodes.

Let t be a Σ-labelled tree. As usual a subset $C \subseteq dom(t)$ is a *cut* of t just if for every maximal path B of t, $B \cap C$ is a singleton set. Let C be a cut of $\mathcal{D}^{\mathrm{cl}}$. We write $n \prec C$ to mean that n is an ancestor of some element of C (read: n is an *interior node* of C); and $n \preccurlyeq C$ to mean that $n \prec C$ or $n \in C$.

Definition 4. We say that a cut C of the tree $\mathcal{D}^{\mathrm{cl}}$ is *complete* if for every $c \in C$, either c is a leaf-node[3] of $\mathcal{D}$, or there is an interior node of C that has the same reified-type binding as c.

(Observe that $\mathsf{open}(\mathcal{D}) = \emptyset$ if, and only if, every node of $\mathcal{D}$ is closed. Hence, if $\mathsf{open}(\mathcal{D}) = \emptyset$, the set of its leaf-nodes is a complete cut; note that $\mathcal{D}$ is finite.)

Algorithm 1: Model Checking

```
input  : HORSC $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$
output: Whether $\mathcal{G} \vDash \mathcal{A}$, with a witness
$\mathcal{D} :=$
singleton tree with label $(S, \mathcal{R}(S) : q_0, \emptyset \vdash \mathcal{R}(S) : q_0)$
$\Theta := \{\alpha \mapsto \emptyset \mid \alpha \in \mathsf{A}\}$
while $\mathcal{D}^{\mathrm{cl}}$ does not have a complete cut do
    foreach $(\Gamma^o \vdash s : \theta^o)$ as $J \in \mathsf{open}(\mathcal{D})$ do
        if $s = t\,u$ then $CloseApp(J)$
        if $s \in \mathcal{N}$ then $CloseNonTerm(J)$
        if $s \in V$ then $CloseVar(J)$
        if $s \in \mathcal{B}$ then $CloseUnion(J)$
        if $s = \mathsf{case}(t, \bar{t})$ then $CloseCase(J)$
        if $s \in \Sigma$ then try $CloseTerm(J)$ with
                        Trace s $\to$ return (NO, $s$)
    end
end
return (YES, $\mathcal{D}$)
```

Procedure $CloseNonTerm$

```
input  : $J = \Gamma^o \vdash F : \theta^o$ in pre-derivation $\Delta$
// $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to \beta$
foreach $\Gamma_1^o \vdash s : \theta_1^o$ in pre-derivation $\Delta$ do
    $\Gamma_1^o := \Gamma_1^o \cup \{F : \theta^o\}$
end
$J := \dfrac{}{\Gamma^o \vdash F : \theta^o}$
foreach $\lambda x_1 \ldots x_n.s \in \mathcal{R}(F)$ do
    Add a fresh node, labelled $(F, \lambda x_1 \ldots x_n.s : \theta^o, J')$, as
    the rightmost child of the node of $\mathcal{D}$ containing $J$ where
    $J' := \dfrac{\{x_i : \alpha_i \mid 1 \le i \le n\} \vdash s : \beta}{\emptyset \vdash \lambda x_1 \ldots x_n.s : \theta^o}$
end
```

Example 8 (Completion of analysing $\mathcal{G}_2$ against $\mathcal{A}_2$)**.** We will now look at the completed data structures, continuing from Example 6. Δ_1, Δ_2 and Δ_3 are the pre-derivations explored in the previous example, with Δ_2, Δ_3 and Δ_4 being required to prove Δ_1, as

[3] which means that $\emptyset \vdash s : \widehat{\Theta}(\theta^o)$ is valid (since c is closed and thanks to Lemma 1, page 8), where $(F, s : \theta^o)$ is the open-type binding of c

Procedure *AddDer*

```
input : Type variable $\alpha$, open type $\theta^o$
// Find intro. of $\alpha$ in pre-derivations of $\mathcal{D}$
if $\exists\,!\ J' = \dfrac{\overline{\Gamma^o \vdash t : \alpha \to \theta^o_1} \quad \cdots}{\Gamma^o \vdash t\ u : \theta^o_1}$ then
    $J' := \dfrac{\Gamma^o \vdash t : \alpha \to \theta^o_1 \quad \Gamma^o \vdash u : \theta^o \quad \cdots}{\Gamma^o \vdash t\ u : \theta^o_1}$
else ($\theta^o = b_i$)
    $\exists\,!\ J' = \dfrac{\overline{\Gamma^o \vdash t : \alpha} \quad \cdots}{\Gamma^o \vdash \mathsf{case}(t, t_1, \ldots, t_n) : \theta^o_1}$
    $J' := \dfrac{\Gamma^o \vdash t : \alpha \quad \Gamma^o \vdash t_i : \theta^o \quad \cdots}{\Gamma^o \vdash \mathsf{case}(t, t_1, \ldots, t_n) : \theta^o_1}$
end
$\Theta := \Theta[\alpha \mapsto \Theta(\alpha) \cup \{\,\theta^o\,\}]$
```

Procedure *CloseApp*

```
input : $J = \Gamma^o \vdash t\ u : \theta^o$
$J := \dfrac{\Gamma^o \vdash t : \alpha \to \theta^o}{\Gamma^o \vdash t\ u : \theta^o}$ ($\alpha$ fresh)
```

Procedure *CloseVar*

```
input : $J = \Gamma^o \vdash x : \theta^o$
$J := \dfrac{}{\Gamma^o \vdash x : \theta^o}$
$AddDer(\Gamma^o(x), \theta^o)$
```

Procedure *CloseTerm*

```
input : $J = \Gamma^o \vdash a : \theta^o$
// $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$
if $(q, a) \notin \delta$ then
    raise (Trace ⟨counter-example trace⟩)
else ($\delta(q, a) = q_1 \ldots q_n$)
    $J := \dfrac{}{\Gamma^o \vdash a : \theta^o}$
    foreach $i \in \{1, \ldots, n\}$ do
        $AddDer(\alpha_i, q_i)$
    end
end
```

Procedure *CloseCase*

```
input : $J = \Gamma^o \vdash \mathsf{case}(t, t_1, \ldots, t_n) : \beta$
$J := \dfrac{\Gamma^o \vdash t : \beta}{\Gamma^o \vdash \mathsf{case}(t, t_1, \ldots, t_n) : \theta^o}$
```

Procedure *CloseUnion*

```
input : $J = \Gamma^o \vdash \mathsf{b}_i : \beta$
$J := \dfrac{\dfrac{}{\Gamma^o \vdash \mathsf{b}_i : \mathsf{b}_i}}{\Gamma^o \vdash \mathsf{b}_i : \beta}$
$AddDer(\beta, \{\mathsf{b}_i\})$
```

can be seen from $\mathcal{D}$ in Table 2. Tracing the computation from Example 6 to this state is left as an exercise to the reader. In this case, the open *pre-derivations* Δ_5 and Δ_6 trivially have the same reified-type binding as Δ_3 and Δ_4 ($H, H : \mathsf{b}_1$ and $H, \mathsf{b}_1 : \mathsf{b}_1$). As a result the environment $\Gamma = \{S : q_0, G : (q_1 \to q_0) \to q_0, H : \mathsf{b}_1\}$ is guaranteed to be a witness to $[\![\mathcal{G}_2]\!] \vDash \mathcal{A}_2$.

Correctness

First we observe that Algorithm 1 never gets stuck: every open judgement is matched by one of the six rules (corresponding to the six *Close-* procedures). We formulate it as an important invariant of the algorithm.

Lemma 1 (Invariant)**.** *Let $(\mathcal{D}, \Theta)$ be a state of the algorithm, n be a node of $\mathcal{D}$, and $\mathcal{D}(n) = (F, s : \theta^o, \Delta)$ where the judgement at the root of the pre-derivation Δ is $\Gamma^o \vdash s : \theta^o$ (where $s \in \mathcal{R}(F)$).*

(i) Every internal judgement (respectively closed judgement) of $\widehat{\Theta}(\Delta)$ is an instance of a rule (respectively axiom) of $\vdash_{\mathcal{A}}$. Hence, if n is closed then $\widehat{\Theta}(\Delta)$ is a valid type derivation, witnessing $\widehat{\Theta}(\Gamma^o) \vdash s : \widehat{\Theta}(\theta^o)$.

(ii) Let $\Gamma^o = \{\, F_1 : \theta^o_1, \ldots, F_l : \theta^o_l \,\}$ and for each i, $\mathcal{R}(F_i) = \{\, s_{i1}, \ldots, s_{ir_i} \,\}$. Then $\bigcup_{i=1}^{l} \{\, n_{i1}, \cdots, n_{ir_i} \,\}$ is the set of successor nodes of n, where $\mathcal{D}(n_{ij}) = (F_i, s_{ij} : \theta^o_i, \Delta_{ij})$ for each i.

Proof. (Sketch) Given $(\mathcal{D}, \Theta)$ we prove that each of the six rules (corresponding to the six *Close-* procedures) preserve these properties. □

Lemma 2. *Let $(\mathcal{D}, \Theta)$ be a state of the algorithm. Suppose there is a complete cut C of $\mathcal{D}^{\mathrm{cl}}$. Define Ξ to be the set:*

$$\Xi := \{\, F : \widehat{\Theta}(\theta^o) \mid \exists n \,.\, n \preccurlyeq C \wedge \mathcal{D}(n) = (F, s : \theta^o, \Delta) \,\}$$

Then $\vdash_{\mathcal{G},\mathcal{A}} \Xi$ (in the sense of Theorem 1).

Proof. Take $n \preccurlyeq C$ with $\mathcal{D}(n) = (F, s : \theta^o, \Delta)$. We need to show that there exists $\Gamma \subseteq \Xi$ such that $\Gamma \vdash_{\mathcal{A}} s : \widehat{\Theta}(\theta^o)$, for *every* $s \in \mathcal{R}(F)$. We may assume that $n \prec C$; for if not, since C is a complete cut, there is some interior node n' of C that has the same reified-type binding as n; and so, we take n' instead of n. Since n is a node in $\mathcal{D}^{\mathrm{cl}}$, by Lemma 1, for some $\Gamma^o = \{\, F_1 : \theta^o_1, \ldots, F_l : \theta^o_l \,\}$, we have

$$F_1 : \widehat{\Theta}(\theta^o_1), \ldots, F_l : \widehat{\Theta}(\theta^o_l) \vdash_{\mathcal{A}} s : \widehat{\Theta}(\theta^o)$$

where the set of successors of n is $N = \bigcup_{i=1}^{l} \{\, n_{i1}, \ldots, n_{ir_i} \,\}$, with $\mathcal{D}(n_{ij}) = (F_i, s_{ij} : \theta^o_i, \Delta_{ij})$ for each $s_{ij} \in \mathcal{R}(F_i)$. If $N = \emptyset$, we are done. Otherwise, take an arbitrary successor of n, say, n_{11}. Since C is a cut, $n_{11} \prec C$ or $n_{11} \in C$. If the latter, since C is complete, there is an interior node of C that has the same reified-type binding as n_{11}. Thus there is a subset N' consisting of interior nodes of C such that the set of reified-type bindings of nodes in N coincide with the set of reified-type bindings of nodes in N', and we are done. Now take $s' \in \mathcal{R}(F)$. By assumption, $s' \preccurlyeq C$, using the same reasoning as before, we can show the desired result.

□

Theorem 2 (Correctness)**.** *Let $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ be a HORSC and $\mathcal{A}$ a DTT.*

(i) If Algorithm 1 returns YES then $\mathcal{A}$ accepts $[\![\mathcal{G}]\!]$.

(ii) If Algorithm 1 returns NO then $\mathcal{A}$ rejects $[\![\mathcal{G}]\!]$.

$$\begin{tikzcd}[column sep=small]
\mathcal{D} = (S, \mathsf{case}(H, G\, succ, G\, pred) : q_0, \Delta_1) \arrow[r] \arrow[dr] \arrow[ddr] & (G, \lambda g.g\; zero : \alpha_1 \to q_0, \Delta_2) & \\
& (H, \mathsf{b}_1 : \beta, \Delta_3) & (H, \mathsf{b}_1 : \beta, \Delta_5) \\
& (H, H : \beta, \Delta_4) \arrow[ur] \arrow[r] & (H, H : \beta, \Delta_6)
\end{tikzcd}$$

$$\Delta_1 = \dfrac{\dfrac{}{\Gamma^o \vdash_{\mathcal{A}} H : \beta}\,\textsc{Var} \qquad \dfrac{\dfrac{}{\Gamma^o \vdash_{\mathcal{A}} G : \alpha_1 \to q_0}\,\textsc{Var} \quad \dfrac{}{\Gamma^o \vdash_{\mathcal{A}} succ : \alpha_2 \to q_0}\,\textsc{Term}}{\Gamma^o \vdash_{\mathcal{A}} G\; succ : q_0}\,\textsc{App}}{\Gamma^o \vdash_{\mathcal{A}} \mathsf{case}(H, G\; succ, G\; pred) : q_0}\,\vee\textsc{-Elim}$$

$$\Delta_2 = \dfrac{\dfrac{\dfrac{}{\Gamma^o \cup \{g : \alpha_1\} \vdash_{\mathcal{A}} g : \alpha_2 \to q_0}\,\textsc{Var} \quad \dfrac{}{\Gamma^o \cup \{g : \alpha_1\} \vdash_{\mathcal{A}} zero : q_1}\,\textsc{Term}}{\Gamma^o \cup \{g : \alpha_1\} \vdash_{\mathcal{A}} g\; zero : q_0}\,\textsc{App}}{\Gamma^o \vdash_{\mathcal{A}} \lambda g.g\; zero : q_0}\,\textsc{Abs}$$

$$\Delta_3 = \dfrac{\dfrac{}{\Gamma^o \vdash_{\mathcal{A}} \mathsf{b}_1 : \mathsf{b}_1}\,\textsc{Base}}{\Gamma^o \vdash_{\mathcal{A}} \mathsf{b}_1 : \beta}\,\vee\textsc{-Intro} \qquad \Delta_5 = \dfrac{\dfrac{}{\Gamma^o \vdash_{\mathcal{A}} \mathsf{b}_1 : \mathsf{b}_1}\,\textsc{Base}}{\Gamma^o \vdash_{\mathcal{A}} \mathsf{b}_1 : \beta}\,\vee\textsc{-Intro}$$

$$\Delta_4 = \dfrac{}{\Gamma^o \vdash_{\mathcal{A}} H : \beta}\,\textsc{Var} \qquad \Delta_6 = \Gamma^o \vdash_{\mathcal{A}} H : \beta$$

$$\Theta = \{\alpha_1 \mapsto \{\alpha_2 \to q_0\}, \alpha_2 \mapsto \{q_1\}, \beta \mapsto \{\mathsf{b}_1\}\} \qquad \Gamma^o = \{G : \alpha_1 \to q_0, H : \beta\}$$

Table 2. A terminating state $(\mathcal{D}, \Theta)$ of the algorithm with input $\mathcal{G}_2$ and $\mathcal{A}_2$ (Example 8).

(iii) Algorithm 1 terminates on every input.

Remark 3 (Round). We organise the computation of the while-loop in Algorithm 1 into *rounds*. In each round, for each $J \in \mathsf{open}(\mathcal{D})$ we apply the appropriate *Close-* procedure repeatedly to the judgement that is opened up, until we reach a non-terminal. Thus at the end of each round, the open judgements (if any) are all non-terminals.

Proof. (i) follows from Lemma 2. (ii) It suffices to show that given a state (Δ, Θ), for every judgement $\Gamma \vdash t : \theta^o \in \Delta$ with q as the result type of θ^o, we can construct a term t' that is a subterm of some u such that $S \to_{\mathcal{G}}^* u$; further if there is a run tree r, then $r(\beta) = q$ where $u(\beta) = t'$. Intuitively this means that every such judgement determines a path in a runtree. For a full proof see Lemma 4 in Appendix B. (iii) Let $\mathcal{N} = \{ F_1, \ldots, F_m \}$ where $F_1 = S$, and let N be the product of the number of rewrite rules and the total number of types (of the relevant kinds) of $\mathcal{G}$. Using our standard notion of round (Remark 3), the open nodes of a state tree $\mathcal{D}$ are necessarily leaf nodes. To show termination of the algorithm (in case of a yes-instance), we aim to exhibit a state tree such that every path in it is either sufficiently long to guarantee a recurrence of a reified type binding, or it ends in a closed node. To this end, we systematically compute all traversals. For each traversal, we keep on extending it until we reach a closed node, or it has induced a path in the state tree of length greater than N. Termination of such a computation of traversals is a consequence of [16, Lemma 14 (long version)]. An alternative argument from first principles is Lemma 3, which is proved in Appendix C. ☐

Correspondence with Traversals

The computation of the algorithm can be represented by a possibly infinite tree, called *justified judgement tree*, which is defined to be a (justified) tree of judgements (i.e. the nodes are labelled by judgements; we shall refer to a node by its label) such that J' is a *successor* of J just if the execution of the call $Close\Xi(J)$ (where the suffix Ξ, which is one of $Abs, NonTerm, Var, Term, Case$ and $Union$, is determined by the head symbol of the term in J) constructs the open judgement J' either in the same pre-derivation as J or in a new pre-derivation. Thus each path in the justified judgement tree represents a sequence of judgements that are successively closed by one of the six $Close\Xi$ procedures. The judgement tree is *justified* in the sense that some nodes have a pointer back to an ancestor node. In Appendix F, we show that the justified judgement tree and the *traversal tree* (in the sense of [16]) are isomorphic with respect to both the successor and pointer relations.

Theorem 3 (Correspondence)**.** *(i) There is a bijective map Φ from maximal paths in the traversal tree to maximal paths in the computation tree. (ii) Further, for every maximal path π, the Σ-projection of π to Σ coincides with the Σ-projection of $\Phi(\pi)$.*

Lemma 3. *If a traversal is well-founded (in the sense that there exists $N \geq 0$ such that all paths that are induced in the state tree have length less than N) then it is finite.*

Optimisations

A crucial optimisation is *Actual Parameter Revisit Avoidance*. Fix a node n with open-type binding $F : \theta^o$, and a variable x that occurs more than once in $\mathcal{R}(F)$. Suppose at state $(\mathcal{D}, \Theta)$, the open judgement $J_2 = \Gamma^o \vdash x : \theta_2^o$ in the pre-derivation of n (call it Δ) is chosen and $CloseVar(J_2)$ is called with $\Gamma^o(x) = \alpha$ and $\theta_0^2 = \beta_1 \to \cdots \to \beta_n \to q$. Suppose at an earlier state, a judgement of the form $J_1 = \Gamma^o \vdash x : \theta_1^o$ with $\theta_1^o = \alpha_1 \to \cdots \to \alpha_n \to q$ was closed (and let J_1 be the first such), and so, we have $\theta_1^o \in \Theta(\alpha)$. Then we optimise as follows.

(i) When executing $AddDer(\alpha, \theta_2^o)$ as called by $CloseVar(J_2)$, do not search for J' nor update it (using the notation of the procedure $AddDer$), instead, after executing $\Theta(\alpha) := \Theta(\alpha) \cup \{ \theta_2^o \}$, perform: for each i, $\Theta(\beta_i) := \Theta(\alpha_i)$; and for each $\theta'^o \in \Theta(\beta_i)$, call $AddDer(\beta_i, \theta'^o)$.

(ii) Subsequently, every call to $AddDer(\alpha_i, \theta'^o)$ automatically triggers a call to $AddDer(\beta_i, \theta'^o)$, for $i \in \{ 1, \cdots, n \}$.

To see why the optimisation is sound, consider $AddDer(\alpha, \theta_1^o)$, which constructs a new open judgement $\Gamma_1^o \vdash u : \theta_1^o$ (say). Eventually for some $i \in \{ 1, \cdots, n \}$, $AddDer(\alpha_i, \theta'^o)$ is called, which performs the update $\Theta(\alpha_i) := \alpha_i \cup \{ \theta'^o \}$ with control then returning the original pre-derivation Δ, seeking to prove a typing θ'^o for the ith-argument (of the first occurrence) of x. Let this sequence of calls to a *Close* procedure between $AddDer(\alpha, \theta_1^o)$ and $AddDer(\alpha_i, \theta'^o)$ be Υ_{i,θ'^o}. Now consider a call to $CloseVar(J_2)$ which calls $AddDer(\alpha, \theta_2^o)$, and which constructs a new open judgement $\Gamma_1^o \vdash u : \theta_2^o$ (say). Note that for each i and θ'^o, Υ_{i,θ'^o} determines a corresponding sequence of calls to a *Close* procedure between $AddDer(\alpha, \theta_2^o)$ and $AddDer(\beta_i, \theta'^o)$. The optimisation removes such call sequences for each i and θ'^o (but not their effects). Our experiments (see Section 5) demonstrate that the optimisation results in close to an order-of-magnitude improvement for HORS of orders 4 or higher.

Translated into the language of traversals, the optimisation says that if the traversal reaches a variable x with state q, instead of jumping to the actual parameter of x, one can immediately traverse downwards with state q' to the i-child of x, provided the traversal has visited another occurrence of x before with state q and subsequently visiting its (the earlier occurrence's) i-child with state q'.

The *canonical types* optimisation aids with the critical part of a *complete cut* (and thus termination) is finding two nodes with the same concrete type bindings. We can increase the chance of finding two such nodes using subtyping to yield canonical types. Given any intersection type $\bigwedge_{i \in I} \theta_i \to \theta$ it is sufficient to consider instead $\bigwedge_{j \in J} \theta_j \to \theta$ where $J \subseteq I$ and for all $k \in I \setminus J$, there exists some $j \in J$ such that $\theta_j \leq \theta_k$ (where $\leq$ is standard intersection type subtyping). Intuitively, this θ_k may be removed because θ_j already places a stronger requirement on a parameter to this function. Any typing tree that uses $x : \theta_k$ could therefore be replaced with one that uses $x : \theta_j$ instead. Removing these redundant types during reification of open types allows us to consider a smaller space of *canonical types*.

At a lower-level, *reification caching* was introduced to handle the relatively expensive calculation of $\widehat{\Theta}$ as the requirement to search for a cut after each round of operation led this to dominate the runtime of the algorithm. By caching the result of $\widehat{\Theta}$ for each α and maintaining a dependency mapping (such that if $\alpha' \in \widehat{\Theta}(\alpha)$ then α depends on α') we can avoid the majority of Θ lookups while preserving correctness by invalidating cache entries in the transitive closure of the dependencies for any α that we update.

Finally, an unguided execution of the algorithm can yield a vast number of subgoals very quickly. Every time a terminal symbol of arity n is encountered, the number of subgoals rises by $n - 1$. To address this, our implementation uses a search guided by the termination check. While searching for a *complete cut* using a breadth-first search of $\mathcal{D}$, any subtree rooted at a node with a type binding already seen is not explored, and any open judgements within this subtree are not expanded at this time. This focuses the attention of the algorithm on areas of the tree that could not currently form part of a *complete cut*. In the extremal case, all open judgements are contained in such subtrees, and the algorithm terminates.

5. Empirical Results and Evaluation

We have constructed TRAVMC, an implementation of Algorithm 1 presented in Section 4. The implementation, and all the examples presented here, can be accessed through a web interface at `http://mjolnir.cs.ox.ac.uk/horsc/`. For comparison we have considered not just HORSC, but also standard HORS, which can be handled by our algorithm as a degenerate case.

Instance	O	S	R	H	G	T	T_B	T'
example2-1	1	2	Y	2	1	34	0	33
fileocamlc	4	21	Y	8	1680	60	23	718
fileocamlc2	4	22	Y	7	1980	58	18	918
fileorder5-2	5	30	Y	109	–	201	167	–
filewrong	4	11	N	0	–	86	47	85
flow	4	7	Y	1	3	32	0	32
g35	3	11	Y	–	136	–	–	–
g41	4	8	Y	–	608	55	15	–
lock2	4	11	Y	10	–	64	23	132
m91	5	25	Y	39	–	429	381	–
order5	5	9	Y	5	–	62	8	46
order5-variant	5	11	Y	12	–	47	7	317
stress	1	13	Y	29	3	187	133	180

Table 3. HORS MC comparison

HORSC	O	S	R	T	T_H
checknz	1	27	Y	46	36
checkpairs	1	86	N	53	93
filepath	1	369	Y	1950	–
filter-nonzero	4	49	N	74	156
filter-nonzero-1	4	69	Y	1756	–
last	1	60	Y	71	45
map-head-filter	2	110	N	62	116
map-head-filter-1	2	190	Y	1080	1538
map-plusone	4	39	Y	83	161
map-plusone-1	4	49	Y	296	860
map-plusone-2	4	63	Y	4144	–
mkgroundterm	1	108	Y	179	96
risers	1	165	Y	113	127
safe-foldr1	2	145	Y	450	625
safe-head	2	106	Y	71	56
safe-init	2	235	Y	209	288
safe-tail	2	154	Y	88	74

RSFD	O	S	R	T	H
gap_id	3	26	Y	248	15
homrep	4	12	Y	1767	7
merge_addr	1	7	Y	52	1
mult	1	5	Y	52	1
remove_b	2	7	Y	54	2
xhtmlm-drop-a	1	33	Y	1252	146
xhtmlm_id	1	33	Y	996	64
xhtmls-remove-meta	1	13	Y	277	9
xhtmlf_id	1	51	Y	–	456

Table 4. HORSC MC results

HORS Model Checking

For HORS, we have used a benchmark suite containing a number of examples from the literature, along with some fresh examples. The columns "O", "S" and "R" in the table indicate the order, number of rules and result of the example respectively. The "H" and "G" columns contain timing data (in milliseconds) for Kobayashi's hybrid (TRECS version 1.32) and game-based algorithms (GTRECS version 0.10[4]). Those labelled "T" or "T_B" (resp. "T'") are for the algorithm introduced in this paper with (resp. without) the *Revisit Avoidance* optimisation at order 1, the subscript B indicating a 'batch' processing mode. Where an algorithm did not terminate within 10 seconds this is indicated by "–".

[4] We did not have access to a GTRECS binary, as a result experiments were carried out through the author's web interface. Timings are not directly comparable, but indicative.

Table 3 shows that for most examples TRAVMC performs approximately an order of magnitude slower than the current version of TRECS. However, given the immature state of our implementation, we believe that this gap may be crossed given careful optimisation. For the very rapid examples (around 100ms and below), we found that the runtime was dominated by the first round of expansion. We believe that this is JIT overhead tied to our use of F# on .NET (both TRECS and GTRECS are implemented in OCaml). This is supported by our batch mode experiment, which saw all examples processed consecutively by a single invocation of the model checker, avoiding the repeated startup overhead commonly associated with JIT compilers and reduced the runtime by around 50ms consistently. One area where we believe significant speedups may be gained are in extending the *Actual Parameter Revisit Avoidance* optimisation to orders 2 and above. Although some savings are still made at higher orders in the current implementation, the amount of work which is potentially avoided can be increased exponentially by extending the optimisation to each order. Furthermore, in order to keep the cost of checking the termination condition low, it is currently somewhat conservative, but it is possible that a more thorough procedure, if carefully engineered, could potentially detect termination earlier. Exploring this trade-off could provide substantial benefits.

It is worth noting that both TRECS and TRAVMC could handle almost all of the examples without trouble, implying that further work on more taxing examples is needed to better understand where each algorithm breaks down. One direction in which both algorithms struggled is a set of examples introduced by Kobayashi [11] known as $\mathcal{G}_{n,m}$. When checked by the hybrid algorithm, these examples require $\mathcal{O}(\mathbf{exp}_n(m))$ expansions to obtain type information for non-terminals at the bottom of a hyper-exponential tree. Our new algorithm's performance improved markedly due to the *Revisit Avoidance* optimisation, checking $\mathcal{G}_{4,1}$ even faster than Kobayashi's linear-time algorithm GTRECS, although higher values of n and m resulted in timeouts. We believe the speedup will be lifted to higher values of n with a full implementation of the *Revisit Avoidance* optimisation.

Such examples display the power of GTRECS fully and it is encouraging to note that TRAVMC seems to be able to handle some such recursion schemes. In more realistic cases, TRAVMC outperforms GTRECS by several orders of magnitude.

HORSC Model Checking

For HORSC, we have generated some examples as the output of an abstraction procedure based on earlier work [17]. The abstraction procedure operates on a *pattern-matching recursion scheme* (PMRS), which can be thought of as an instance of a simply-typed programming language with higher-order, recursive functions and pattern-matching over algebraic data-types. The abstract models that are produced are not strictly HORSC, since they can have patterns on the left-hand side of grammar rules which include free variables (though such variables are not allowed to appear on the right-hand side of grammar rules), so they are first put through a translation which is detailed in Appendix E. For some examples (those with numbers appended) we performed refinement of the abstraction and here we give the timings for each round of model checking. See Table 4, where the columns are labelled as before.

In order to evaluate the usefulness of a primitive case analysis construct, which is afforded by HORSC, we have compared the results of checking these HORSC model checking instances with corresponding HORS encodings (using TRAVMC in both cases). In each case, the HORS encoding of the HORSC is obtained by determinising and modelling the constants as projection functions. Unavoidably, this raises the order and arity, and hence worst-case complexity significantly (see Remark 2). The time to check the original instance is given in column "T" and to check the encoding can be seen in the column "T_H". For some examples, particularly the simpler ones, checking HORS is fast enough, but as the size and order of the example increases, this approach breaks down. We believe that this offers a compelling argument for the introduction of HORSC.

Pattern-match safety An important verification problem in functional programming is that of ensuring that partial pattern matches never receive one of the missing cases and so are 'safe'. Pattern-match safety is reducible to reachability, and the results for these can be seen at the top of the table. One simple example is the list-processing function *last*, which assumes that its input is a non-empty list. The CATCH tool [15] targets this verification problem, and we have used some of the same examples: the *Risers* program and *Safe* and *FilePath* libraries, which contain partial pattern matching that we verify to be safe. The input HORSC is in both cases rather large, but the algorithm still terminates quickly.

A more complex example uses *filter* to remove empty lists from the input before invoking *head* on the remaining lists (*map-filter-head*). The *mkgroundterm* program contains a counting function that sums the values of constants within a ground term. By guarding the input to this partial function (by removing variables), we are able to prove that the program is safe.

Output term While pattern-match safety reduces to reachability, we can check more interesting properties such as verifying some structure of the output of a function. The *filter-nonzero* example uses *filter* with a *nonzero* function and verifies that the output list contains no element equal to zero. For the *map-plusone* example, we add one to all elements of an input list of naturals and verify again that the output list contains no zeroes.

RSFD Kobayashi, Tabuchi and Unno model check *recursion schemes with finite data domains* (RSFD) as part of their work [14]. RSFD form a sub-class of HORSC in which there are additional typing restrictions on the scrutinee appearing in each case analysis. Since each RSFD can be viewed as a HORSC, our tool is also able to solve the RSFD model checking problem. We have compared the performance of our tool (column "T") versus the TRECS (version 1.32) tool of Kobayashi *et al.* (column "H") in the second part of Table 4. The data reveals that, perhaps unsurprisingly, the specialist RSFD checker is more efficient in all examples. Indeed, the particular additional restrictions imposed in the definition of RSFD make the class particularly appealing from an algorithmic point of view, though one which is not expressive enough for our purposes. However, even at higher orders or with a large number of automaton states, our tool can solve almost all the example instances.

6. Related Work

MSO Model Checking Problem The MSO model checking problem for order-n recursion schemes was first proved to be decidable (with optimal complexity of n-EXPTIME) by Ong [16]. His proof used game semantics to reduce the model checking problem to the solution of parity games over *variable profiles*. To date, three other proofs are known, employing different methods to build appropriate parity games. Hague et al. [5] constructed configuration graphs of collapsible pushdown automata; Kobayashi and Ong [13] used intersection types; and Salvati and Walukiewicz [19] appealed to

Krivine machines. For the restricted class of *trivial automata* (but for the full hierarchy of HORS), Aehlig [1] gave a decidability proof based on a novel finite semantics for simply-typed lambda calculus. Kobayashi's proof of the same result, which was based on intersection types [9], used a similar idea.

Practical Model Checking Algorithms for HORS As discussed in the Introduction, the first practical model checking algorithm for HORS against trivial automata was Kobayashi's *hybrid algorithm* [8], which was implemented in the model checker TRecS [10]. There are important differences between the hybrid algorithm and our traversal algorithm. The hybrid algorithm extracts intersection types by partial evaluation of the HORS followed by an over-approximation; whereas the traversal algorithm (following game semantics) harvests *variable profiles* from the traversals in game semantics. Secondly the hybrid algorithm uses a loop—each iteration being a greatest fixpoint construction starting from a seed type environment—which will eventually compute a $\vdash_{\mathcal{G},\mathcal{A}}$-complete type environment in case $(\mathcal{G}, \mathcal{A})$ is a yes-instance. In contrast, the traversal algorithm builds a $\vdash_{\mathcal{G},\mathcal{A}}$-complete type environment "from below".

Kobayashi's FoSSaCS'11 algorithm [11] is inspired by game semantics, even though the formal development of the algorithm is purely type-theoretic, and no concrete relationship with game semantics is known. A notable feature of the algorithm is its simplicity, which consists of two fixpoint constructions, first least then greatest. Thanks to Rehof and Mogensen's optimisation [18], a consequence of the fixpoint design is its linear-time complexity in the size of the HORS, assuming that the other parameters are fixed. The main innovation of the algorithm lies in the least fixpoint computation. Given a candidate type environment Γ, for each subset $\Gamma_1 \subseteq \Gamma$, and each $F : \theta \in \Gamma$, more "expansive" versions of Γ_1 and θ, namely, Γ' and θ' (satisfying $\Gamma_1 \preceq_O \Gamma'$ and $\theta \preceq_P \theta'$) respectively, are selected such that $\Gamma' \vdash \mathcal{R}(F) : \theta'$. (The expansive relations $\preceq_O$ and $\preceq_P$ represent Opponent and Proponent moves respectively.) The type environment that is constructed in the next iteration consists of Γ extended by $\Gamma' \cup \{ F : \theta' \}$, for all $F : \theta$ and for all such Γ' and θ'. Our traversal algorithm may be viewed as a process of approximating a (canonical) $\vdash_{G,\mathcal{A}}$-complete type environment from below. There are however two differences. First the successive approximants are not related by containment. Secondly, our algorithm selects just one such pair of Γ' and θ', as determined by the traversal development.

7. Conclusions and Further Directions

We have presented a practical algorithm for the universal model checking problem for higher-order recursion schemes with cases (HORSC) against deterministic trivial automata. The algorithm is based on *traversals*, and is induced by the fully abstract game semantics of the recursion schemes, but presented as a goal-directed construction of derivations in an intersection and union type system. We view HORSC model checking as a suitable backend for an approach to verify functional programs (presented as *pattern-matching recursion schemes*) via an abstraction-refinement procedure. Preliminary experiments with our tool implementation TravMC indicate that the algorithm performs remarkably well on a number of small but realistic examples generating schemes with hundreds of rules. We hope to explore the scalability of our approach by verifying larger examples of pure functional programs from the literature.

8. Acknowledgements

We would like to thank our reviewers for their helpful comments on the first version of this paper. We would also like to thank Naoki Kobayashi for his assistance when benchmarking against his tools, and for offering an automated approach for performing the translation from HORSC to HORS (see Section 5).

References

[1] Klaus Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. *Logical Methods in Comp. Sci.*, 3(3), 2007.

[2] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[4] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics & abstract machines. In *LICS*, pages 394–405, 1996.

[5] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, pages 452–461, 2008.

[6] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[7] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS*, pages 205–222, 2002.

[8] Naoki Kobayashi. Model-checking higher-order functions. In *PPDP*, pages 25–36, 2009.

[9] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, pages 416–428, 2009.

[10] Naoki Kobayashi. http://www-kb.is.s.u-tokyo.ac.jp/~koba/trecs/. 2009.

[11] Naoki Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FOSSACS*, pages 260–274, 2011.

[12] Naoki Kobayashi and C.-H. Luke Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In *ICALP (2)*, pages 223–234, 2009.

[13] Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188, 2009.

[14] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL*, pages 495–508, 2010.

[15] Neil Mitchell and Colin Runciman. Not all patterns, but enough - an automatic verifier for partial but sufficient pattern matching. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 49–60. ACM, September 2008.

[16] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90, 2006. Long version (55 pp.) http://www.cs.ox.ac.uk/people/luke.ong/personal/publications/ntree.pdf.

[17] C.-H. Luke Ong and Steven J. Ramsay. Verifying functional programs with pattern matching algebraic data types. In *POPL*, pages 587–598, 2011.

[18] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. *Sci. Comput. Program.*, 35(2):191–221, 1999.

[19] Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. In *ICALP (2)*, pages 162–173, 2011.

A. Proof of Lemma 1

We prove that $\mathcal{D}$ always contains valid *pre-derivations* by showing that each of the six rules (corresponding to the six *Close*- procedures) preserves this property. Let Δ be a Θ-pre-derivation. We consider each of the four procedures that may be applied to an open judgement $J = \Gamma^o \vdash u : \theta^o$ in Δ in turn.

- $\mathit{CloseApp}$; $u = s\,t$. Δ' is obtained from Δ by an application of the APP rule where $n = 0$.
- $\mathit{CloseNonTerm}$; $u = F$. After augmenting the typing environment of every judgement in Δ by the binding $F : \theta^o$, the original judgement J is closed by the VAR rule; hence Δ' is a Θ-pre-derivation. The new node that is created is labelled by a pre-derivation which is obtained by the equivalent of n applications of the ABS rule to $\emptyset \vdash \mathcal{R}(F) : \theta^o$.
- $\mathit{CloseVar}$; $u = x$. The original judgement J is again closed by the VAR rule, through the extension of the Θ-mapping for $\Gamma^o(x) = \alpha$ by θ^o. The point of introduction of α must be of the shape of J' in AddDer. The use of APP in the right-hand fragment remains valid as the enlargement of the intersection represented by α is matched exactly by the introduction of the open node $\Gamma^o_1 \vdash t : \theta^o$.
- $\mathit{CloseTerm}$; $u = a$. The judgement J is closed using the TERM rule. As shown in the $\mathit{CloseVar}$ case, the procedure AddDer preserves the validity of the *pre-derivations* by adding a new derivation for each extension of a type variable. The closure of the judgement is made valid by setting each α_i to $\{ q_i \}$ as required by TERM.

B. A Reduction Lemma

Lemma 4 (Reduction)**.** *Given a state (Δ, Θ), for every judgement $\Gamma \vdash t : \theta^o \in \Delta$, where $state(\theta^o) = q$, there exists a corresponding term t' that is a subterm of some u such that $u(\beta) = t'$ for some occurrence β, and $S \to^*_{\mathcal{G}} u$ where the reduction sequence is in a 1-1 correspondence with applications of the $CloseNonTerm$ rule. If there exists a run tree r, then $r(\beta) = q$.*

Proof. For each $\Gamma^o \vdash t : \theta^o$, the corresponding t' can be defined by a function reify defined as follows:

$$\mathsf{resolve}(\Gamma^o, t) := t[\mathsf{intro}(\Gamma^o(x))/x \mid x \in \mathsf{FV}(t)]$$

$$\mathsf{intro}(\alpha_j) := \mathsf{resolve}(\Gamma^o, t_j) \text{ where } \frac{\overline{\Gamma^o_1 \vdash t : \alpha_1 \to \cdots \alpha_n \to \theta^o} \qquad \cdots}{\Gamma^o_1 \vdash t\, t_1 \ldots t_n : \theta^o}$$

$$\mathsf{reify}(\Gamma^o \vdash t : \overline{\alpha} \to q) := \mathsf{resolve}(\Gamma^o, t)\, \mathsf{intro}(\alpha_1) \ldots \mathsf{intro}(\alpha_n)$$

We show that $t' := \mathsf{reify}(\Gamma^o \vdash t : \theta^o)$ is a witness to the lemma by induction over the rules of the algorithm.

(i) Base case $\emptyset \vdash \mathcal{R}(S) : q_0$. In this case $t' = \mathsf{reify}(\emptyset \vdash \mathcal{R}(S) : q_0) = \mathcal{R}(S)$. This is certainly a subterm of $u = \mathcal{R}(S)$ where $S \to_G \mathcal{R}(S)$. Given that $t' = u$, we have $t' = u(\epsilon)$ and as required $r(\epsilon)$ must necessarily be q_0.

(ii) $\mathit{CloseNonTerm}$. From the induction hypothesis there is a witness of the form $t' = F\ t_1 \ldots t_m$. The new t'' is

$$\mathsf{reify}(\{x_i : \alpha_i \mid 1 \le i \le m\} \vdash s : q) = s[t_1/x_1, \ldots, t_m/x_m]$$

where $F\ t_1 \ldots t_m \to_{\mathcal{G}} s[t_1/x_1, \ldots, t_m/x_m]$. The inductive t' is a subterm of u and t'' is a subterm of u' (where $u \to_{\mathcal{G}} u'$ by reducing the redex that is t'). Clearly then if $S \to^k u$, $S \to^{k+1}_{\mathcal{G}} u'$ and $u'(\beta) = t''$. As we are still talking about the same location in the run tree (β), it must still be labelled q, which is the type in the new judgement.

(iii) $\mathit{CloseVar}$. This is the simplest case where the inductive case gives the same term as the new case:

$$\begin{aligned} t' &= \mathsf{reify}(\Gamma^o \vdash x\ t_1 \ldots t_n : \theta^o) \\ &= \mathsf{reify}(\Gamma^o_1 \vdash u : \alpha_1 \to \cdots \to \alpha_n \to \theta^o). \end{aligned}$$

(iv) $\mathit{CloseTerm}$. Here we progress the constraints on the run tree. The induction hypothesis gives us $t' = \mathsf{reify}(\Gamma^o \vdash a\ t_1 \ldots t_n : \theta^o)$ as a witness where q is the rightmost atomic type. If $(q, a) \notin \delta$ we are trivially done. Otherwise for each of these new judgements of the form $\Gamma^o_1 \vdash t_i : q_i$ it is clear that applying reify gives a subterm of t' which will be $u(\beta\, i)$. From the definition of a value tree and $S \to^*_{\mathcal{G}} u$, $[\![\mathcal{G}]\!](\beta) = a$. Given the definition of a run tree necessarily for each $i \in \{1, \ldots, m\}$, $r(\beta\, i) = q_i$ and we are done.

□

C. Proof of Lemma 3

Well-founded reduction sequence

Let $\twoheadrightarrow$ be the union of the standard HORS reduction $\to$ with the relation

$$\{ (f\, s_1 \ldots s_{arity(f)}, s_i) \mid f \in \Sigma, 1 \le i \le arity(f) \}.$$

Given a possibly infinite $\twoheadrightarrow$-reduction τ (say), starting from the start symbol S

$$\tau \ :\ S = u_1 \twoheadrightarrow u_2 \twoheadrightarrow u_3 \twoheadrightarrow \ldots$$

such that the non-terminals that are introduced in the reduction step $u_\ell \to u_{\ell+1}$ (by unfolding the head symbol of u_ℓ) are coloured ℓ, with ℓ ranging over $\{ 1, 2, \ldots \}$.

Let $\mathcal{N}^\omega = \{ F^j_i \mid 1 \le i \le m, j \in \omega \}$ whereby the colour of F^j_i is the superscript j. We introduce a relation, $> \subseteq \mathcal{N}^\omega \times \mathcal{N}^\omega$, as follows:

$$F^\ell_i > F^{\ell'}_j \text{ just if the head symbol of } u_{\ell'} \text{ is } F^\ell_i$$

Thus (omitting subscripts for simplicity) writing $\twoheadrightarrow^*$ for the reflexive, transitive closure of $\twoheadrightarrow$, $F^\ell > F^{\ell_1} > F^{\ell_2} > F^{\ell_3} > \ldots$ means that

$$\begin{aligned} & \underbrace{(F^\ell \overline{u})}_{u^{\ell_1}} \twoheadrightarrow (\ldots F^{\ell_1} \ldots)\overline{[u_i/x_i]} \\ \twoheadrightarrow^* \ & \underbrace{(F^{\ell_1} \overline{v})}_{u^{\ell_2}} \twoheadrightarrow (\ldots F^{\ell_2} \ldots)\overline{[v_i/x_i]} \\ \twoheadrightarrow^* \ & \underbrace{(F^{\ell_2} \overline{w})}_{u^{\ell_3}} \twoheadrightarrow (\ldots F^{\ell_3} \ldots)\overline{[w_i/x_i]} \twoheadrightarrow \ldots \end{aligned}$$

We say that τ is *well-founded* just if there is no infinite sequence of the form:

$$S = F_1 > F^{\ell_1}_{i_1} > F^{\ell_2}_{i_2} > F^{\ell_3}_{i_3} > \ldots$$

The following is a key technical lemma in the soundness proof of Kobayashi and Ong's type system [13].

Theorem 4. *If a possibly-infinite reduction sequence π is well-founded, then it is finite.*

Linear head reduction and traversals

Next we introduce a term-based formulation of *traversal* which is based on *linear head reduction* [4]. For convenience, we use a shorthand for closed, ground-kinded terms, writing

$$\left[\begin{array}{c}\overline{u}\\ \overline{x}\end{array}\right] s \;:=\; (\lambda\overline{x}.s)\,\overline{u}$$

and

$$\left[\begin{array}{c}\overline{u_1}\\ \overline{x_1}\end{array}\right]\left(\left[\begin{array}{c}\overline{u_2}\\ \overline{x_2}\end{array}\right] s\right) \;=\; \left[\begin{array}{cc}\overline{u_1} & \overline{u_2}\\ \overline{x_1} & \overline{x_2}\end{array}\right] s$$

Thus we have

$$\left[\begin{array}{c}\overline{u_1}\cdots\overline{u_n}\\ \overline{x_1}\cdots\overline{x_n}\end{array}\right] s \;=\; (\lambda\overline{x_1}.(\lambda\overline{x_2}.\ldots((\lambda\overline{x_n}.s)\,\overline{u_n})\ldots)\,\overline{u_2})\,\overline{u_1}$$

Let $n \geq 0$ and $\mathbb{M} := \left[\begin{array}{c}\overline{u_1}\cdots\overline{u_n}\\ \overline{x_1}\cdots\overline{x_n}\end{array}\right]$ where $\overline{x_i} = x_{i1}\ldots x_{ir_i}$ and $\overline{u_i} = u_{i1}\ldots u_{ir_i}$. We define a reduction relation $\curvearrowright$ over coloured terms by the following rules, assuming that variables are renamed afresh where necessary.

(VAR) $\mathbb{M}\,(x_{ij}\,\overline{s}) \;\curvearrowright\; \mathbb{M}\,(u_{ij}\,\overline{s})$

(TERM) $\mathbb{M}\,(f\,s_1\ldots s_r) \;\curvearrowright\; \mathbb{M}\,s_i$ where $i \in \{\,1,\ldots,r\,\}$ with $r = \mathit{arity}(f)$.

(NONTERM) $\mathbb{M}\,(F^{\ell'}\,\overline{u}) \;\curvearrowright\; \mathbb{M}\left(\left[\begin{array}{c}\overline{u}\\ \overline{x}\end{array}\right] t'\right)$ where $\mathcal{R}(F) = \lambda\overline{x}.t$ and $t' = t\overline{[F^{\ell}_i/F_i]}$, assuming that the redex is the ℓ-th term of a given reduction sequence.

Whenever we write $\mathbb{M}\,s$, it is assumed that the term is closed and ground-kinded, and s is not a β-redex. By abuse of terminology, we say that the head symbol of a term $\mathbb{M}\,(\$\,\overline{u})$ is $\$$. Fix a possibly-infinite linear head reduction sequence starting from S.

$$\tau \;:\; S = u_1 \curvearrowright u_2 \curvearrowright u_3 \curvearrowright \ldots$$

Similarly as before, we define a relation $> \subseteq \mathcal{N}^{\omega} \times \mathcal{N}^{\omega}$:

$F^{\ell}_i > F^{\ell'}_j$ just if the head symbol of $u_{\ell'}$ is F^{ℓ}_i

Next we define a function $\textsc{sub}()$:

$$\textsc{sub}(\left[\begin{array}{c}\\ \\\end{array}\right] s) := s \qquad \text{where } s \text{ is not a } \beta\text{-redex}$$

$$\textsc{sub}(\left[\begin{array}{c}\overline{u}\\ \overline{x}\end{array}\right] s) := (\textsc{sub}(s))\overline{[u_i/x_i]}$$

Lemma 5. *If $s \curvearrowright s'$ by rule ρ then $\textsc{sub}(s) \overset{n}{\rightarrowtriangle} \textsc{sub}(s')$ where*

$$n \;=\; \begin{cases} 1 & \textit{if } \rho \textit{ is } (\textsc{Term}) \textit{ or } (\textsc{NonTerm}) \\ 0 & \textit{otherwise i.e. } \rho \textit{ is } (\textsc{Var}) \end{cases}$$

Proof. Using the notation $\mathbb{M}$ as above, we write $(\text{-})^*$ to mean the iterated substitution $(\text{-})\overline{[u_{ni}/x_{ni}]}\ldots\overline{[u_{1i}/x_{1i}]}$.

(i) ρ is (NONTERM): We have $\textsc{sub}(\mathbb{M}\,(F\,\overline{u})) = F\,\overline{u^*} \rightarrowtriangle t\overline{[u^*_i/x_i]}$; and

$$\textsc{sub}(\mathbb{M}\,(\left[\begin{array}{c}\overline{u}\\ \overline{x}\end{array}\right] t)) = (t\overline{[u_i/x_i]})^* = t^*\overline{[u^*_i/x_i]} = t\overline{[u^*_i/x_i]}$$

because the variables in t are fresh (disjoint from x_{ij}).

(ii) ρ is (TERM): $\textsc{sub}(\mathbb{M}\,(f\,s_1\ldots s_r)) = f\,s^*_1\ldots s^*_r \rightarrowtriangle s^*_i$; and $\textsc{sub}(\mathbb{M}\,(s_i)) = s^*_i$.

(iii) ρ is (VAR): $\textsc{sub}(\mathbb{M}\,(x_{ij}\,\overline{s})) = u^{\dagger}_{ij}\,\overline{s^*}$; and $\textsc{sub}(\mathbb{M}\,(u_{ij}\,\overline{s})) = u^{\dagger}_{ij}\,\overline{s^*}$, where $(\text{-})^{\dagger}$ means $(\text{-})\overline{[u_{nj}/x_{nj}]}\ldots\overline{[u_{i+1,j}/x_{i+1,j}]}$.

□

Let τ be a possibly-infinite linear head reduction sequence

$$\tau \;:\; S = u_1 \curvearrowright u_2 \curvearrowright u_3 \curvearrowright \ldots$$

such that for each i, $u_{n_i} \curvearrowright u_{n_i+1}$ is an instance of the rule (NONTERM). Thanks to Lemma 5, there is a corresponding reduction sequence

$$\tau' \;:\; S = v_1 \rightarrowtriangle v_2 \rightarrowtriangle v_3 \rightarrowtriangle \ldots$$

and an increasing sequence of numbers, $m_1 < m_2 < m_3 < \cdots$, such that

(i) the terms that are headed by a non-terminal are

$$v_{m_1}, v_{m_2}, v_{m_3}, \ldots$$

(ii) for each $i \geq 0$, $\textsc{sub}(\widetilde{u_{n_i}}) = v_{m_i}$, where $\widetilde{(\text{-})}$ erases the colours (note that the v_is are not coloured).

It follows that τ is well-founded if, and only if, τ' is well-founded; and τ is finite if, and only if, τ' is. Hence, Lemma 3 follows from Theorem 4.

D. Proof of Theorem 1

We follow Kobayashi's argument [9].

We write $\vdash_{\mathcal{A}} (\mathcal{G}, t) : \sigma$ to mean that there exists a finite type environment Γ (of bindings of non-terminals of $\mathcal{G}$) such that for all $F : \theta \in \Gamma$

(i) θ is a prime type of the kind of $\mathcal{R}(F)$

(ii) $\Gamma \vdash_{\mathcal{G},\mathcal{A}} \lambda\overline{x}.t : \theta$ for all $\lambda\overline{x}.t \in \mathcal{R}(F)$

(iii) $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t : \sigma$.

Lemma 6 (Subject Reduction). *If $\vdash_{\mathcal{A}} (\mathcal{G}, t) : \sigma$ and $t \to t'$ then $\vdash_{\mathcal{A}} (\mathcal{G}, t') : \sigma$.*

The proof is a straightforward case analysis of the rules for proving $t \to t'$.

We extend the standard definition of the *concrete part* of t, $t^{\perp}$ by the clause $\mathsf{case}(s, \overline{t})^{\perp} := \perp$. The automaton $\mathcal{A}^{\perp}$ is as defined by Kobayashi [9].

Lemma 7. *If $\vdash_{\mathcal{A}} (\mathcal{G}, t) : q$, then (the tree) $t^{\perp}$ is accepted by $\mathcal{A}^{\perp}$ from state q.*

The same proof in [9] works. The proof of soundness then follows exactly the argument in [9].

For completeness, we first define $[\![t]\!]$ to mean $[\![\mathcal{G}']\!]$ where $\mathcal{G}' := \langle\Sigma, \mathcal{N} \cup \{\,S'\,\}, \mathcal{R} \cup \{\,S' \to t\,\}, S'\rangle$. The type system is augmented by the axiom $\Gamma \vdash_{\mathcal{G}',\mathcal{A}} \perp : q$ for all $q \in Q$.

Lemma 8. *If $t^{\perp}$ (which is a Σ-labelled tree) is accepted by $\mathcal{A}^{\perp}$ from state q, then $\vdash_{\mathcal{A}^{\perp}} (\mathcal{G}, t^{\perp}) : q$.*

Lemma 9 (Subject Expansion). *Let β be a ground-kinded type and Γ a type environment. If $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t' : \beta$ for all outermost reductions $t \to t'$, then $\vdash_{\mathcal{A}} (\mathcal{G}, t) : \beta$.*

Proof. We prove by a case analysis of the syntactic shape, and an induction on the structure, of t.

- Suppose $t = \mathsf{case}(s, t_1, \ldots, t_n)$, say. Either $s = \mathsf{b}_i$ for some i, or it is not. Suppose the former, say, $s = \mathsf{b}_1$. If $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t_1 : \beta$ then we can construct a derivation of $\Gamma \vdash_{\mathcal{G},\mathcal{A}} \mathsf{case}(\mathsf{b}_1, t_1, \ldots, t_n) : \beta$ from $\Gamma \vdash_{\mathcal{G},\mathcal{A}} \mathsf{b}_1 : \bigvee \{ \mathsf{b}_1 \}$ and $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t_1 : \beta$.

  Now suppose $s \neq \mathsf{b}_i$. By assumption, there exists Γ such that for each $s \to s_k$, we have $\Gamma \vdash_{\mathcal{G},\mathcal{A}} \mathsf{case}(s_k, t_1, \ldots, t_n) : \beta$, which gives derivations for $\Gamma \vdash_{\mathcal{G},\mathcal{A}} s_k : \bigvee_{j \in J_k} \mathsf{b}_j$, and $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t_j : \beta$ for each $j \in J_k$. Now take $J = \bigcup_k J_k$. Thanks to the ($\vee$-INTRO)-rule, we have $\Gamma \vdash_{\mathcal{G},\mathcal{A}} s_k : \bigvee_{j \in J} \mathsf{b}_j$ for each k. Hence, by the IH, we have $\Gamma' \vdash_{\mathcal{G},\mathcal{A}} s : \bigvee_{j \in J} \mathsf{b}_j$ for some Γ'. Thus, using it, and the derivation of $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t_j : \beta$ for each $j \in J$, we can construct a derivation of $\Gamma \cup \Gamma' \vdash_{\mathcal{G},\mathcal{A}} \mathsf{case}(s, t_1, \ldots, t_n) : \beta$ as required.

- Suppose $t = F\, s_1 \ldots s_n$ and $\mathcal{R}(F) = \{ \lambda\overline{x}.t_1, \ldots, \lambda\overline{x}.t_l \}$. Take an arbitrary $j \in \{ 1, \ldots, l \}$. By assumption $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t_j[\overline{s}/\overline{x}] : \beta$. Let P_i^j be the set of prime types assigned to s_i in the derivation for $\Gamma \vdash_{\mathcal{G},\mathcal{A}} t_j[\overline{s}/\overline{x}] : \beta$ (note that $P_i^j = \emptyset$ in case x_i does not occur in t_j). Then we can construct a derivation for

  $$\Gamma, x_1 : \bigwedge P_1^j, \ldots, x_n : \bigwedge P_n^j \vdash_{\mathcal{G},\mathcal{A}} t_j : \beta.$$

  Now let Γ' be the type environment obtained from Γ by replacing $\Gamma(F)$ by

  $$\Gamma(F) \wedge (\bigwedge_{j=1}^{l} \bigwedge P_1^j \to \ldots \to \bigwedge_{j=1}^{l} \bigwedge P_n^j \to \beta)$$

  Then we have a derivation for $\Gamma' \vdash_{\mathcal{G},\mathcal{A}} F\, s_1 \ldots s_n : \beta$.

- $t = f\, s_1 \ldots s_i \ldots s_n$. The simple induction is omitted.

□

Suppose $[\![\mathcal{G}]\!] \subseteq \mathcal{L}(\mathcal{A})$. Let $m \geq 0$. We construct a recursion-free version of $\mathcal{G}$, $\mathcal{G}^m$ as follows. Let the non-terminals of $\mathcal{G}$ be $F_1, \ldots, F_n$. The non-terminals of $\mathcal{G}^m$ are $F_{i,j}$ where $i \in \{ 1, \ldots, n \}$ and $j \in \{ 1, \ldots, m \}$. Suppose $\mathcal{G}$ has rewrite rules $F_i\, \overline{x} \to t_{i1} \mid \ldots \mid t_{il_i}$. Then $\mathcal{G}^m$ has the following rules. For $i \in \{ 1, \ldots, n \}$

$$\begin{array}{ll} F_{i,k+1}\, \overline{x} \to t_{i1}\dagger_k \mid \ldots \mid t_{il_i}\dagger_k & \text{for } k \in \{ 0, \ldots, m-1 \} \\ F_{i,0}\, \overline{x} \to \bot & \end{array}$$

where $\dagger_k$ is the substitution (-)$[F_{1,k}/F_1, \ldots, F_{n,k}/F_n]$. By definition, $[\![\mathcal{G}^m]\!]$ approximates $[\![\mathcal{G}]\!]$ in the sense that for each $t \in [\![\mathcal{G}^m]\!]$, there exists $t' \in [\![\mathcal{G}]\!]$ such that t is obtained from t' by replacing some subtrees by $\bot$. Hence $[\![\mathcal{G}^m]\!]$ is accepted by $\mathcal{A}^\bot$. It follows from Lemma 8 that $\vdash_{\mathcal{A}^\bot} (\mathcal{G}^m, t) : q_0$ for each $t \in [\![\mathcal{G}^m]\!]$. Since each $t \in [\![\mathcal{G}^m]\!]$ is a finite Σ-labelled tree, we have $S \to^* t$. And so, by Lemma 9, $\vdash_{\mathcal{A}^\bot} (\mathcal{G}^m, S) : q_0$. It follows from the construction of $\mathcal{G}^m$ that $\Gamma_{\mathrm{gfp}} \vdash_{\mathcal{G},\mathcal{A}} S : q_0$, as required.

E. Transformation from wPMRS to HORSC

For each type base type b, we will use the term *b-base* to refer to any set of patterns P of type b which is both:

(Exhaustive) For every $t : b \in T(\Sigma)$ there exists a substitution σ and a pattern $p \in P$ such that $\sigma p = t$.

(Non-overlapping) For every p and q in P, if there exist substitutions σ and τ such that $\sigma p = \tau q$ then necessarily $p = q$.

We define a function H which, given a b-base:

$$I = \left\{ \begin{array}{c} \xi_1\, p_1^1 \cdots p_1^m, \\ \cdots, \\ \xi_k\, p_k^1 \cdots p_k^m \end{array} \right\}$$

maps I to the set of exhaustive sets of bases constructed from the sub-patterns:

$$\left\{ p_i^j \;\middle|\; i \in [1..k],\, j \in [1..m] \right\}$$

The function H is defined to act as follows:

$$\begin{array}{rl} & H(I) \\ := & \left\{ F(\xi_i\, p_i^1 \cdots p_i^{j-1} \,\square\, p_i^{j+1} \cdots p_i^m) \;\middle|\; i \in [1..k],\, j \in [1..m] \right\} \end{array}$$

where the auxilliary function F is defined such that the image of the term

$$\xi\, p_1 \cdots p_{j-1} \,\square\, p_{j+1} \cdots p_m$$

under F is given by the set:

$$\begin{array}{r} \{ q \mid \exists\sigma,\, \tau \cdot \forall i \in ([1..m] \setminus \{ j \}) \cdot \exists q_i \cdot \sigma q_i = \tau p_i\ \& \\ \xi\, q_1 \cdots q_{j-1}\, q\, q_{j+1} \cdots q_m \in I \} \end{array}$$

So, given a term s of the base I containing a hole, F computes the set of patterns which can be matched in the position of the hole by any term which is an instance of s. For example, consider the base:

$$I_1 = \left\{ \begin{array}{ccc} a & z & (s\, x), \\ a & z & z, \\ a & (s\, z) & y, \\ a & (s\, (s\, z)) & y, \\ b & & \end{array} \right\}$$

Applying F to the third term with the hole in the first argument $F(a \,\square\, y)$ yields the set $\{ z,\, s\, z,\, s\, (s\, z) \}$ (because terms that are instances of $a \square y$ are also instances of $a \square z$ and $a \square (sz)$). Applying F to the fourth term with the hole in the first argument yields $\{ y \}$.

Every set in the range of F is a base.

Lemma 10. *Let I be a base and $\xi\; p_1 \cdots p_{j-1} \,\square\, p_{j+1} \cdots p_m$ be some term in I with one type-b argument replaced by a hole. For all $t \in T(\Sigma)$ of type b, there exists a pattern $q \in F(\xi\; p_1 \cdots p_{j-1} \,\square\, p_{j+1} \cdots p_m)$ and a substitution θ such that $\theta q = t$.*

Proof. Let σ be a closed substitution such that, for all $i \in ([1..m] \setminus \{ j \})$, $\sigma p_i \in T(\Sigma)$. Then $t' := \xi \sigma p_1 \cdots \sigma p_{j-1} t \sigma p_{j+1} \cdots \sigma p_m \in T(\Sigma)$. Hence, there is some element of the base $\xi\; q_1 \cdots q_m \in I$ and substitution τ such that $\tau(\xi\; q_1 \cdots q_m) = t'$. Consequently, for all $i \in ([1..m] \setminus \{ j \})$, $\sigma p_i = \tau q_i$ and, by definition, $q_j \in F(\xi\, p_1 \cdots p_{j-1} \,\square\, p_{j+1}\, p_m)$. □

Lemma 11. *Let I be a base and $\xi\; p_1 \cdots p_{j-1} \,\square\, p_{j+1} \cdots p_m$ be some term in I with one type-b argument replaced by a hole. For all $p,\, q \in F(\xi\; p_1 \cdots p_{j-1} \,\square\, p_{j+1} \cdots p_m)$, if p and q unify, then $p = q$.*

Proof. Let $q,\, q' \in F(\xi\; p_1 \cdots p_{j-1} \,\square\, p_{j+1} \cdots p_m)$ and let there exist substitutions σ and τ such that $\sigma q = \tau q'$. Then, by definition of F, there are witnessing terms $\xi\; q_1 \cdots q_{j-1}\; q\; q_{j+1} \cdots q_m$ and $\xi\; q_1' \cdots q_{j-1}'\; q'\; q_{j+1}' \cdots q_m'$ in I, that is: there exist substitutions α, θ, β and α' such that, for all $i \in ([1..m] \setminus \{ j \})$, $\alpha p_i = \theta q_i$ and $\beta p_i = \theta' q_i'$. Assume, for the purposes of obtaining a contradiction, that $q \neq q'$. Then the two witnesses are distinct and, since I is a base, they necessarily do not unify. Hence, there must be some $l \in ([1..m] \setminus \{ j \})$ such that q_l and q_l' do not unify. For the purposes of the exposition, say that this l is less than j (the other case is handled symmetrically). Since the elements of the base are variable disjoint, we can construct the compound substitution $\hat{\theta} := \theta \cup \theta'$. Similarly, since the elements of a base are linear, we can consider the substitution:

$$\epsilon(x) := \begin{cases} \beta(x) & \text{if } x \in \mathsf{FV}(p_l) \\ \alpha(x) & \text{otherwise} \end{cases}$$

Consider next, the term:

$$t := \xi\ \hat{\theta}q_1 \cdots \hat{\theta}q_{l-1}\ \hat{\theta}q_l'\ \hat{\theta}q_{l+1} \cdots \hat{\theta}q_{j-1}\ q\ \hat{\theta}q_{j+1} \cdots \hat{\theta}q_m$$

and let γ be a closed substitution so that $\gamma t \in T(\Sigma)$. Since I is base, it contains an element $\xi\ r_1 \cdots r_m$ and there is a substitution δ such that $\gamma t = \xi\ \delta r_1 \cdots \delta r_m$. This element of the base is also a witness in the computation of $F(\xi\ p_1 \cdots p_{j-1}\ \Box\ p_{j+1} \cdots p_m)$ since, for all $i \in ([1..m] \setminus \{\, l,\, j \,\})$, $\gamma(\epsilon p_i) = \gamma(\alpha p_i) = \gamma(\theta q_i) = \gamma(\hat{\theta} q_i) = \delta r_i$ and $\gamma(\epsilon p_l) = \gamma(\beta p_l) = \gamma(\theta' q_l) = \gamma(\hat{\theta} q_l) = \delta r_l$. Hence $\xi\ r_1 \cdots r_m$ $\Box$

We construct an RSFD*as $\mathcal{W}^\# = \langle \Sigma^\#, \mathcal{N}^\#, \mathcal{R}^\#, S^\# \rangle$ over the single type o. First define a mappings on base types as follows: $\mathsf{treeType}(b) = o$ and $\mathsf{pairType}(b) = (o \to o \to o) \to o$. We abuse notation by identifying the mappings with their unique homomorphic extensions on all simple types. The construction follows:

$$H_F(\tau) = \begin{cases} \mathsf{pairType}(\tau) & \text{when } F[i] \text{ is pure} \\ \mathsf{treeType}(\tau) & \text{otherwise} \end{cases}$$

$$J_F(x) = \begin{cases} x & \text{when } F[i] \text{ is pure} \\ \Pi_2\ x & \text{otherwise} \end{cases}$$

$$A_F = B_1 \times \cdots \times B_n$$

where $F : \tau_1 \to \cdots \to \tau_n \to \tau \in \mathcal{R}$ and

$$B_i = \begin{cases} \{\, x \,\} & \text{if } F[i] \text{ is pure} \\ \Sigma_{\tau_i} & \text{otherwise} \end{cases}.$$

$$\Sigma_b^\# = \{\, c_q \mid q \in \mathsf{pats}(P_b) \,\}$$

$$\Sigma^\# = \bigcup_{b \in B} \Sigma_b^\#$$

and the definition of $\mathcal{N}^\#$ and $\mathcal{R}^\#$ are in Table 5

F. Correspondence with Traversals

We assume the reader is familiar with *traversals* and *traversal trees* in the sense of Ong [16]. For technical convenience, we consider an equivalent version of Algorithm 1 for HORS that are presented in η-long normal form. For ease of exposition, we assume deterministic HORS; the extension of the correspondence to non-deterministic HORS is straightforward. In the following we let \$ range over terminals (Σ), non-terminals ($\mathcal{N}$) and variables (V).

Justified Judgement Tree

Henceforth we fix a HORS $\mathcal{G}$ in η-long normal form, and a deterministic trivial automaton $\mathcal{A}$. The computation of the algorithm can be represented by a possibly infinite tree, called *justified judgement tree*, which is defined to be a (justified) tree of judgements (i.e. the nodes are labelled by judgements; we shall refer to a node by its label) such that J' is a *successor* of J just if the execution of the call $Close\Xi(J)$ (where the suffix Ξ, which is one of

Algorithm 2: η-Long HORS Model Checking

```
input  : HORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$
output: Whether $\mathcal{G} \vDash \mathcal{A}$, with a witness
$\mathcal{D}$ := singleton tree with label $(S : q_0, \emptyset \vdash \mathcal{R}(S) : q_0)$
$\Theta := \{\alpha \mapsto \emptyset \mid \alpha \in \mathsf{A}\}$
while $\mathsf{open}(\mathcal{D}) \neq \emptyset$ do
    foreach $(\Gamma^o \vdash t : \theta^o)$ as $J \in \mathsf{open}(\mathcal{D})$ do
        if $t = \lambda x_1 \cdots x_m.\$\, t_1 \cdots t_n$ then $CloseAbs(J)$
        if $t \in \mathcal{N}$ then $CloseNonTerm(J)$
        if $t \in V$ then $CloseVar(J)$
        if $t \in \Sigma$ then try $CloseTerm(J)$ with Trace t $\to$
        return $(\mathrm{No}, t)$
    end
end
return $(\mathrm{Yes}, \mathcal{D})$
```

Procedure *AddDer*

```
input  : Type variable $\alpha_i$, open type $\theta^o$
// Find intro. of $\alpha_i$ in pre-derivations of $\mathcal{D}$
// Let $\theta_1^o = \alpha_1 \to \cdots \to \alpha_n \to q$

$\exists\,!\,J' = \dfrac{\overline{\Gamma_1^o \vdash \$ : \theta_1^o} \qquad \cdots}{\Gamma_1^o \vdash \$\, t_1 \cdots t_n : q}$

$J' := \dfrac{\overline{\Gamma_1^o \vdash \$ : \theta_1^o} \qquad \Gamma_1^o \vdash t_i : \theta^o \qquad \cdots}{\Gamma_1^o \vdash \$\, t_1 \cdots t_n : q}$
$\Theta := \Theta[\alpha_i \mapsto \Theta(\alpha_i) \cup \{\, \theta^o \,\}]$
```

Procedure *CloseAbs*

```
input  : Judgement $J = \Gamma^o \vdash \lambda x_1 \cdots x_m.\$\, t_1 \cdots t_n : \theta^o$
// $\theta^o = \alpha_1 \to \cdots \to \alpha_m \to q$
$J := \dfrac{\dfrac{\Gamma^o \cup \{\overline{x} : \overline{\alpha}\} \vdash \$ : \beta_1 \to \cdots \to \beta_n \to q}{\Gamma^o \cup \{\overline{x} : \overline{\alpha}\} \vdash \$\, t_1 \cdots t_n : q}\ (\overline{\beta}\text{ fresh})}{\Gamma^o \vdash \lambda x_1 \cdots x_m.\$\, t_1 \cdots t_n : \theta^o}$
```

Procedure *CloseNonTerm*

```
input  : Judgement $J = \Gamma^o \vdash F : \theta^o$ in pre-derivation $\Delta$
// $\theta^o = \alpha_1 \to \cdots \to \alpha_m \to q$
foreach judgement $\Gamma_1^o \vdash s : \theta_1^o$ in pre-derivation $\Delta$ do
    $\Gamma_1^o := \Gamma_1^o \cup \{F : \theta^o\}$
end
$J := \overline{\Gamma^o \vdash F : \theta^o}$ // Make $J$ an axiom
Add a fresh node, labelled $(F : \theta^o, J')$, as the rightmost
child of the node of $\mathcal{D}$ containing $J$ where
$J' :=\ \emptyset \vdash \mathcal{R}(F) : \theta^o$
```

Procedure *CloseVar*

```
input  : Judgement $J = \Gamma^o \vdash x : \theta^o$

$J := \overline{\Gamma^o \vdash x : \theta^o}$ // Make $J$ an axiom
$AddDer(\Gamma^o(x), \theta^o)$
```

$$
\begin{array}{rl}
\mathcal{N}^{\#} = & \{\, K_k : \mathsf{treeType}(\tau) \mid k:\tau \in \Sigma \,\} \\
\cup & \{\, P_k : \mathsf{pairType}(\tau) \mid k:\tau \in \Sigma \,\} \\
\cup & \{\, F^{\#} : \mathsf{pairType}(\tau) \mid F \in \Sigma \,\} \\
\cup & \{\, F : H_F(\tau_1) \to \cdots \to H_F(\tau_n) \to \mathsf{pairType}(\tau) \mid F : \tau_1 \to \cdots \to \tau_n \to \tau \,\} \\
\cup & \{\, K_i : o \to o \to o \mid i \in [1,2] \,\} \\
\cup & \{\, \Pi_i : ((o \to o \to o) \to o) \to o \mid i \in [1,2] \,\} \\
\cup & \{\, \mathit{Pair} : o \to o \to (o \to o \to o) \to o \,\} \\
\cup & \{\, S^{\#} : o \,\}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{R}^{\#} = & \left\{ K_k\, c_{q_1} \cdots c_{q_n} \longrightarrow \mathsf{abs}_b(k\, q_1 \cdots q_n) \;\middle|\; \begin{array}{l} k : b_1 \to \cdots \to b_n \to b \in \Sigma \\ \& \quad \forall i \in [1..n] \cdot c_{q_i} \in \Sigma_{b_i} \end{array} \right\} \\
\cup & \{\, P_k\, x_1 \cdots x_n\, f \longrightarrow \mathit{Pair}(k\,(\Pi_1\, x_1) \cdots (\Pi_1\, x_n))(K_k\,(\Pi_2\, x_1) \cdots (\Pi_2\, x_n))\, f \mid k : n \in \Sigma \,\} \\
\cup & \left\{ P_k\, x_1 \cdots x_n\, f \longrightarrow \mathit{Pair}(k\,(\Pi_1\, x_1) \cdots (\Pi_1\, x_n)) c_q\, f \;\middle|\; \begin{array}{l} k \in \Sigma^n \\ \exists \theta \cdot \theta(k\, x_1 \cdots x_n) = q \end{array} \right\} \\
\cup & \{\, F^{\#}\, x_1 \cdots x_n\, f \longrightarrow F\,(J_F\, x_1) \cdots (J_F\, x_n)\, f \mid F \in \mathcal{N}^n \,\} \\
\cup & \left\{ F\, s_1 \cdots s_n\, f \longrightarrow \theta t[P_k/k][F^{\#}/F] \;\middle|\; \begin{array}{l} F\, p_1 \cdots p_n \longrightarrow t \in \mathcal{R} \\ \langle s_1, \ldots, s_n \rangle \in A_F \\ \exists \theta \cdot \langle \mathit{pat}(s_1), \ldots, \mathit{pat}(s_n) \rangle = \theta \langle p_1, \ldots, p_n \rangle \end{array} \right\} \\
\cup & \{\, K_i\, x_1\, x_i \longrightarrow x_i \mid i \in [1,2] \,\} \\
\cup & \{\, \Pi_i\, p \longrightarrow p\, K_i \mid i \in [1,2] \,\} \\
\cup & \{\, \mathit{Pair}\, x\, y\, f \longrightarrow f\, x\, y \,\} \\
\cup & \{\, S^{\#} \longrightarrow \Pi_1\, S \,\}
\end{array}
$$

Table 5. Definition of $\mathcal{N}^{\#}$ and $\mathcal{R}^{\#}$.

Procedure $\mathit{CloseTerm}$

```
input : Judgement $J = \Gamma^o \vdash a : \theta^o$
// $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$
if $(q, a) \notin \delta$ then
    raise (Trace ⟨counter-example trace⟩)
else ($\delta(q, a) = q_1 \cdots q_n$)
    $J := \overline{\Gamma^o \vdash a : \theta^o}$ // Make $J$ an axiom
    foreach $i \in \{1, \cdots, n\}$ do
        $AddDer(\alpha_i, q_i)$
    end
end
```

$Abs, NonTerm, Var$ and $Term$, is determined by the head symbol of the term in J) constructs the open judgement J' either in the same pre-derivation as J or in a new pre-derivation. Thus each path in the justified judgement tree represents a sequence of judgements that are successively closed by one of the four $Close\Xi$ procedures. The judgement tree is *justified* in the sense that some nodes have a pointer back to an ancestor node.

Our aim is to show that the justified judgement tree and the *traversal tree* (in the sense of [16]) are isomorphic with respect to both the successor and pointer relations.

Let $(\mathcal{D}, \Theta)$ be a state of the algorithm. Take an open judgement $J = \Gamma^o \vdash s : \theta^o$ in some pre-derivation of $\mathcal{D}$. It is a consequence of the design of the algorithm that the term s has one of the following forms:

- *Lambda*: $\lambda x_1 \cdots x_m.\$\, t_1 \cdots t_n$ where $m, n \geq 0$
- *Non-Lambda*: $\$ \in \mathcal{N} \cup \Sigma \cup V$

where V is the set of variables that occur in $\mathcal{G}$. Note that we distinguish $\lambda.a$ and a: the former—which has a "dummy lambda"—is in Lambda form, and the latter is in Non-Lambda form.

Recall that a traversal of the HORS $\mathcal{G}$ is a justified sequence of elements of the set

$$\{\, @_F \mid F \in \mathcal{N} \,\} \cup V \cup \Sigma \cup \{\, \lambda \overline{x}. \mid \overline{x} \subseteq V \,\}$$

We can now define the traversal determined by a path in the justified judgement tree. In case J has a Lambda form, the traversal node determined by it is the expression $\lambda x_1 \cdots x_n.$ (or simply $\lambda \overline{x}.$). The execution of $\mathit{CloseAbs}(J)$ replaces J in its pre-derivation Δ (say) by

$$\cfrac{\cfrac{\Gamma^o \cup \{\overline{x} : \overline{\alpha}\} \vdash \$: \beta_1 \to \cdots \to \beta_n \to q}{\Gamma^o \cup \{\overline{x} : \overline{\alpha}\} \vdash \$\, t_1 \cdots t_n : q} \;(\overline{\beta}\text{ fresh})}{\Gamma^o \vdash \lambda x_1 \cdots x_m.\$\, t_1 \cdots t_n : \theta^o}$$

I.e. J has successor node

$$J' \;=\; \Gamma^o \cup \{\overline{x} : \overline{\alpha}\} \vdash \$: \beta_1 \to \cdots \to \beta_n \to q$$

There are three cases.

(i) $\$ = F$, a non-terminal. Then J' does not have a pointer. The traversal node determined by J' is $@_F$.

(ii) $\$ = y_j$, a variable. Then J' points to the judgement J'' in the same pre-derivation Δ whose term is the Lambda form $\lambda \overline{y}.s$; i.e. J' introduced $\overline{y}$ to Γ^o. The traversal node determined by J' is \$ which points to the traversal node determined by J'', namely, $\lambda \overline{y}..$

(iii) $\$ = a$, a terminal. Then J' has no pointer. The traversal node determined by J' is \$.

In case J has a Non-Lambda form \$, there are three cases.

(i) $\$ = F$, a non-terminal. The execution of $CloseNonTerm(J)$ adds a fresh node to $\mathcal{D}$ with the following singleton pre-derivation

$$\emptyset \vdash \mathcal{R}(F) : \theta^o$$

where $\mathcal{R}(F) = \lambda\overline{x}.\$\, t_1 \cdots t_n$. The traversal node determined by J' is $\lambda\overline{x}$. which points to $@_F$, the traversal node determined by J.

(ii) $\$ = x_i$, a variable. Then J determines the traversal node x_i. The execution of $CloseVar(J)$ calls $AddDer(\Gamma^o(x_i), \theta^o)$ which creates the open judgement $J' = \Gamma^o_1 \vdash t_i : \theta^o$ (using the notation in the procedure $AddDer$), which is J's successor. Note that t_i is in Lambda Form.

(iii) $\$ = a$, a terminal of arity n (say). Then J determines the traversal node a. Now the execution of $CloseTerm(J)$ calls $AddDer(\alpha_i, q_i)$ for each $i \in \{1, \cdots, n\}$, which creates n corresponding open judgements $J_i = \Gamma^o_1 \vdash t_i : q_i$ where t_i is in Lambda Form. Thus J has n successors, namely, $J_1, \cdots, J_n$.

Thus we have shown:

Theorem 5 (Correspondence)**.** *The justified judgement tree is isomorphic to the traversal tree.*

Remark 4. When formulating the correspondence, there is no need to consider the instantiation map because we are not concerned with types.