Higher-Order Model Checking with Traversals

Robin P. Neatherway

Merton College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

Trinity Term 2014

Abstract

Higher-order recursion schemes are a powerful model of functional computation that grew out of traditional recursive program schemes and generalisations of grammars. It is common to view recursion schemes as generators of possibly-infinite trees, which Ong showed to have a decidable monadic second order theory and opened the door to applications in verification. Kobayashi later presented an intersection type characterisation of the model checking problem, on which most subsequent applied work is based. In recent work, recursion schemes have been considered to play a role similar to Boolean programs in verification of first-order imperative programs: a natural target for abstraction of programs with very large or infinite data domains.

In this thesis we focus on the development of model checking algorithms for variants of recursion schemes. We start our contributions with a model checking algorithm inspired by the fully abstract game semantics of recursion schemes, but specified as a goal-directed approach to intersection type inference, that offers a unification of the views of Ong and Kobayashi. We build on this largely theoretical contribution with two orthogonal extensions and practical implementations.

First, we develop a new extension of recursion schemes: higher-order recursion schemes with cases, which add non-determinism and a case construct operating over a finite data domain. These additions provide us with a more natural and succinct target for abstraction from functional programs: encoding data using functions inevitably results in an increase in the order and arity of the scheme, which have a direct impact on the worst-case complexity of the problem. We characterise the model checking problem using a novel intersection and union type system and give a practical algorithm for type inference in this system. We have carried out an empirical evaluation of the implementation — the tool TravMC — using a variety of problem instances from the literature and a new suite of problem instances derived via an abstraction-refinement procedure from functional programs.

Second, we extend our approach from safety properties to all properties expressible in monadic second order logic using alternating parity tree automata as our specification language. We again provide an implementation and an empirical evaluation, which shows that despite the challenges accompanying liveness properties our tool scales beyond the current state of the art.

Acknowledgements

I would like to thank Luke Ong for giving me the opportunity to undertake a DPhil in the first place. I have learned many things that I would otherwise probably never have heard of, and for that I am very grateful. Our collaboration, along with Steven Ramsay, was very enjoyable and set me on the path that led to this dissertation.

My other office mates Christopher Broadbent, David Hopkins, Martin Lester, Michael Vanden Boom, Chang Yan, Emanuele D'Osualdo, Jonathan Kochems, Conrad Cotton-Barratt, Egor Ianovski and Marcelo Sousa all helped to create a welcoming and stimulating environment. I am lucky to have their friendship and wish them all the best of luck.

Daniel Kroening saw me through the transfer and confirmation process, and Dan Ghica joined him for the final examination. I am indebted to both for spending their limited time and for their useful comments and encouragement.

My Merton friends Stephen Buckley, Craig Lumb, Derek Hollman and Will Upcher were always ready, first to offer many distractions from the office and second to commiserate as I pushed towards the finish line.

The support of my family was of course invaluable. My parents Phil and Jackie for their constant support and optimism, my brother Laurie for many reasons (even though he wished I'd become a banker), and my wife Jean who encouraged me to start down this road and always believed I would reach the end.

Contents

Chapter 1

Introduction

1.1 Software Verification

Writing correct software is notoriously hard. We encounter computer systems — whether through explicit interaction or otherwise — in many aspects of everyday life, and the general public is almost inured to experiencing software defects in the course of these encounters. Errors in many pieces of software only cause frustration or delay, but errors in domains such as business software can be costly. Even during development, programmers spend around half their time fixing defects [Boehm et al., 2005] and a classic report by the US National Institute of Standards and Technology (NIST) in 2002 estimated the annual cost to the US economy of inadequate software testing as ranging from $22.2 to $59.5 billion [Tassey, 2002]. These figures do not include failures in so-called *mission critical* software that can lead to "loss of life or catastrophic failure." In many problem domains such as aerospace control systems and cryptographic libraries, the consequences of writing incorrect software can be extremely serious. The destruction of the first launch of the Ariane 5 rocket [Dowson, 1997] — caused by an unguarded conversion of an integer from 64-bit to 16-bit — is one case while more recent examples include security flaws in the Apple OS X and iOS SSL implementation [CVE-2014-1266] and the open-source GnuTLS library [CVE-2014-0092]. The latter two bugs both lay dormant for years prior to their discovery.

Minimising software defect rates is usually done using testing, and in more formal environments by also following software design methodologies such as *Correctness by Construction* [Hall and Chapman, 2002] and industry standards such as DO-178C [RTCA, 2011]. Common criteria for test suites mandate a certain percentage of coverage of lines, branches or paths for increasing level of assurance. The highest levels of assurance such as DO-178C Level A can mandate even stronger levels of coverage, in this case modified condition/decision coverage (known as

MC/DC). However, the cost of testing as a percentage of total development time is rising [Tassey, 2002], and in the end testing can give no formal guarantees. As Dijkstra [1972] said:

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

With the cast-iron guarantees that many verification techniques can give, and increasing recognition of formal methods in standards [Various, 2011] their use is becoming more and more attractive. Verification has traditionally been divided into two distinct areas: that based on logical inference (often interactive) and that based on model checking (usually automated).

Verification by inference

Techniques for verifying programs were first explored by Floyd [1967] in the 1960s, using a flowchart model of programs and logical inference to deduce properties of the programs at various points during execution. This was further developed by Hoare [1969], giving rise to Hoare logic. A Hoare triple $P\,\{Q\}\,R$ states that given a precondition P may be assumed to hold, following the execution of program Q postcondition R will hold. Coupled with a set of inference rules, Hoare triples describing different sections of a program may be composed to give a proof of partial (or even total) correctness. Dijkstra [1975] built on this foundation with his introduction of the notion of *weakest-precondition*. One can use his algorithm to infer the weakest P' that would have been required for R to hold after Q, reducing the problem of proving $P\,\{Q\}\,R$ to solving the first-order formula $P \implies P'$ (i.e. the original precondition was *at least* as strong as necessary). Weakest-preconditions are used in many automated and interactive tools for verifying programs today such as SPARK [Barnes, 2003] and Coq [Yves Bertot, 2004]. These tools can be used to prove full functional correctness (usually with respect to some formal specification), but it is more common to use them to rule certain common classes of error such as buffer overflows and unhandled exceptions. This approach is sometimes described as *extended static checking* and is embraced by the ESC/Java [Flanagan et al., 2002] and Spec# [Barnett et al., 2005] systems, which both generate verification conditions to be farmed out to an external automated theorem prover such as Simplify [Detlefs et al., 2005].

Model checking

Despite the limited success of inference-based methods of program verification, the early 1980s saw the arrival of a new method of formally analysing programs. Given

the difficulty of performing multiple proofs by hand and the undecidability of first-order logic preventing a fully automated inference engine, state space exploration methods were developed. Emerson and Clarke [1980] pioneered this work (with independent developments from Queille and Sifakis [1982]), motivated by the difficulty of analysing concurrent programs. The most general statement of the model checking problem is: given a formula ϕ in some logic, and some structure M, is M a model of ϕ? In early work, Kripke structures were often used with a temporal logic such as CTL or LTL used to specify properties of interest. Tools such as the SPIN model checker [Holzmann, 1997] based on automata-theoretic results due to Vardi and Wolper [1986] solve the model checking problem for finite-state systems. Here decidability is clear — a simple state-space enumeration would suffice — but performance is key.

Model checking was first widely adopted for verification of circuit designs [Burch et al., 1990; Beer et al., 1994], while in software development the first major industrial success was the Microsoft SLAM [Ball and Rajamani, 2002] project, which checks safety properties of C programs in the form of low level device drivers. Known as the Static Driver Verifier, the suite allowed specification of properties using a familiar 'C-like' syntax rather than directly as temporal logic formulae. The routine verification of correct acquisition and release of resources among other properties has been credited with a significant improvement in stability of Windows [Ball et al., 2004].

In order to handle systems with recursive procedures and corresponding possibly-infinite state spaces, as SLAM must do, symbolic representations of the state space are necessary. The SLAM backend model checker Bebop [Ball and Rajamani, 2000a] accepts recursive Boolean programs as input. In the automata-theoretic setting, the natural infinite-state system is a pushdown automaton, where the stack can be seen as an analogue of the call stack in a recursive program. The first algorithms for model checking of pushdown automata were given by Bouajjani et al. [1997].

Orthogonally, researchers have also expanded the logics we can use to specify richer properties. These include the μ-calculus [Kozen, 1983] and monadic second-order logic (MSO) as well as quantitative and probabilistic logics.

1.2 Functional Programming

Another way to attack the problem is to consider *how* we write software in the first place. The field of programming language research is rich with attempts to design new languages with more facilities for abstraction and fewer opportunities to introduce errors. A popular group of such languages is the *functional languages*. Development in this area started with ML [Milner, 1978], used as the metalanguage

in the University of Edinburgh's LCF proof system, which expanded into a family of related languages including SML, OCaml and F#. While the ML family allow the use of mutable state and references, Haskell [Peyton Jones et al., 2003] prizes referential transparency and *pure* computation. All these languages share a strong static type system based on the Damas-Milner type system [Milner, 1978; Damas and Milner, 1982], but this is by no means a requirement — Erlang and Clojure are dynamically typed. On the other hand all functional languages certainly have *first-class functions*. As opposed to traditional programming languages, where data and the code that operates on data are separated, in a functional language functions are not distinguished in this way and may be passed to other functions. Functions that accept other functions are said to be *higher-order*, with a function that operates just on integers, say, being *first-order*. A natural hierarchy then emerges, whereby a function that has as argument functions of at most order n is said to be order-$(n+1)$.

Perhaps the prototypical example of a higher-order function is `map`, given below in Haskell syntax, which applies a given function `f` to every member of a list.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Functional languages have been becoming more popular in recent years, particularly in the financial and scientific sectors [Minksy and Weeks, 2008; Leroy, 2007]. The clarity of numerical algorithms can often be improved with the additional expressive power and strong static type systems eliminate entire classes of errors, rendering programs that contain them illegal. Features such as anonymous functions have also been incorporated into more mainstream languages such as Java [Reinhold et al., 2014] and C# [ECMA International, 2006].

Functional programming languages have attracted interest as verification targets, just as imperative languages have. Up until recently, model checking has not played a large role in the verification of functional programs; existing techniques can be divided into two broad camps.

Types and static analysis

Type systems based on Damas-Milner are commonplace in functional languages and are often described as the most successful application of static analysis. The completeness of type inference in Damas-Milner allows the programmer to avoid inserting any type annotations at all, making the use of the type system extremely lightweight. Indeed, 'weak' static type systems such as that in C require *more* annotations while offering weaker guarantees. The type system of Haskell adds many features such as *generalised algebraic data types* [Peyton Jones et al., 2006] and

higher-rank types [Peyton Jones et al., 2007], which allow greater abstraction at the cost of inserting some type annotations.

Other lightweight automated static analyses include *control-flow analysis* as seen in the k-CFA framework due to Shivers [1991]. Simulating the execution of the program using an abstract machine allows for a fixed-point computation that yields an over-approximation of the reachable states of the program. CFA can be tuned by adjusting the abstract domain and the depth of call stack (k) used to distinguish program states. Flow analysis has been used for inference of type information in dynamically typed languages [An et al., 2011] and for lifting standard compiler optimisations to the higher-order setting. A line of work due to Jones and Andersen [2007] presents a related but more operational formulation based on grammars, while Sereni extended work on size-change termination to higher orders [Lee et al., 2001; Sereni, 2007].

Rondon et al. [2008] introduced *Logically Qualified Data Types* (Liquid Types), a system for verifying safety properties by inferring and checking a restricted form of refinement types based on user-provided predicates. The innovation lies in reducing the type checking problem to quantifier-free logical formulae, which can be fed to an off-the-shelf SMT solver. With some annotation, then, this allows programs to be checked for certain kinds of functional correctness and has applications to compiler optimisation, such as avoiding unnecessary run time checks. The latest version, for Haskell [Vazou et al., 2013], incorporates termination checking based on standard well-founded orderings.

Game semantics has been used to construct fully abstract models of programming languages, including famously providing a solution for PCF [Hyland and Ong, 2000]. This result has led to a model for Idealized Algol [Ong, 2002] and a series of tools for deciding observational equivalence from Hopkins and Ong [2009]; Hopkins et al. [2012] and checking safety properties from Bakewell and Ghica [2007]. Ghica and Bakewell [2009] also introduced a novel semantics-drivin technique for approximation of a program – *clipping* – which they have used as part of an iterative refinement procedure for verifying safety properties.

Theorem proving

In contrast to static analyses, which are typically regarded as being more lightweight (requiring less interaction and checking shallower properties), theorem proving can be used to check very sophisticated properties of programs, but often requires a correspondingly sophisticated user.

The Curry-Howard isomorphism, relating terms of the lambda calculus to proofs of intuitionistic logic, makes functional programming languages ideal for the construction of proof systems. Coq and the HOL family of interactive theorem provers, among many other projects, make use of this relationship. Coq contains a

strongly-normalising dependent functional programming language Gallina for building definitions and specifying algorithms. Proof development is done using tactics, both those built-in and those specified by the user in the tactic language Ltac. Coq has been used to formalise a proof of the four-colour theorem [Gonthier, 2007] and to construct CompCert [Leroy, 2009], a formally verified C compiler, by extracting OCaml code from the proof of correctness.

The Isabelle/HOL [Nipkow et al., 2002] system has been used to produce a proof of functional correctness of an OS microkernel [Klein et al., 2014] and HOL4 [Slind and Norrish, 2008] to develop a verified ML system, CakeML [Kumar et al., 2014].

Agda [Norell, 2009], on the other hand, leans more towards being a functional programming language extended with dependent types, although it is also described as a proof assistant. Rather than using tactics as in Coq, proofs are constructive and described through a functional program. Other recent developments in the field of dependent type systems include Idris [Brady, 2013] and F* [Swamy et al., 2013].

Theorem proving can also be applied in an automated fashion, more in the spirit of the static analyses described in Section 1.2. Terauchi [2010] gives a method for inferring dependent types sufficient to prove that assertions do not fail by using an automated theorem prover to refine the types in a counterexample-guided refinement loop. The work of Xu et al. [2009] allows a Haskell programmer to specify contracts on functions using Haskell itself. Proof obligations are generated by wrapping the functions in their contracts and performing symbolic execution, and then sent to an external automated theorem prover. More recent work on Haskell contracts due to Vytiniotis et al. [2013] uses the language's denotational semantics to convert the entire program and its contracts into a first-order formula, intended to be checked by a theorem prover.

1.3 Higher-Order Model Checking

Historically then, model checking has not had a strong place in verification of functional programs. However, a recent string of results around *higher-order recursion schemes* (HORS) promises to change this. HORS, as a kind of higher-order grammar, are a natural model of higher-order computation.

In the seventies and eighties, *recursive program schemes* were used as an approach to giving a formal semantics to programs [Courcelle and Nivat, 1978; de Roever and de Bakker, 1972]. These schemes used first-order functions and uninterpreted data to define a system of equations. These equations can then be interpreted similarly to a grammar, yielding a possibly-infinite tree representing program's behaviour syntactically. A second step, using a provided interpretation of symbols used in

the grammar can be used to find the meaning of the program. A critical insight was that for many properties, analysis of a program can be lifted to reasoning about the tree language generated by the scheme. Higher-order recursion schemes were introduced by Damm [1982] in the early eighties, offering a natural lifting of recursive program schemes to permit the use of higher-order functions in the equations.

From the automata-theoretic side, as previously mentioned the decidability of the model checking problem for pushdown automata is well established; in 2002 this was extended by Knapik et al. [2002] to all higher-order pushdown automata (where an order-n pushdown automaton has a stack alphabet of order-$(n-1)$ stacks). Unfortunately, the relationship between classical pushdown automata, recursive Boolean programs, and context free grammars does not extend cleanly to higher-order computation. An order-n pushdown automaton corresponds to an order-n *safe* HORS. In this context, the restriction of safety is a rather complex syntactic condition that guarantees that variable capture cannot occur during reduction.

The question of decidability of the MSO theories of trees generated by HORS was finally resolved directly by Ong [2006] and it was not until 2008 [Hague et al., 2008] that *collapsible pushdown automata* were defined and shown to be equi-expressive to unrestricted HORS, reestablishing the connection between grammars and automata.

In 2009, Kobayashi [2009a] showed the first practical application of higher-order model checking (to the *resource usage problem* as studied by Igarashi and Kobayashi [2005]) by encoding the behaviour of a functional program into the tree generated by a HORS. In the same paper he presented a characterisation of the HORS model checking problem as a type assignment problem in a particular intersection type system parameterised by the property. Informally speaking, in this way the infinite state space of the HORS is somehow quotiented with the respect to the property, yielding a finite set of *relevant* behaviours.

The characterisation as a type assignment problem opened the door to detailed algorithmic analysis. Techniques for abstraction also followed, using both novel and existing approaches such as predicate abstraction to apply HORS model checking to functional programs in an automated manner. Direct application to fragments of OCaml and Haskell have been demonstrated, verifying pattern-match safety and forms of partial correctness; we give more details in Section 2.3. Despite the difficulty of the problem, tools developed so far show encouraging empirical results. We find higher-order model checking at an exciting time!

1.4 Contributions and Outline

In this dissertation we are concerned with algorithms for solving the HORS model checking problem, and the application of such algorithms to verification of higher-order functional programs. The text is intended to be read in order and is structured as follows:

- In Chapter 2 we introduce the technical foundations of HORS and the automaton model we use for specifying properties. This allows us to formally introduce the statement of the HORS model checking problem and describe the intersection type approach to characterising the problem. The type system introduced here is used as the basis for most practical algorithms for HORS model checking, including our own. We give examples of typing derivations in the system to help build the reader's intuition. We also describe a number of applications of higher-order model checking in verification to show how this work may be applied in practice.

- The original proof of decidability of the HORS model checking problem used a notion of *traversal*, obtained from the fully abstract game semantics of HORS [Ong, 2006]. This was followed by an alternative proof using an intersection type system [Kobayashi, 2009a; Kobayashi and Ong, 2009]. Neither proof immediately gave rise to an efficient algorithm: both solutions suffered from the worst-case n-ExpTime behaviour for all inputs. Research into "practical" algorithms, which do not suffer from worst-case behaviour in all cases, followed with Kobayashi later extracting one such algorithm from the proof in [Kobayashi and Ong, 2009], implemented in the tool TRecS [Kobayashi, 2009b] and another, GTRecS, inspired by game semantics [Kobayashi, 2011]. In Chapter 3 we offer a new practical algorithm inspired by traversals, but grounded in intersection types, and extract from the algorithm insight into the relationship between the two. Previous work in the area has focused exclusively on one approach or the other. We believe that our work here helps to show that although type theory and game semantics are very different in flavour, when applied to HORS model checking, they are two ways of encoding the same information.

- Most practical applications of model checking use an abstraction step to over-approximate the input program (typically Turing-complete) to a simpler model with a decidable model checking problem. A classic example of such an abstract model in imperative model checking is *Boolean programs* [Ball and Rajamani, 2000b], which use standard procedural control flow — possibly with `goto` — along with a non-deterministic branch statement and Boolean variables. This use of non-determinism and abstraction of the program data

to a finite domain are standard in abstract models. In higher-order model checking, the abstract model is a HORS. In Chapter 4 we discuss how non-determinism and finite data can be encoded using higher-order functions and look at the deleterious effect of this encoding on the complexity of the problem. With this motivation we suggest a new abstract model: *higher-order recursion schemes with cases* (HORSC) and develop a novel intersection and union type system for characterising the HORSC model checking problem. We develop an extension of our algorithm from Chapter 3 and give an empirical evaluation of a tool implementing the algorithm. Our test set includes a demonstration of HORSC as an abstract model for a verification algorithm [Ong and Ramsay, 2011] and a comparison with other higher-order model checkers. The results show model checking HORSC to be faster than equivalent HORS, giving support to the theoretically motivated concerns about encoding data using higher-order functions that prompted us to develop HORSC.

- Ong [2006] proved decidability of the HORS model checking problem for properties specified using monadic second-order logic over trees (or equivalently μ-calculus or alternating parity tree automata). However, the majority of effort on development of model checking algorithms and tools has focused on *safety* properties (specified using deterministic tree automata where all states are accepting). More expressive *liveness* properties (including termination) are also of interest to the model checking community. In Chapter 5 we give an extension to our algorithm (orthogonal to that in Chapter 4) to allow specification of properties using unrestricted alternating parity tree automata. To achieve this it is necessary to handle both non-determinism in the automaton transition function and the more complex acceptance condition. Our algorithm continues to follow the traversal inspiration, constructing a single tree that represents the non-deterministic choices made by the automaton. Although acceptance in this general setting is, we avoid the algorithmic overhead with a novel termination condition. This differs from the existing competitor, TRecS-APT [Fujima et al., 2013], which solves a (possibly expensive) parity game after every round of its computation. We have implemented the extension in our tool, and give an encouraging comparison with TRecS-APT. Our tool is almost always faster, and in particular for the larger examples we consistently outperform TRecS-APT.

- Finally, in Chapter 6 we conclude with a summary of the results and a discussion of the possibilities for future work building on this dissertation.

Chapter 2

Recursion Schemes and Model Checking

In this chapter we will introduce the technical foundations on which our contributions are built. We start with an explanation of our choice of HORS as a model of higher-order functional computation. We will then formally define HORS and an automaton model for specifying properties of the generated trees; from these the definition of the model checking problem will follow. Finally we will explain the characterisation of the model checking problem as a type inference problem, which is central to our algorithmic approach.

2.1 A Model of Functional Programs

As outlined in Chapter 1, HORS are a lifting of recursive program schemes to the higher-order setting. Imagine that we wish to check reachability in a Haskell or ML program; a natural first step would be to apply an abstraction to the program and soundly reduce to a decidable problem. We will argue that HORS are a suitable model of higher-order computation (and thus target for such an abstraction), much as the popular formalism of Boolean programs are viewed as a suitable model for first-order computation.

Defunctionalisation [Reynolds, 1998] applies the insight that a program contains only finitely many syntactic λ-expressions, and so each can be represented by a tag. An apply function can then be defined over the set of tags, containing the logic required to invoke the function corresponding to the tag with the appropriate arguments. In this way partial applications can also be handled and the program is reduced to first-order. One example of using this technique to apply first-order reasoning to a higher-order program is in term rewriting [Arroyo et al., 2008]. It could also be used to reduce the verification of a Haskell program, say, to verification of a first-order program suitable for analysis by SLAM and similar

tools. However, the reduction tends to obscure the control flow of the original program, and confuses it with the data, requiring sophisticated reasoning to avoid too much imprecision during abstraction.

If we do not wish to reduce the verification to a first-order problem, then we must look for an appropriate abstraction target. HORS represent a smooth generalisation of finite state and pushdown model checking, in a manner we will make precise in the sequel. They allow accurate representation of higher-order functions from the original program: unlike simpler formalisms, only the data need be abstracted. HORS are also extremely expressive as generators of trees – they may contain infinitely many non-isomorphic subtrees. For these reasons we believe they are the most natural target for abstraction of higher-order programs. Here we will start by introducing HORS formally, but the reader will find a natural example of representing the behaviour of funtional program at the start of Section 2.3.

Syntax

HORS are strongly related to the λ-calculus, but of course the untyped λ-calculus allows one to define fixed-point combinators and is Turing powerful. In fact HORS may be thought of as terms of the simply-typed lambda calculus, equipped with recursion and first-order uninterpreted constants. Salvati and Walukiewicz [2011] exploit this similarity in their work, working directly with the λY-calculus. This restriction is critical for decidability of model checking. We typically use only a single base type, o, when typing HORS, but additional base types do not present a problem. For certain extensions that wish to talk about data directly, multiple base types can be useful.

The requirement for terms to be simply-typed guarantees some kind of finiteness of behaviours, and Tsukada and Kobayashi [2010] showed that a weaker typing in a intersection type system with finite width and depth is also sufficient. Interestingly this would also permit schemes typed using ML-style "let-polymorphism". Nevertheless, for this dissertation we restrict ourselves to the standard presentation with simple typing. In this work we refer to simple types as *kinds* in order to distinguish them from the *intersection types*, which will be crucial to our approach to the model checking problem.

Definition 2.1 (Kinds)**.** Given a set of base kinds $(b, o \in)\ \mathcal{B}$ the kinds over $\mathcal{B}$, $\mathbb{S}_{\mathcal{B}}$, are given by:

$$\kappa ::= o \mid \kappa_1 \to \kappa_2$$

As usual the *arity* (or *rank*) and *order* of kinds is:

$$\mathsf{arity}(o) = 0$$
$$\mathsf{arity}(\kappa_1 \to \kappa_2) = 1 + \mathsf{arity}(\kappa_1)$$

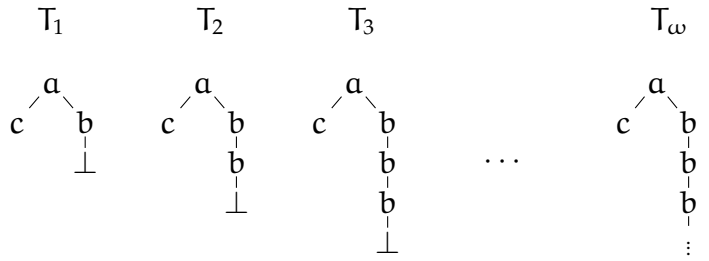


Figure 2.1: A family of labelled trees

and

$$\begin{aligned} \mathsf{order}(o) &= 0 \\ \mathsf{order}(\kappa_1 \to \kappa_2) &= \mathsf{max}(1 + \mathsf{order}(\kappa_1), \mathsf{order}(\kappa_2)) \end{aligned}$$

A base kind (defined without the arrow constructor) is also said to be *ground*. Naturally, a system of kinds will not be much use without another system of terms to apply it to. In order for our terms to define trees, we first need some notion of terminal symbols or tree constructors.

Definition 2.2 (Ranked alphabets)**.** A *ranked alphabet* is a map from some finite set of constants to natural numbers. Given a ranked alphabet $(a, b, c \in)\ \Sigma$, $\Sigma(a)$ is the *arity* of a. We will often consider an extension of Σ to include a special distinguished symbol $\bot \notin \mathsf{dom}(\Sigma)$, so that $\Sigma^\bot = \Sigma \cup \{\bot \mapsto 0\}$.

Definition 2.3 (Σ-labelled trees)**.** A *tree* is a prefix-closed subset of $\mathbb{N}^*$, and a Σ-*labelled tree* is a function $T : \mathsf{dom}(T) \longrightarrow \Sigma$ where $\mathsf{dom}(T)$ is a tree. We define a partial order on $\Sigma^\bot$-labelled trees such that $T_1 \sqsubseteq T_2$ just if $\mathsf{dom}(T_1) \subseteq \mathsf{dom}(T_2)$ and $\forall \pi \in \mathsf{dom}(T_1) \cdot T_1(\pi) = T_2(\pi) \vee T_1(\pi) = \bot$. Given a set S of $\Sigma^\bot$-labelled trees, we write $\bigsqcup S$ for the least upper bound of S with respect to $\sqsubseteq$, if it exists.

Example 2.4 (Upper bound of partially defined trees)**.** Take the ranked alphabet $\Sigma_1 = \{a \mapsto 2, b \mapsto 1, c \mapsto 0\}$. In Figure 2.1 is a family of Σ_1-labelled trees with a growing path of b's as the right-hand branch. Each T_{i+1} is more defined than T_i in a precise sense, for example $\mathsf{dom}(T_1) = \{\varepsilon, 1, 2, 21\} \subseteq \mathsf{dom}(T_2) = \{\varepsilon, 1, 2, 21, 211\}$. The upper bound of this family $\bigsqcup\{T_1, T_2, \cdots\}$ is given by the tree T_ω on the right, which contains the path $a\ b^\omega$.

Definition 2.5 (Terms)**.** Given finite sets of variables $(x, y, z \in)\ \mathcal{V}$, function symbols $(F, G, H \in)\ \mathcal{F}$ and a ranked alphabet $(a, b, c \in)\ \Sigma$, terms are defined by the following grammar:

$$s, t ::= a \mid x \mid F \mid s\ t \mid \lambda x.t$$

Terms defined without λ-abstraction are *applicative*, while terms having a sequence of abstractions outermost such as $\lambda x_1 \cdots x_n.t$ are often written using the shorthand $\lambda\widetilde{x}.t$. As normal the size of a term t, $|t|$ is given by:

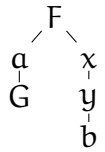
$$\begin{aligned} |a| &= 1 \\ |F| &= 1 \\ |x| &= 1 \\ |s\ t| &= 1 + |s| + |t| \\ |\lambda x.t| &= 1 + |t| \end{aligned}$$

We also require a notion of *free variables*, which are those not bound by any enclosing lambda and defined by a function FV:

$$\begin{aligned} \mathsf{FV}(a) &= \emptyset \\ \mathsf{FV}(F) &= \emptyset \\ \mathsf{FV}(x) &= \{x\} \\ \mathsf{FV}(s\ t) &= \mathsf{FV}(s) \cup \mathsf{FV}(t) \\ \mathsf{FV}(\lambda x.t) &= \mathsf{FV}(t) \setminus \{x\} \end{aligned}$$

If $\mathsf{FV}(t) = \emptyset$ then t is said to be *closed*.

We observe at this point that there is a natural notion of a term tree. For example, given the applicative term $s = F\ (a\ G)\ (x\ (y\ b))$ we may also view this as a labelled tree:



Definition 2.6 (Subterm)**.** Given a term t, we may wish to talk about a subterm of t. Given a path w (that is a node in the corresponding term tree), $t|_w$ denotes the subterm rooted at position w, so that $t|_\varepsilon = t$. By example using the term s above, $s|_1 = a\ G$ and $s|_{2\,2} = y\ b$.

Definition 2.7 (Kind assignment)**.** A *kind environment* is a partial function from some set of symbols to kinds, typically written $\{x_1 : \kappa_1, \cdots, x_n : \kappa_n\}$. We often write $\Gamma, x : \kappa$ for $\Gamma \uplus \{x : \kappa\}$ (when $x \notin \mathsf{dom}(\Gamma)$). Assignment of kinds to terms in the context of some set of constants Σ is achieved using the standard simple type system shown in Table 2.1. A term t may be assigned kind κ just if for some kind environment Γ there is a legal derivation rooted at the judgement $\Gamma \vdash t : \kappa$. A kinded term $\Gamma \vdash t : \kappa$ takes the order and arity of its kind, so that $\mathsf{order}(t) = \mathsf{order}(\kappa)$

$$\frac{\Sigma(a) = n}{\Gamma \vdash a : \underbrace{o \to \cdots \to o}_{n} \to o}\ (\text{K-Cst}) \qquad \frac{}{\Gamma, x : \kappa \vdash x : \kappa}\ (\text{K-Var})$$

$$\frac{\Gamma \vdash s : \kappa' \to \kappa \qquad \Gamma \vdash t : \kappa'}{\Gamma \vdash s\ t : \kappa}\ (\text{K-App}) \qquad \frac{\Gamma, x : \kappa \vdash \lambda x.t : \kappa' \to \kappa}{\Gamma \vdash \lambda x.t : \kappa' \to \kappa}\ (\text{K-Abs})$$

Table 2.1: Kind assignment system

and $\mathsf{arity}(t) = \mathsf{arity}(\kappa)$. Note that this system of kind assignment ensures that our notions of arity for ranked alphabets, kind environments and terms coincide i.e. if the terminal a has $\Sigma(a) = n$ then necessarily the term a has $\mathsf{arity}(a) = n$. As a result we identify ranked alphabets with kind environments containing kinds of order at most one.

Definition 2.8 (Higher-order recursion schemes)**.** A (deterministic) *higher-order recursion scheme* $\mathcal{G}$ is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where:

1. Σ is a ranked alphabet of terminal symbols;
2. $\mathcal{N}$ is a kind environment of non-terminal symbols;
3. $\mathcal{R}$ is a function from non-terminals $\mathsf{dom}(\mathcal{N})$ to closed terms of the form $\lambda x_1 \cdots x_n.t$ where t is an applicative term of ground kind. When convenient, we may also write a member of $\mathcal{R}$ using an alternative syntax – $F\, x_1 \cdots x_n = t$. $\mathcal{R}$ satisfies a well-kindedness condition so that for any $F \in \mathsf{dom}(\mathcal{N})$, $\mathcal{N} \vdash \mathcal{R}(F) : \mathcal{N}(F)$; and
4. S is a distinguished start symbol where $\mathcal{N}(S) = o$.

A few observations about this definition:

(i) The restriction that a rewrite rule $F\, x_1 \cdots x_n = t$ maps to a term t of kind o affords us some technical convenience, but note that we can always η-expand to reach this form.

(ii) We hope that at this point a reader familiar with Haskell or another functional programming language will notice the similarity between function definitions and HORS rewrite rules. A critical restriction in expressivity is the lack of pattern-matching, although higher-order functions still allow the expression of complex control flows.

$$\dfrac{\dfrac{\Sigma(c) = 0}{\mathcal{N} \vdash c : o} \quad \dfrac{}{\mathcal{N} \vdash F : o \to o}}{\mathcal{N} \vdash F\,c : o}$$

$$\dfrac{\dfrac{\dfrac{\Sigma(a) = 2}{\Gamma \vdash a : o \to o \to o} \quad \dfrac{}{\Gamma \vdash x : o}}{\Gamma \vdash a\,x : o \to o} \quad \dfrac{\dfrac{}{\Gamma \vdash F : o \to o} \quad \dfrac{\dfrac{\Sigma(b) = 1}{\Gamma \vdash b : o \to o} \quad \dfrac{}{\Gamma \vdash x : o}}{\Gamma \vdash b\,x : o}}{\Gamma \vdash F\,(b\,x) : o}}{\Gamma \vdash a\,x\,(F\,(b\,x)) : o}}{\mathcal{N} \vdash \lambda x.a\,x\,(F\,(b\,x)) : o \to o}$$

Table 2.2: Kinding derivations for $\mathcal{G}_1$ ($\Gamma = \mathcal{N} \uplus \{x : o\}$)

(iii) The kinds of the variables in a rewrite rule are determined entirely by the kind of the non-terminal. So if $F : \kappa_1 \to \cdots \to \kappa_n \to \kappa \in \mathcal{N}$ then necessarily $\mathcal{R}(F) = \lambda x_1 \cdots x_n.t$ and for all $i \in [1..n]$, $x_i : \kappa_i$.

(iv) The order of a recursion scheme is the largest order of any nonterminal symbol, $\max\{\mathsf{order}(\kappa) \mid F : \kappa \in \mathcal{N}\}$.

With this definition in mind, the reader is invited to recall the definitions of regular and context-free tree grammars [Rounds, 1969; Engelfriet and Schmidt, 1977]. If we consider order-0 HORS, all non-terminals must have kind o. As a result they cannot have any arguments and all rewrite rules are exactly regular tree grammar productions. In the order-1 case, each non-terminal will have kind of the form $o \to \cdots \to o \to o$ (the arguments must be ground terms). Here we find ourselves with the productions of a context-free tree grammar, where during reduction the variables are always substituted with concrete trees. The relationship between finite automata and regular grammars on the one hand and pushdown automata and context-free grammars on the other is well known, and so we have a correspondence between the first two levels of the HORS hierarchy and these classical long-standing structures.

Example 2.9 (Recursion scheme $\mathcal{G}_1$)**.** We define a recursion scheme $\mathcal{G}_1$, often used in the literature, using terminal symbols a, b and c of arities 2, 1 and 0 respectively and non-terminal symbols $\mathcal{N} = \{S : o, F : o \to o\}$. The rewrite rules are:

$$S = F\,c$$
$$F = \lambda x.a\,x\,(F\,(b\,x))$$

which are well-kinded under $\mathcal{N}$ as witnessed by the derivations in Table 2.2.

Semantics

Definition 2.10 (Reduction)**.** Recursion schemes have a notion of reduction defined in terms of the rewrite rules $\mathcal{R}$:

$$\frac{\mathcal{R}(F) = \lambda x_1 \cdots x_n.e}{F\ t_1\ \cdots t_n \to_{\mathcal{G}} e[t_1/x_1, \cdots, t_n/x_n]} \qquad \frac{t \to_{\mathcal{G}} t'}{s\ t \to_{\mathcal{G}} s\ t'} \qquad \frac{t \to_{\mathcal{G}} t'}{t\ s \to_{\mathcal{G}} t'\ s}$$

We write $\to_{\mathcal{G}}^*$ for the reflexive, transitive closure of $\to_{\mathcal{G}}$ and typically elide the subscript where the scheme in question is clear. The reduction as defined here may take place in arbitrary contexts thanks to the two right-hand rules.

There are three modes of reduction considered by Damm [1982]: outermost-innermost (OI), innermost-outermost (IO) and unrestricted. The scheme $\mathcal{G}_1$ has only a single possible reduction sequence from the start symbol S:

$$S \to \underline{F\ c} \to a\ c\ (\underline{F\ (b\ c)}) \to a\ c\ (a\ (b\ c)\ (\underline{F\ (b\ (b\ c))})) \to \cdots$$

thus generating the term $a\ c\ (a\ (b\ c)\ (a\ (b\ (b\ c)))\ (\cdots))$.

Example 2.11 (An order-2 recursion scheme $\mathcal{G}_2$)**.** We will introduce a second example in order to make clear difference the between the reduction strategies. We use again terminal symbols a and b of arities 2 and 1 respectively with non-terminal symbols $\mathcal{N} = \{S : o, F : o \to o, G : (o \to o) \to o, H : o \to o\}$ and rewrite rules:

$$\begin{aligned} S &= F\ (G\ b) \\ F &= \lambda x.a\ (H\ x)\ x \\ G &= \lambda x.b\ (G\ x) \\ H &= \lambda x.H\ x \end{aligned}$$

This HORS admits many reduction sequences. If we adopt the OI strategy one such sequence is:

$$\begin{aligned} S &\to \underline{F\ (G\ b)} \\ &\to a\ (H\ (G\ b))\ (\underline{G\ b}) \\ &\to a\ (H\ (G\ b))\ (b\ (\underline{G\ b})) \\ &\to a\ (H\ (G\ b))\ (b\ (b\ (\underline{G\ b}))) \to \cdots \end{aligned}$$

while under IO we see:

$$\begin{aligned} S &\to F\ (\underline{G\ b}) \\ &\to F\ (b\ (\underline{G\ b})) \\ &\to F\ (b\ (b\ (\underline{G\ b}))) \to \cdots \end{aligned}$$

An example of an unrestricted reduction sequence (where after the first two steps we make no further progress):

$$\begin{aligned} S &\to F\,(\underline{G\ b}) \\ &\to \underline{F\,(b\ (G\ b))} \\ &\to a\ (\underline{H\ (G\ b)})\ (b\ (G\ b)) \\ &\to a\ (\underline{H\ (G\ b)})\ (b\ (G\ b)) \\ &\to a\ (\underline{H\ (G\ b)})\ (b\ (G\ b)) \to \cdots \end{aligned}$$

Damm showed that the OI and unrestricted strategies are equivalent in the sense any normal form reachable an unrestricted reduction sequence is also reachable via an OI reduction sequence. This is clearly not the case for IO, however.

Haddad [2012] probed the difference between the strategies and gave a pair of transformations such that given a scheme $\mathcal{G}$ that produces trees $[\![\mathcal{G}]\!]_{OI}$ and $[\![\mathcal{G}]\!]_{IO}$ according to the two strategies, the images of the transformations (schemes $\mathcal{G}_{OI}$ and $\mathcal{G}_{IO}$) will produce the corresponding tree no matter what reduction strategy is used i.e. $[\![\mathcal{G}]\!]_{OI} = [\![\mathcal{G}_{OI}]\!]$ and $[\![\mathcal{G}]\!]_{IO} = [\![\mathcal{G}_{IO}]\!]$. Note that $(\cdot)_{OI}$ increases the order of the scheme by one, whereas $(\cdot)_{IO}$ produces a scheme of the same order. Kobele and Salvati [2013] also explored the hierarchy (in terms of type-theoretic order) induced by each strategy.

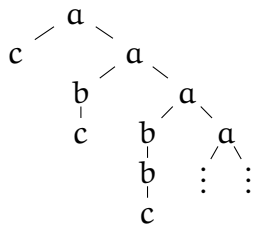
However, in this dissertation, as in most of the recent work on HORS, we consider reduction to be unrestricted. There is a clear, albeit loose, connection between the OI and IO strategies and the call-by-name and call-by-value reduction strategies used in programming languages. The construction of a formal connection to call-by-value requires a notion of values as seen in Tsukada and Kobayashi's [2014] work.

Definition 2.12 (Value tree of a recursion scheme)**.** We view recursion schemes as generators of trees, and intuitively we can see the *value tree* as a kind of limit of $\to_{\mathcal{G}}$. The value tree of a recursion scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, $[\![\mathcal{G}]\!]$, is a $\Sigma^{\perp}$-labelled tree defined in terms of a Σ-preserving function on closed applicative terms:

$$\begin{aligned} (a\ t_1 \cdots\ t_n)^{\perp} &= a\ t_1^{\perp} \cdots t_n^{\perp} \\ (F\ t_1 \cdots\ t_n)^{\perp} &= \perp \end{aligned}$$

Then $[\![\mathcal{G}]\!] = \bigsqcup\{t^{\perp} \mid S \to_{\mathcal{G}}^{*} t\}$. Note that the confluence of $\to_{\mathcal{G}}$ guarantees the existence of the least upper bound in this case and hence $[\![\mathcal{G}]\!]$ is well defined.

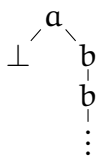
Example 2.13 (Value tree of $\mathcal{G}_1$)**.** Recall the recursion scheme from Example 2.9. The value tree, $[\![\mathcal{G}_1]\!]$, can be seen below.



As expected, the definition yields an infinite tree labelled by a, b and c. The tree consists of an infinite spine of 'a's with each branch containing an increasing number of 'b's terminated by a 'c'. Note that even this relatively simple recursion scheme has infinitely many non-isomorphic subtrees. Furthermore, $[\![\mathcal{G}_1]\!]$ does not contain $\perp$ anywhere, as every sequence of reductions eventually leads to a terminal symbol appearing in head position. For $\mathcal{G}_1$ this is clear as the only non-terminal used recursively, F, has a rewrite rule with the terminal a in head position.

Definition 2.14 (Productivity)**.** Recursion schemes that, like $\mathcal{G}_1$, have a value tree not containing $\perp$ are called *productive*. At times in this work we will restrict ourselves to considering productive schemes if we wish to avoid prematurely choosing a semantics for divergence. In the case of safety properties it seems natural to accept any subtree equal to $\perp$ and we will make this explicit. In more complex cases it is not so clear-cut. Note that we can restrict ourselves to productive schemes without loss of generality as we can always insert fresh terminal symbols outermost on the right-hand side of each rewrite rule to make divergent behaviour explicit.

Example 2.15 (Value tree of $\mathcal{G}_2$)**.** Our second example scheme, on the other hand, is not productive. The rewrite rule $H = \lambda x.H\ x$ will never produce a terminal symbol, and so this path of $[\![\mathcal{G}_2]\!]$ will be $\perp$:



2.2 Decidability of Model Checking

We have defined HORS, and the means by which they generate trees. In order to define a model checking problem, we must have a means of specifying properties to check. Over trees, the modal μ-calculus [Kozen, 1983] and monadic second-order logic are known [Emerson and Jutla, 1991] to be equi-expressive with *alternating parity tree automata*. Throughout this work, we will use the latter formalism of alternating parity automata for properties. There are several equivalent formulations

of such automata; here we follow Kupferman et al. [2000]. Rather than an explicit *alternation* between universal (or conjunctive) and existential (or non-deterministic) transitions, a Boolean formula allows a more concise specification.

Alternating parity tree automata

Given a finite set X, the set $B^+(X)$ of *positive Boolean formulas* over X is defined by the grammar

$$B^+(X) \ni \theta ::= t \mid f \mid x \mid \theta \wedge \theta \mid \theta \vee \theta$$

where x ranges over X. We say that a subset Y of X *satisfies* θ if assigning true to elements in Y and false to elements in $X \setminus Y$ makes θ true. It follows that t is satisfied by all subsets of X, and f by none. Since $\theta \in B^+(X)$ is positive, if a set Y satisfies θ, so does every superset of Y.

Definition 2.16 (Alternating parity tree automata)**.** An *alternating parity tree automaton* (or APT for short) over Σ-labelled trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \Omega \rangle$ where:

(i) $(a, b, c \in)\ \Sigma$ is a ranked alphabet;

(ii) Q is a finite set of states;

(iii) $\delta : Q \times \Sigma \longrightarrow B^+(\mathbb{N} \times Q)$ is the transition function, where for each $q \in Q$ and $a \in \Sigma$, $\delta(q, a) \in B^+([1..\mathsf{arity}(a)] \times Q)$;

(iv) $q_0 \in Q$ is the initial state; and

(v) $\Omega : Q \longrightarrow \mathbb{N}$ is the priority function.

For any pair (q, a), $\delta(q, a)$ can be thought of as a set of satisfying assignments (explicitly so through conversion to DNF), each of which are a potential continuation of the run-tree. A satisfying assignment consists of a set of pairs (i, q') ($i \in [1..\mathsf{arity}(a)]$, $q' \in Q$) where each such pair corresponds to sending a copy of the automaton in state q' to child i of the current position in the tree.

Definition 2.17. (Run tree) Given a Σ-labelled tree T (a possible member of the language of $\mathcal{A}$), we again follow Kupferman et al. [2000] and define the *run-tree* of $\mathcal{A}$ over T as the $(\mathsf{dom}(T) \times Q)$-labelled (unranked) tree R satisfying

(i) The root $R(\varepsilon) = (\varepsilon, q_0)$.

(ii) For every $\beta \in \mathsf{dom}(R)$ with $R(\beta) = (\alpha, q)$, there is a (possibly empty) set S that satisfies $\delta(q, T(\alpha))$; and for each $(i, q') \in S$, there is some j such that $\beta j \in \mathsf{dom}(R)$ and $R(\beta j) = (\alpha i, q')$.

A run-tree R is *accepting* just if on every infinite path of R, the maximum priority of the states that occur infinitely often is even. A Σ-labelled tree T is *accepted* by an automaton $\mathcal{A}$ just if there is an accepting run-tree of $\mathcal{A}$ over T. The *tree language* of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the smallest set of Σ-labelled trees that are all accepted by $\mathcal{A}$.

There are a number of interesting restricted forms of APT. We say that an APT is *deterministic* just if its transition function δ is deterministic i.e. for each $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is t or f or has the shape $\bigwedge_{i=1}^{\mathsf{arity}(a)}(i, q_i)$, typically written $q_1 \cdots q_n$. It is *conjunctive* just if δ maps every pair to a disjunction-free formula. Finally, it is *disjunctive* just if δ maps every pair to a conjunction-free formula.

Alternating Büchi tree automata An *alternating Büchi tree automaton* (ABT) simply limits the codomain of Ω to $\{1, 2\}$, to yield the classical Büchi acceptance condition.

Alternating weak tree automata Equi-expressive with *weak* MSO and *alternation-free* μ-calculus, there is a notion of *alternating weak tree automata* (AWT) [Muller et al., 1992]. An AWT is an ABT that satisfies *weakness*: there is a partial order $\leqslant$ over a partition $\{ Q_1, \cdots, Q_n \}$ of Q such that

(i) for each i, every state in Q_i has the same priority; and

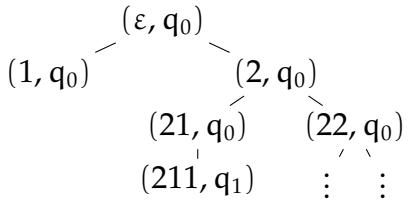
(ii) for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\delta(q, a)$, for some $a \in \Sigma$, we have $Q_j \leqslant Q_i$.

It follows that every infinite path of an run-tree of an AWT ultimately gets trapped within some Q_j. A *deterministic weak tree automaton* (DWT) is an AWT that is deterministic.

Trivial automata A *trivial* (resp. *co-trivial*) automaton has only one priority: 2 (resp. 1). Thus a tree is accepted by a co-trivial automaton if (and only if) there is a run-tree that has no infinite paths. We abbreviate alternating trivial automata as ATT and deterministic trivial automata as DTT. A trivial automaton is written as a quadruple, eliding the priority function. Trivial automata specify *safety* properties, where rejection must be witnessed by a finite trace. It is consistent to allow divergent behaviour when checking safety properties, and so given a trivial automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ where $\bot \notin \Sigma$ we define $\mathcal{A}^\bot = \langle \Sigma^\bot, Q, \delta \uplus \{(q, \bot) \mapsto \varepsilon\}, q_0 \rangle$.

Example 2.18 (A DTT $\mathcal{A}_1$)**.** Take the ranked alphabet Σ of Example 2.9; $[\![\mathcal{G}_1]\!]$ is accepted by $\mathcal{A}_1 = \langle \Sigma, \{q_0, q_1\}, \delta, q_0 \rangle$, where $\delta = \{(q_0, a) \mapsto q_0 q_0, (q_0, b) \mapsto q_1, (q_1, b) \mapsto q_1, (q_1, c) \mapsto \varepsilon, (q_0, c) \mapsto \varepsilon\}$. Thus $\mathcal{A}_1$ accepts a Σ-labelled tree t if, and only if, b

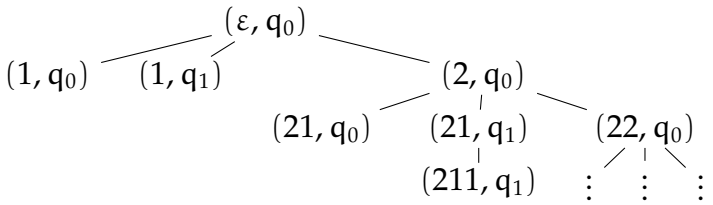
and c are seen only after an a. A prefix of the run-tree is included below. In the case of deterministic automata, like this one, the run-tree has exactly the same shape as the value tree.



Example 2.19 (An APT $\mathcal{A}_2$)**.** We define $\mathcal{A}_2 = \langle \Sigma, \{q_0, q_1\}, \delta_2, q_0, \Omega \rangle$, with the following transition and priority functions:

$$\begin{aligned}
&\delta_2(q_0, a) = (1, q_0) \wedge (2, q_0) \wedge ((1, q_1) \vee (2, q_1)) \\
&\delta_2(q_1, a) = (1, q_1) \wedge (2, q_1) && \Omega(q_0) = 0 \\
&\delta_2(q_0, b) = \varepsilon && \Omega(q_1) = 1 \\
&\delta_2(q_1, b) = (1, q_1) \\
&\delta_2(q_1, c) = \varepsilon
\end{aligned}$$

State q_0 has an even priority, while q_1 has an odd priority, so all infinite paths in the run-tree must not contain q_1 infinitely often. Hence the automaton accepts trees where each terminal a has either its left or right child being the root of a finite tree. The conjunction with $(1, q_0) \wedge (2, q_0)$ ensures that this check occurs at every node in the tree. We know that in the tree $[\![\mathcal{G}_1]\!]$ every left branch from a is a word in b^*c, and so this property will hold. A prefix of the run-tree can be seen below. The only infinite path is the spine, corresponding to the spine of a's in the value tree. As each node in the spine is labelled with q_0, which has an even priority, this is an accepting run-tree.



Definition 2.20 (Complementation)**.** Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \Omega \rangle$ be an APT. Given a

Boolean formula ϕ, we define the de Morgan dual of that formula ϕ^c by:

$$\begin{aligned} x^c &= x \\ t^c &= f \\ f^c &= t \\ (\phi_1 \wedge \phi_2)^c &= \phi_1^c \vee \phi_2^c \\ (\phi_1 \vee \phi_2)^c &= \phi_1^c \wedge \phi_2^c \end{aligned}$$

and the dual automaton $\mathcal{A}^c = \langle \Sigma, Q, \delta^c, q_0, \Omega^c \rangle$ where $\delta^c(q, a) = (\delta(q, a))^c$ and $\Omega^c(q) = \Omega(q) + 1$.

We also write the complement of the language $\mathcal{L}(\mathcal{A})$ as $(\mathcal{L}(\mathcal{A}))^c$ and define it simply as all those Σ-labelled trees not accepted by $\mathcal{A}$. It is a theorem due to Muller and Schupp [1987] that $\mathcal{L}(\mathcal{A}^c) = (\mathcal{L}(\mathcal{A}))^c$.

Parity Games

A *parity game* [Grädel et al., 2002] is a determined two player game $\mathbb{G} = \langle V_\forall, V_\exists, v_0, E, \Omega \rangle$ where:

(i) $V = V_\forall \uplus V_\exists$ is a set of vertices.

(ii) $v_0 \in V$ is the initial vertex.

(iii) $E \subseteq V \times V$ is the edge relation of a directed graph.

(iv) $\Omega : V \longrightarrow \mathbb{N}$ is a priority function.

A *play* in a parity game consists in the players, Éloïse and Abelard, taking turns to move a token along the edges of the graph starting from v_0. At any point in a play, if $v \in V_\exists$ (respectively $v \in V_\forall$) then Éloïse (respectively Abelard) chooses an edge $(v, v') \in E$ and moves the token to v'. A play is a finite or infinite path $\pi = v_0 v_{n_1} v_{n_2} \cdots$ in the graph. If a play π is maximal i.e. it is infinite or it ends in a vertex v such that there is no $(v, v') \in E$ then the winner is determined by:

- If π is finite and the final vertex $v \in V_\exists$ (respectively $v \in V_\forall$) then the winner is Abelard (respectively Éloïse).
- If π is infinite, then the Éloïse wins just if the maximal priority seen infinitely often in the sequence $\Omega(v_0)\Omega(v_{n_1})\Omega(v_{n_2}) \cdots$ is even. If it is odd then Abelard wins.

A memoryless strategy for Éloïse (respectively Abelard) is a function $\sigma : V_\exists \longrightarrow V$ (respectively $\sigma : V_\forall \longrightarrow V$). A strategy for a player is winning if every play that conforms to σ (but where the other player is not restricted) is winning for that player. Finally, we say that a parity game is winning for Éloïse if, and only if, there exists a winning strategy for Éloïse.

The Model Checking Problem

In this work we consider variants of the following problem:

Definition 2.21 (Higher-order recursion scheme model checking problem)**.** Given a higher-order recursion scheme $\mathcal{G}$ and an alternating parity tree automaton $\mathcal{A}$, is the tree generated by $\mathcal{G}$, $[\![\mathcal{G}]\!]$, accepted by $\mathcal{A}$?

We are interested in certain restrictions of the problem where the property automaton is drawn from a restricted class of APT. When, for example, the class in question is DTT, we refer to the restricted case as the "HORS/DTT" model checking problem.

Theorem 2.1 (Decidability)**.** *The HORS/APT model checking problem is* n-ExpTime *complete, where* n *is the order of the HORS.*

The complexity class n-ExpTime contains those problems solvable in the worst case in time bounded by a tower of exponentials of height n. More formally, $\mathcal{O}(\mathsf{exp}_n(p(m)))$ where p is some polynomial in the size of the problem instance m:

$$\mathsf{exp}_0(x) = x \qquad\qquad \mathsf{exp}_{i+1}(x) = 2^{\mathsf{exp}_i(x)}$$

The original proof based on game semantics is due to Ong [2006]. This technique, however, did not yield an efficient algorithm immediately. An attempt to solve the model checking problem naïvely via this technique would suffer from the worst-case behaviour in *every* case.

Although at first glance a problem in this complexity class would be considered to be intractable, there is reason to believe that it arises only from the hyper-exponentially compact representation of programs possible with higher-order functions. In our experience, functional programmers do not tend to write programs that use the full hyper-exponential succinctness of higher-order functions. A well-known possible analogue is the DExpTime complete complexity of Damas-Milner type inference, which does not arise in practice.

Intersection Types

A practical approach to the problem was not found until 2009 when Kobayashi [2009a,b] presented an approach based on intersection types that allowed one to avoid the punishing worst-case complexity in many cases. The original work and practical algorithm restricted the property automaton to a DTT; the underlying theory was subsequently generalised in joint work between Kobayashi and Ong [2009] to full APT.

Intersection types were introduced by Coppo and Dezani-Ciancaglini [1978] to address problems with the simply typed lambda calculus of Curry, which did not assign types to many seemingly well-behaved terms. For example, the fixed-point combinators, and terms with self-application such as $\lambda x.x\ x$ cannot be assigned a simple type. The intersection types were found to characterise all those terms which are strongly normalising (later strengthened to weakly normalising), a powerful result. However, the problem of type inference was also found to be undecidable for such a system. By introducing the intersection operator as well as the arrow (seen in our kind system in Table 2.1), the type system can assert that a term can behave in multiple ways. As an example, the term $t = \lambda x.x\ x$ can be assigned the intersection type $((o \to o) \wedge o) \to o$, which expresses that any term that t is applied to must be able to behave as both a function (of type $o \to o$) and an argument to that function (of type o).

Intersection types have been used both in the theoretical sphere such as research arising out of Barendregt et al.'s work on filter models [Barendregt et al., 1983] and also in applied research. Examples of the latter include the $\mathbb{C}$Duce language for XML processing [Benzaken et al., 2003] and a type inference approach for Ruby [An et al., 2011]. The relationship between intersection types and *duck typing* in Ruby and other dynamic languages can be constructed using a system with a base type for each syntactic method in the program; a class can be assigned the type that is an intersection of the methods it implements.

In the HORS world, Kobayashi's key insight was the introduction of a type system parameterised by the property automaton such that a given base type describes terms that generate trees accepted from the corresponding automaton state. His type system also only permits intersection types that refine the existing kindings associated with the HORS, which results in a finite space of intersection types. We present here the original type system for characterising the HORS/DTT model checking problem. As we previously mentioned, for properties specified using trivial automata, we wish to accept any divergent path and so the statement of the problem is slightly different:

Definition 2.22 (HORS/DTT model checking problem)**.** Given a HORS $\mathcal{G}$ and a DTT $\mathcal{A}$, is $[\![\mathcal{G}]\!]$ accepted by $\mathcal{A}^{\perp}$?

Well-Kinded Types

We have reserved *type* until now for the system of intersection types that follow. Fix a DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$. 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:

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

Any intersection type σ such that $\sigma :: \kappa$ is derivable in the above system is a well-kinded type. For example, given $Q = \{q_0, q_1\}$, $q_1 \to q_0$ and $((q_1 \to q_1) \wedge (q_0 \to q_0)) \to q_0$ are *well-kinded types* (respectively of kind $o \to o$ and $(o \to o) \to o$) 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 set of well-kinded types, and Type_κ for those types that refine κ. 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 $\bigwedge \emptyset$. Note that for technical convenience intersection is only allowed on the left of an arrow and so we can define a function state that returns the rightmost element of a type:

$$\mathsf{state}(\theta) = \begin{cases} q & \text{if } \theta = q \\ \mathsf{state}(\theta_2) & \text{if } \theta = \theta_1 \to \theta_2 \end{cases}$$

Type System

Intuitively, a typing for a term t describes the tree generated by t. For example, the typing $c : q_0$ indicates that the trivial tree c is accepted from state q_0. This extends to higher orders so that $\lambda x.s : (q_0 \wedge q_1) \to q_0$ asserts that if function $\lambda x.s$ is applied to a tree accepted from *both* states q_0 and q_1 then it is guaranteed to generate a tree accepted from state q_0.

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. Similarly to *kind* environments, we often write $\Gamma, x : t$ to mean $\Gamma \cup \{x : t\}$. We will also extend the notion of well-kindedness to a relation on type and kind environments so that $\Gamma :: \mathcal{N}$ just if for every $F : \theta$ in Γ, $\theta :: \mathcal{N}(F)$.

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 term. Where $\mathcal{A}$ is clear from the context we use the turnstile without subscript. A judgement is valid just if it can be derived in the system in Table 2.3. The Var, App and Abs rules are standard in intersection type systems; note that the App rule requires that if a term $t_1 : (\bigwedge_{i \in I} \theta_i) \to \theta$ is applied to some term t_2, then it is necessary to prove $t : \theta_i$ for every $i \in I$. Dually for Abs, every type

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

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

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

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

Table 2.3: DTT Intersection type assignment system

in the leftmost intersection is added to the environment. However, the novelty lies in the Term rule where the rule premise depends on the transition function of the automaton to assign types to terminal symbols. In $\mathcal{A}_1$, the transition $\delta(q_1, b) \mapsto q_1$ can be used to infer the type $b : q_1 \to q_1$, meaning that if b is applied to a term t accepted from q_1 then the resulting term $b\ t$ is also accepted from q_1. This neatly follows the definition of a run-tree of an automaton.

Example 2.23 (Type assignment)**.** Recall the right-hand side of S from Example 2.9: $F\ c$. This term may be assigned the type q_0 in $\vdash_{\mathcal{A}_1}$ under the environment $\Gamma_1 = \{F : q_0 \wedge q_1 \to q_0\}$:

$$\frac{\dfrac{}{\Gamma_1 \vdash_{\mathcal{A}_1} F : q_0 \wedge q_1 \to q_0} \qquad \dfrac{}{\Gamma_1 \vdash_{\mathcal{A}_1} c : q_0} \qquad \dfrac{}{\Gamma_1 \vdash_{\mathcal{A}_1} c : q_1}}{\Gamma_1 \vdash_{\mathcal{A}_1} F\ c : q_0}$$

Consider further the right-hand side of G from Example 2.11: $\lambda x.b\ (G\ x)$. This term may be assigned the type $q_0 \to q_1$ in $\vdash_{\mathcal{A}_1}$ under the environment $\Gamma_2 = \{G : q_0 \to q_1\}$:

$$\frac{\dfrac{\dfrac{}{\Gamma_2, x : q_0 \vdash b : q_1 \to q_1} \qquad \dfrac{\dfrac{}{\Gamma_2, x : q_0 \vdash G : q_0 \to q_1} \qquad \dfrac{}{\Gamma_2, x : q_0 \vdash x : q_0}}{\Gamma_2, x : q_0 \vdash G\ x : q_1}}{\Gamma_2, x : q_0 \vdash b\ (G\ x) : q_1}}{\Gamma_2 \vdash \lambda x.b\ (G\ x) : q_0 \to q_1}$$

Although the type system allows us to type individual terms, as it stands it does not tell us about reduction. As our goal is that a judgement $\Gamma \vdash t : q$ should guarantee that t should reduce to a tree accepted from state q, we require that

subject reduction should hold. This is the property from which Milner's [1978] slogan "well-typed programs can't go wrong" (and associated theorem) is derived. We must impose a notion of *consistency* on type environments that in some sense corresponds to our program (HORS) being well-typed.

Definition 2.24 (Consistency)**.** Fix a HORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ and DTT $\mathcal{A}$. We say that a type environment Γ is $(\mathcal{G}, \mathcal{A})$-consistent when:

(i) $\Gamma :: \mathcal{N}$

(ii) $\forall (F : \theta) \in \Gamma : \Gamma \vdash_{\mathcal{A}} \mathcal{R}(F) : \theta$

The first condition here, namely that Γ refines the kinding environment $\mathcal{N}$ gives our first hint at decidability. As there are only finitely many well-kinded intersection types of each kind, there are only finitely many type environments that refine a given $\mathcal{N}$. The second condition requires that each rewrite rule be consistent with any types we assume for the corresponding non-terminal, and it is this that ensures that reduction preserves typability.

Example 2.25 (Consistent type environments)**.** The type environment $\Gamma_1 = \{F : (q_0 \wedge q_1) \to q_0\}$ used in Example 2.23 is $(\mathcal{G}_1, \mathcal{A}_1)$-consistent as $\Gamma_1 \vdash_{\mathcal{A}_1} \lambda x.a\ x\ (F\ (b\ x)) : (q_0 \wedge q_1) \to q_0$.

Take the type environment $\Gamma_3 = \{F : q_0 \to q_0\}$, and observe that we could have used it in place of Γ_1 in Example 2.23 above i.e. $\Gamma_3 \vdash_{\mathcal{A}_1} F\ c : q_0$. However, $\Gamma_3 \nvdash_{\mathcal{A}_1} \mathcal{R}(F) : q_0 \to q_0$ because one of the subgoals in the typing derivation will be $\Gamma_3, x : q_0 \vdash_{\mathcal{A}_1} b\ x : q_0$. Using the TERM rule, the only appropriate type for b is $q_1 \to q_0$ (from transition $\delta(q_0, b) = q_1$) which then requires x to have type q_1. It follows that Γ_3 is *not* $(\mathcal{G}_1, \mathcal{A}_1)$-consistent.

The key theorem due to Kobayashi [2013] ensures that the type system characterises the model checking problem precisely. First, with the soundness property we discussed above, if the start symbol S is known to be typed with the initial state of the automaton, q_0, in a consistent type environment, then the tree generated from S must be accepted by the automaton. Kobayashi also proved a completeness result guaranteeing that for a YES instance of the model checking problem, there must exist a witnessing type environment.

Theorem 2.2 (Soundness and completeness of $\vdash_{\mathcal{A}}$)**.** *$[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}^{\perp}$ if and only if there exists a $(\mathcal{G}, \mathcal{A})$-consistent type environment Γ such that $S : q_0 \in \Gamma$.*

Example 2.26 (Witness for $[\![\mathcal{G}_1]\!] \in \mathcal{L}(\mathcal{A}_1^{\perp})$)**.** We already know from Example 2.25 that $\Gamma_1 = \{F : (q_0 \wedge q_1) \to q_0\}$ is $(\mathcal{G}_1, \mathcal{A}_1)$-consistent. However, it does not satisfy the requirement that it contains $S : q_0$. We also know from Example 2.23 that $\Gamma_1 \vdash F\ c : q_0$. As $F\ c$ is the right-hand side of S, we may add $S : q_0$ to Γ_1 and preserve

consistency. Thus, $\{S : q_0, F : (q_0 \wedge q_1) \to q_0\}$ is a witness to $[\![\mathcal{G}_1]\!]$ being accepted by $\mathcal{A}_1^{\perp}$ in the sense of Theorem 2.2.

Shrink and the Naïve algorithm

We previously remarked that due to the requirement that a consistent type environment refine the kind environment provided with the HORS, there are only finitely many possible such environments. The definition of consistency of type environments whereby each type binding in the environment must hold for the right-hand side of the corresponding non-terminal suggests a natural filtering function, which Kobayashi named Shrink:

$$\mathsf{Shrink}(\Gamma) = \{F : \theta \mid F : \theta \in \Gamma, \Gamma \vdash_{\mathcal{A}} \mathcal{R}(F) : \theta\}$$

This function removes bindings from Γ that do not satisfy the consistency requirement. If we organise the space of well-kinded type environments into a lattice, using the subset ordering, then $\emptyset$ is the bottom element and the set of all well-kinded type environments, Γ_{max}, is the top element. Shrink is a monotone function on this lattice and therefore necessarily has fixed-points. It follows from the definition that any fixed-point will also be a consistent type environment, as there are no further bindings to remove. As usual in a finite lattice, the greatest fixed-point can be calculated by repeated application of the function to the top element. By definition the greatest fixed-point is the largest consistent type environment and so if it does not contain $S : q_0$, then no consistent type environment does:

$$S : q_0 \in \nu\mathsf{Shrink} \iff [\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{A}^{\perp})$$

This yields what Kobayashi dubbed the "naïve" algorithm, due to the huge constant factor caused by the initial type environment Γ_{max}.

Complexity

The complexity of the naïve algorithm was analysed by Kobayashi and found to be hyper-exponential in the order (n) of the scheme, with exponent polynomial in the maximum arity (A) of the scheme and number of states ($|Q|$) of the automaton for any $\varepsilon > 0$:

$$\mathcal{O}(|\mathcal{G}|\mathsf{exp}_n((A \times |Q|)^{1+\varepsilon}))$$

This is a direct consequence of the space of possible intersection types i.e. the size of Γ_{max}.

Kobayashi and Ong [2011] also gave a tighter bound for *disjunctive* APT, where the space of possible types is reduced. For this class the model checking problem

is $(n-1)$-ExpTime hard; the result was also extended to DTT, by complementation of disjunctive ATT. However, as yet no model checking algorithms have exploited these tighter bounds.

Model Checking Tools

Two tools for solving the HORS/DTT model checking problem predate our own work.

TRecS The tool TRecS [Kobayashi, 2009b] implements Kobayashi's *hybrid* algorithm, which was the first practical algorithm in the sense that it did not display worst-case behaviour on all inputs. The hybrid algorithm uses the same key observation as the naïve algorithm i.e. that a fixed-point of the Shrink function is necessarily a consistent type environment, and attempts to search for a smaller initial candidate Γ_{max}.

The algorithm proceeds in three phases, first constructing a *configuration graph*, which is a finite prefix of reduction of the recursion scheme in tandem with the transition function of the automaton, starting from the initial configuration (S, q_0). If the head symbol of the term in a configuration is a non-terminal, then the term is reduced to yield a new child configuration with the same state. Otherwise, the head symbol is a terminal, so the current configuration is $(a\ t_1 \cdots t_n, q)$, and if $\delta(q, a) = q_1 \cdots q_n$ then a child (t_i, q_i) is created for each $i \in [1..n]$. If there is no transition for (q, a), then the algorithm returns No. Following this pattern of expansion, each configuration describes a term and a state of the automaton from which the tree it generates must be accepted.

The graph is expanded for some finite number of iterations and then in the second phase types are extracted from the graph, which are all somehow relevant, as they correspond to how the HORS actually behaves. The type extraction is defined inductively on the kind of subterms in configurations, building up from the ground-kind assertions ($t : q$ for each configuration (t, q)). This uses a method from the proof technique used in [Kobayashi and Ong, 2009]. As the configuration graph is not complete, some behaviours may not be captured by it, and there is an element of guessing in generating a candidate type environment from the extracted types.

Once a candidate type environment Γ has been extracted, the greatest fixed-point Γ' of Shrink below Γ is computed. Compared to Γ_{max}, Γ is typically much smaller, and so this process is likely to terminate quickly. As Γ' is guaranteed to be consistent, if it also contains $S : q_0$ then the algorithm returns Yes. Otherwise, the configuration graph is expanded further, eventually uncovering sufficient useful type information or encountering a bad configuration.

GTRecS The GTRecS algorithm [Kobayashi, 2011] was developed in response to the complexity-theoretic analysis of the hybrid algorithm, which revealed that the size of the input HORS was found in the (height n) exponent. Although TRecS performed well for many inputs, certain artificial schemes such as the $\mathcal{G}_{m,n}$ family revealed the pathological behaviour. This family of examples, parameterised by the order (m) and depth of nesting (n), generates a finite tree of hyper-exponential size.

The algorithm is designed around two fixed-point calculations, first computing a greatest fixed-point Γ of a (increasing) function on type environments that uses information from the right-hand sides of the rules to determine how the non-terminals may interact. The greatest-fixed point of Shrink under Γ can then be computed, as for the hybrid algorithm, to obtain a consistent type environment. The algorithm returns Yes just if $S : q_0$ is present in the final environment. This algorithm is fixed-parameter tractable i.e. if the order and arity of the HORS and the number of states of the automaton are bounded, then the runtime is linear in the number of rules of the scheme. As the algorithm does not base its analysis on partial reduction of the HORS, Γ may include many irrelevant types, and this proved to be a significant bottleneck. However, GTRecS did show that it is possible to overcome the weaknesses of TRecS with respect to certain pathological classes of schemes.

2.3 Applications to Verification

Although the relationship between higher-order functional programs and HORS is clear, exactly how to leverage the decidability result is not so clear. In this section we will present some examples of verification problems that can be solved by reduction to HORS model checking.

Resource Usage

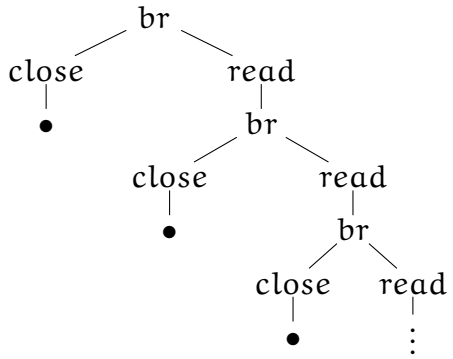
In Kobayashi's original introduction of the intersection-type approach, he also outlined an application to the resource usage verification problem [Igarashi and Kobayashi, 2005]. The resource usage problem: "Does a given program access resources such as files and memory in a valid manner?" is phrased using a simple call-by-value calculus. Rather than introducing the calculus here, we give an example in ML-style syntax, with $*$ representing a non-deterministic Boolean value:

```
let rec g x = if * then close(x) else (read(x); g(x)) in
let d = open_in "foo" in g(d)
```

We may wish to verify that the program only accesses files in manner conforming with the specification $\texttt{read}^*\texttt{close}$. In order to model the call-by-value program behaviour using a recursion scheme we need to perform a continuation-passing-style (CPS) transform [Plotkin, 1975; Danvy and Filinski, 1992] to enforce the correct reduction strategy, which could result in the HORS below.

$$\begin{aligned} S &= G\ d\ \bullet \\ G &= \lambda x k.\mathtt{br}\ (\mathit{close}\ k)\ (\mathit{read}\ (G\ x\ k)) \end{aligned}$$

This scheme can be seen to closely follow the original program, and with the control flow in CPS, terminal symbols can be inserted into the program to generate a tree of possible computations. The new terminal, br, represents the nondeterministic choice between closing and reading the file. Every path in the value tree, seen below, represents a possible computation. A suitable automaton can easily be constructed to verify the original property, treating the br symbols conjunctively.



The reduction from the call-by-value calculus, as it did not have infinite data types, was shown to be sound and complete. However, for richer languages, some sort of abstraction process is required, and we will describe three such now.

Tree transducers

The first attempt from Kobayashi et al. [2010] was to introduce an extension of tree transducers to the higher-order case. Higher-order, multi-parameter, tree transducers (HMTTs), allow for tree destruction as well as the construction offered by HORS. The HMTT verification problem is to determine whether an HMTT, given n trees known only to be accepted by given trivial tree automata, outputs a tree accepted by another automaton. The strength of the computation of HMTTs makes the problem undecidable, but the authors presented a sound reduction to the model checking problem for a syntactically extended form of HORS: *recursion schemes with finite data domains* (RSFD). We will discuss RSFD in greater detail later

in this dissertation, for now we note only that they are equi-expressive with HORS as generators of tree languages and that their model checking problem is thus decidable.

The abstraction in this case lies in the reduction to RSFD, where the finite data domain only approximates the input trees as they are destructed by the HMTT. Kobayashi et al. gave reductions from a number of problems, including the resource usage problem, verification of XML-processing programs and string analyses to HMTT verification.

Predicate abstraction

As we mentioned in Chapter 1, HORS are a natural higher-order lifting of Boolean programs. Kobayashi et al. [2011] used this observation to motivate a corresponding lifting of the predicate abstraction technique used in the SLAM framework [Ball and Rajamani, 2002]. The predicates, expressions such as $\lambda x.x > 0$, play the part of base types in a system of *dependent abstraction types* which can then be used to abstract the program to a *higher-order Boolean program*, reducible to an RSFD or HORS.

They incorporate a similar *counterexample-guided abstraction refinement* loop into their work, so that spurious counterexamples may be ruled out by strengthening the abstraction types assigned to program functions. The loop comes with a proof of relative (with respect to the dependent type system) completeness; the type system was also later extended to be complete (relative to the underlying logic) by addition of "dummy" variables to allow the type of higher-order functions to depend on its parameters [Unno et al., 2013]. The authors report encouraging results in this and a subsequent, optimisation-focused paper [Sato et al., 2013], although scalability of the approach to large programs is still an open question.

Pattern abstraction

Flow analysis such as the k-CFA framework Shivers [1991] and the grammar abstraction of Jones and Andersen [2007] has for some years been the most successful automated approach to inferring properties of higher-order functional programs. Ong and Ramsay [2011] took inspiration from the flow analysis community and designed a CEGAR loop based on abstracting variables in pattern matches. The verification problem they addressed was for *pattern-matching recursion schemes* (PMRS), which are essentially a simply-typed, pure, call-by-value functional language. Given a regular set of input terms, they look to answer whether the output of the PMRS falls within another regular set of terms. By using a flow analysis to over-approximate the terms that may flow to the pattern-match variables using a regular grammar, they construct a so-called *weak* PMRS that leaves the pure

variables unaffected and is strictly more precise than the Jones and Andersen algorithm. Notice that similarly to the HMTT work, abstracting the behaviour when destructing terms is the crucial step that yields decidability, as the model checking problem for weak PMRS is decidable. Given a spurious counter-example trace, the refinement step of the algorithm unrolls the pattern-match expression whose approximation led to the spurious behaviour, causing a larger prefix of the matched term to be represented precisely. This gives a semi-completeness result for No instances, but does mean that some classes of spurious behaviour cannot be ruled out. We make use of an implementation of this algorithm in Chapter 4 to generate large input instances for our model checking algorithm in order to probe its scalability.

Chapter 3

Traversals for Model Checking

In this chapter we will present our first novel contribution. We will introduce the notion of *traversals* over *computation trees* due to Ong [2006] and used in his proof of decidability of the HORS model checking problem. Using the concept of traversals, we will derive a new model checking algorithm using intersection types for the HORS/DTT model checking problem. Finally, we will show that traces of computation of this algorithm are isomorphic to traversals, thus clarifying the relationship between the game semantics approach [Ong, 2006] and the newer intersection type approach [Kobayashi, 2009a; Kobayashi and Ong, 2009]. This work was carried out jointly with Ong and Ramsay and some material was published in the proceedings of ICFP 2012 [Neatherway et al., 2012]; in that paper the connection with game semantics was only sketched. I was responsible for the development and formalisation of the algorithm presented here, along with the proofs of its correctness. Ong provided guidance with the proofs of completeness and correspondence, and I am very grateful to both my co-authors for many fruitful discussions during the course of this work.

3.1 Traversals of Recursion Schemes

We will start by giving the intuition behind traversals and their relationship to the call-by-name reduction semantics of HORS. We defer the precise definitions to the proof of isomorphism towards the end of this chapter.

A computation tree $\lambda(\mathcal{G})$ of a recursion scheme $\mathcal{G}$ is a regular, possibly-infinite tree analogous to a Böhm tree and generated by the *long transform* $\overline{\mathcal{G}}$ of $\mathcal{G}$. The long transform is obtained by recursively η-expanding the rewrite rules, replacing λ by a constant symbol $\boldsymbol{\lambda}$ and introducing a placeholder for application of non-terminals '@'. The result is a regular grammar (equivalently an order-0 HORS),

which generates the syntax of the original HORS.

Example 3.1 (Construction of long transform)**.** Recalling the rewrite rules from Example 2.9, applying the transform yields:

$$\mathcal{G}_1 \begin{cases} S = F\,c \\ F = \lambda x.a\;x\;(F\;(b\;x)) \end{cases} \quad \mapsto \quad \overline{\mathcal{G}_1} \begin{cases} S = \boldsymbol{\lambda}.@\;F\;(\boldsymbol{\lambda}.c) \\ F = \boldsymbol{\lambda} x.a\;(\boldsymbol{\lambda}.x)\;(\boldsymbol{\lambda}.@\;F\;(\boldsymbol{\lambda}.b\;(\boldsymbol{\lambda}.x))) \end{cases}$$

The output of the transform contains so-called "dummy lambdas", so that for example the transform of the application $b\;x$ is $b\;(\boldsymbol{\lambda}.x)$, where $\boldsymbol{\lambda}$ should be thought of as binding an empty list of variables. This affords us some technical convenience by assuring that lambda-abstractions alternate with non-lambda-abstractions.

The computation tree $\lambda(\mathcal{G}_1)$ is the underlying tree in Figure 3.1 whose nodes are labelled by the symbols $\boldsymbol{\lambda}, \boldsymbol{\lambda}x, @, x, a, b$ and c. Note that the tree is just that obtained by unfolding the rules of $\overline{\mathcal{G}_1}$. The annotations describe two traversals that compute those paths labelled $a \cdot c$ and $a \cdot a \cdot b \cdot c$ in the value tree, $[\![\mathcal{G}_1]\!]$, which is reproduced on the right for reference.

To trace the first of these, first take the path in $\lambda(\mathcal{G}_1)$ from the root to (1), following the left branch at both @ and a. Taking the left branch at the @ symbol corresponds to a reduction of the nonterminal F. On reaching x at (1), we jump to the segment starting at (1) i.e. from formal parameter x to actual parameter, and terminate at arity-0 terminal c.

For the second we can instead take the right branch at a and proceed into another unfolding of F, taking the path to (2). Again we jump from formal parameter to actual parameter starting from (2), moving over b, before encountering another x at (3). Note that this x is bound by the enclosing λ and so we jump to (3) to find the concrete parameter c.

We considered two traversals:

(i) $\boldsymbol{\lambda} \cdot @ \cdot \boldsymbol{\lambda}x \cdot a \cdot \boldsymbol{\lambda} \cdot x \cdot \boldsymbol{\lambda} \cdot c$

(ii) $\boldsymbol{\lambda} \cdot @ \cdot \boldsymbol{\lambda}x \cdot a \cdot \boldsymbol{\lambda} \cdot @ \cdot \boldsymbol{\lambda}x \cdot a \cdot \boldsymbol{\lambda} \cdot x \cdot \boldsymbol{\lambda} \cdot b \cdot \boldsymbol{\lambda} \cdot x \cdot \boldsymbol{\lambda} \cdot c$

with Σ-projections $a \cdot c$ and $a \cdot a \cdot b \cdot c$. The prefix of the (infinitely many) remaining traversals takes the right-hand branch at the second a in $\lambda(\mathcal{G}_1)$ and each has as Σ-projection a path from the value tree. It is a theorem (due to Ong [2006]) that these traversals are in a correspondence with paths in the value tree:

Theorem 3.1 (Path-Traversal Correspondence)**.** *Fix a HORS $\mathcal{G}$.*

(i) There is a 1-1 correspondence, $p \mapsto t_p$, between maximal paths p in the value tree $[\![\mathcal{G}]\!]$ and maximal traversals t_p in the computation tree $\lambda(\mathcal{G})$.

(ii) The Σ-projection of a maximal path p in $[\![\mathcal{G}]\!]$ and the corresponding traversal t_p coincide.

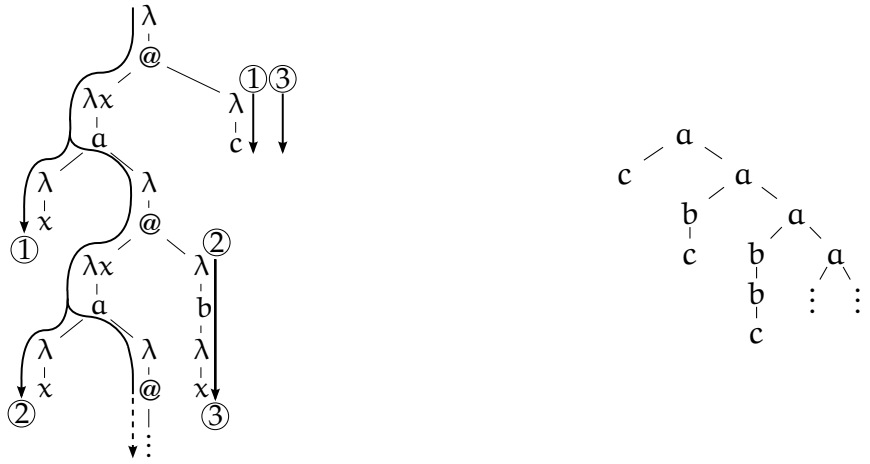


Figure 3.1: Traversals in $\lambda(\mathcal{G}_1)$ and $[\![\mathcal{G}_1]\!]$

Traversals versus Intersection Types

We have given this brief introduction to traversals in order that the reader might see the structure of traversals in the way that the model checking algorithm operates. For the development of the algorithm itself we use the machinery of intersection types rather than traversals. It would certainly be possible to define the algorithm differently, associating the behaviour of formal parameters with what Ong dubbed *variable profiles*, rather than intersections of types. However, working with types may be easier for a reader (as well as this author!) coming from a programming languages or program analysis background, and although the same modularity afforded by most type systems can also be found in the game semantics underlying traversals, we find the intersection type presentation to be more accessible.

3.2 An Algorithm for Model Checking

Our approach to the model checking problem exploits the characterisation by the intersection type system stated in Theorem 2.2. We give a decision procedure working from the premises of the theorem, which require that any witnessing type environment must (i) be consistent, and (ii) contain $S : q_0$.

To this end, we build typing derivations lazily, focusing on the return type of terms, that is, the type of tree they are required to produce in order to satisfy the property. Consider a term $t_0\, t_1 \cdots t_n$ that 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 equally be viewed as a typing judgement $\vdash t_0\ t_1 \cdots\ t_n : q$, which, after n applications of the App rule, yields the subgoal $\vdash t_0 : \theta$ 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:

- If t_0 is a terminal symbol, then we must use the TERM rule. Given $\delta(q, a)$ is unique, all α_i will be fully determined, yielding n judgements of the form $t_i : \alpha_i$ to prove.
- If we encounter a non-terminal, say $t_0 = F$, then we must assume $F : \theta$ to use the VAR rule. To maintain consistency in the sense of Theorem 2.2 we must also start a new derivation showing that $\mathcal{R}(F) : \theta$.
- In case t_0 is a variable (i.e. a formal parameter), we must ensure that the corresponding actual parameter has type θ.

This use of type variables (such as the α_i above) allows us to capture the connection made by term variables between typing derivations and ensure that the derivations we construct in search of our witnessing type environment are always mutually consistent. Type variables are instantiated to a set of *open types*, which may contain type variables in turn and may be reified to concrete intersection types.

For the purposes of an example, we first give a slightly modified version of $\mathcal{A}_1$ from Chapter 2. $\mathcal{A}_3 = \langle \Sigma, \{q_0, q_1\}, \delta, q_0 \rangle$ where $\Sigma = \{a \mapsto 2, b \mapsto 1, c \mapsto 0\}$ as before, and δ is:

$$\begin{aligned} \delta(q_0, a) &= q_1 q_0 \\ \delta(q_1, b) &= q_1 \\ \delta(q_1, c) &= \varepsilon \end{aligned}$$

Thus $\mathcal{A}_3$ accepts a Σ-labelled tree t just if a and b are seen only on the left of a c.

Example 3.2 (Building a derivation for "$[\![\mathcal{G}_1]\!] \in \mathcal{L}(\mathcal{A}_3^{\perp})$?")**.** As we said, a requirement of the theorem is that a witnessing type environment contains $S : q_0$, so assuming this to be in our environment it immediately becomes necessary to build a derivation rooted at $\emptyset \vdash \mathcal{R}(S) : q_0$ in order to maintain consistency. Here we apply the APP rule, to obtain the subgoal $F : \alpha_1 \to q_0$ where α_1 is a fresh type variable, and then aim to show that $\mathcal{R}(F) : \alpha_1 \to q_0$. This leaves us with two more derivations as follows, one (the latter) only partially complete:

$$\dfrac{\dfrac{}{\{F : \alpha_1 \to q_0\} \vdash F : \alpha_1 \to q_0}\,\textsc{Var}}{\{F : \alpha_1 \to q_0\} \vdash F\,c : q_0}\,\textsc{App}$$

and

$$\frac{\{x : \alpha_1\} \vdash a\ x\ (F\ (b\ x)) : q_0}{\emptyset \vdash \lambda x.a\ x\ (F\ (b\ x)) : \alpha_1 \to q_0}\ \textsc{Abs}$$

where α_1 is instantiated to $\bigwedge \emptyset = \top$ initially, avoiding the need to show any typing for c, the argument to F in $\mathcal{R}(S)$. Notice that taking the two non-terminal typings as $\Gamma = \{S : q_0, F : \top \to q_0\}$ in the sense of Theorem 2.2, the derivations to ensure that the right-hand sides match the typings are already in place, although as yet incomplete. After two more vacuous applications of the App rule in our incomplete derivation, the term in the current subgoal will be the single terminal c, and so we use the Term rule to close the judgement—see the first open-derivation in Table 3.1. This requires α_3 and α_2 to take on the values q_1 and q_0 respectively, triggering the addition of an open judgement for each argument ('x' and '$F\ (b\ x)$') of a.

$$\textsc{Term}\ \frac{\textsc{App}\ \dfrac{\textsc{App}\ \dfrac{\textsc{App}\ \dfrac{\{x : \alpha_1\} \vdash a : \alpha_3 \to \alpha_2 \to q_0 \quad \{x : \alpha_1\} \vdash x : q_1}{\{x : \alpha_1\} \vdash a\ x : \alpha_2 \to q_0} \textsc{Var} \qquad \{x : \alpha_1\} \vdash F\ (b\ x) : q_0}{\{x : \alpha_1\} \vdash a\ x\ (F\ (b\ x)) : q_0}}{\emptyset \vdash \lambda x.a\ x\ (F\ (b\ x)) : \alpha_1 \to q_0}\ \textsc{Abs}}$$

$$\textsc{Var}\ \frac{\{F : \alpha_1 \to q_0\} \vdash F : \alpha_1 \to q_0 \qquad \{F : \alpha_1 \to q_0\} \vdash c : q_1}{F : \alpha_1 \to q_0 \vdash F\ c : q_0}\ \textsc{Term}\ \textsc{App}$$

Table 3.1: Examples of open derivations

To complete this example, we take the open judgement $\{x : \alpha_1\} \vdash x : q_1$. In order to close this with the Var rule, we require that α_1 contain q_1. After making this adjustment, the use of the App rule to close the judgement $F : \alpha_1 \to q_0 \vdash F\ c : q_0$ is no longer valid, and we must add an extra judgement (see the second open derivation in Table 3.1), which in turn is closed by the Term rule. This captures informally how we build up the typing derivations.

The relationship to traversals can be seen in the path taken by the construction of the typing derivations. Recall the first traversal we took in $\lambda(\mathcal{G}_1)$:

$$\boldsymbol{\lambda} \cdot @ \cdot \boldsymbol{\lambda} x \cdot a \cdot \boldsymbol{\lambda} \cdot x \cdot \boldsymbol{\lambda} \cdot c$$

Taking @ to be in correspondence with F, and modulo the inserted "dummy lambdas", the completed sequence of constructed judgements has terms with the same head symbols as those encountered in the traversal:

$$F\ c \cdot F \cdot \lambda x.a\ x\ (F\ (b\ x)) \cdot a\ x\ (F\ (b\ x)) \cdot a\ x \cdot a \cdot x \cdot c$$

and furthermore the same "jump" is made from formal to actual parameter. We will make this correspondence precise after formally defining the algorithm.

Notice that if we take Γ to be the union of all non-terminal type bindings in the various derivations then

(i) $\Gamma :: \mathcal{N}$,

(ii) If all judgements are closed then $\forall (F : \theta) \in \Gamma \cdot \Gamma \vdash \mathcal{R}(F) : \theta$, and

(iii) $S : q_0 \in \Gamma$.

Clearly if the tree generated by a HORS 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 3.2. First we introduce *open types*, which represent intersection types using type variables. An open type takes the form $\alpha_1 \to \cdots \to \alpha_n \to q$ where each type variable α_i ranges over finite sets of open types and $q \in Q$. In the context of an instantiation map (to be defined shortly), an open type can be *reified* to an intersection type. Formally we define open types inductively. We take for each kind κ a denumerable set A_κ of type variables, and further define the set $\mathbb{P}_\kappa$ of open types of kind κ as:

$$\mathbb{P}_o := Q \qquad \mathbb{P}_{\kappa_1 \to \kappa_2} := \{ \alpha \to \theta^o \mid \alpha \in A_{\kappa_1}, \theta^o \in \mathbb{P}_{\kappa_2} \}$$

Let $A := \bigcup_{\kappa \in \mathbb{S}_o} A_\kappa$ and $\mathbb{P} := \bigcup_{\kappa \in \mathbb{S}_o} \mathbb{P}_\kappa$. We say that a function $\Theta : A \to \mathcal{P}(\mathbb{P})$ is an *instantiation map* if it is

(i) *finite*: there exists a finite subset C of A such that Θ maps every element of $(A \setminus C)$ to $\emptyset$, and

(ii) *kind-respecting*: for each kind κ, Θ restricts to a function from A_κ to the set $\mathcal{P}_{\text{fin}}(\mathbb{P}_\kappa)$ of finite subsets of $\mathbb{P}_\kappa$.

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

$$\widehat{\Theta}_o(q) := q$$

$$\widehat{\Theta}_{\kappa_1 \to \kappa_2}(\alpha \to \theta^o) := \left(\bigwedge_{\theta_1^o \in \Theta(\alpha)} \widehat{\Theta}_{\kappa_1}(\theta_1^o) \right) \to \widehat{\Theta}_{\kappa_2}(\theta^o)$$

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

Example 3.3. 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:

$$\begin{aligned}
&\alpha_1 \mapsto \{\, \alpha_3 \to q_2, \alpha_4 \to q_1 \,\} \\
&\alpha_2 \mapsto \{\, q_1 \,\} \\
&\alpha_3 \mapsto \emptyset \\
&\alpha_4 \mapsto \{\, \alpha_5 \to q_0 \,\} \\
&\alpha_5 \mapsto \{\, q_0 \,\}
\end{aligned}$$

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 intersection types.

We first introduce some notational conventions. We use the superscript 'o' to mean *open* in the sense of containing variables. We use $\theta^o, \theta^o_1, \cdots$ to range over open types and $\Gamma^o, \Gamma^o_1, \cdots$ to range over *open-type environments*. We also extend this notation to typing judgements and derivations.

Definition 3.4 (Open judgements)**.** Open judgements are judgements constructed from open types which take the form $\Gamma^o \vdash t : \theta^o$. We sometimes wish to abbreviate an open judgement just as J (indicated by $J = \Gamma^o \vdash t : \theta^o$), and set $\mathsf{env}(J) = \Gamma^o$, $\mathsf{tm}(J) = t$ and $\mathsf{ty}(J) = \theta^o$. In the context of a typing derivation, we allow *judgements* which are as yet *unjustified*. Such judgements do not have a line over them and must be leaves.

Definition 3.5 (Open derivations)**.** Open derivations are trees of open judgements and we typically use $\Delta^o, \Delta^o_1, \Delta^o_2, \cdots$ to refer to such derivations. We call an open derivation *justified* if all the open judgements it contains are in turn justified. In the context of an instantiation map Θ and given some Δ^o we define the corresponding (reified) derivation Δ such that $\mathsf{dom}(\Delta) = \mathsf{dom}(\Delta^o)$ and for every $n \in \mathsf{dom}(\Delta^o)$, $\Delta(n) = \widehat{\Theta}(\Delta^o(n))$. A *valid* open derivation is one where in the reified derivation, every justified judgement must be an instance of a rule or axiom of the appropriate type system. Note that a valid justified open derivation rooted at $\Gamma^o \vdash t : \theta^o$ therefore witnesses $\widehat{\Theta}(\Gamma^o) \vdash t : \widehat{\Theta}(\theta^o)$. We write D for the set of open 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 $(F : \theta^o)$

where F is a non-terminal and θ^o is an open type; and n represents the subgoal of building a derivation for the judgement $\Gamma^o \vdash \mathcal{R}(\mathsf{F}) : \theta^o$ 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 creation of new nodes (corresponding to the subgoals); and Θ is updated. The root node is associated with the binding $(S : q_0)$, 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 $((\mathcal{N} \times \mathbb{P}) \times \mathsf{D})$-labelled tree, and Θ is an instantiation map. Each node n of $\mathcal{D}$ is labelled by a triple, $\mathcal{D}(n) = (\mathsf{F} : \theta^o, \Delta)$, such that the judgement at the root of the open derivation Δ^o has the form $\Gamma^o \vdash \mathcal{R}(\mathsf{F}) : \theta^o$ for some nonterminal F and open-type environment Γ^o. Henceforth we shall refer to Δ^o as the *open derivation* of n, and the pairs $(\mathsf{F} : \theta^o)$ and $(\mathsf{F} : \widehat{\Theta}(\theta^o))$ 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 *justified* if its open derivation Δ^o is justified (and we shall see—Lemma 3.2—that it follows that $\widehat{\Theta}(\Delta^o)$ is a valid type derivation of $\vdash_{\mathcal{A}}$); otherwise, the node is *unjustified*. The function $\mathsf{unjustified}$, when applied to $\mathcal{D}$, returns the set of judgements J that are currently unjustified (in some unjustified open derivation of $\mathcal{D}$).

The top loop of the algorithm is shown in Algorithm 1 and follows the ideas outlined in Example 3.2. As mentioned earlier, we must start with the unjustified judgement $\emptyset \vdash \mathcal{R}(S) : q_0$ and this informs the initialisation. Each *unjustified* judgement, $\Gamma^o \vdash s : \theta^o$, is then justified in turn by application of the appropriate A– rule (found in Table 3.2) depending on the shape of s. The A-α rule will be triggered by any use of A-Cst (resp. A-Var), both of which modify the instantiation map and require $\mathsf{arity}(a)$ (resp. one) additional judgement(s) to be added to keep the open derivations valid. The expansion is organised (by the inner **while** loop) into rounds, so that we expand all the unjustified judgements up to and including a nonterminal. This eases the proof of correctness by guaranteeing that the unjustified nodes are always the freshly created leaves.

The A– rules for constructing new open judgements are defined as tree transformations with a given state being found in the tree (typically a derivation with an unjustified judgement as a leaf) and some other side conditions being used as an input to modify the matching part of the derivation tree and produce a new state. If reading in colour, the red text can be used as a guide to the changes in the new state.

The termination of the loop is defined in terms of a *complete cut* of the justified initial subtree of $\mathcal{D}$.

Definition 3.6. Fix a state $(\mathcal{D}, \Theta)$. Then

$$\mathcal{D}^j := \{n \in \mathsf{dom}(\mathcal{D}) \mid n \text{ and all its } \mathcal{D}\text{-ancestors are justified}\}.$$

Algorithm 1: 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 $[\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{A}^{\perp})$
$\mathcal{D} :=$ singleton tree with label $(S : q_0, \emptyset \vdash \mathcal{R}(S) : q_0)$
$\Theta := \{\alpha \mapsto \emptyset \mid \alpha \in A\}$
while $\mathcal{D}^{j}$ does *not* have a complete cut **do**
  $W :=$ unjustified$(\mathcal{D})$
  while $W \neq \emptyset$ **do**
    $\Gamma^{o} \vdash s : \theta^{o} :=$ any element of W
    if state$(\theta^{o}) = q \wedge s \in \Sigma \wedge (q, s) \notin \delta$ **then**
      return No
    else
      Apply any matching A– rule possibly followed by A-α
    end
    $W := W \setminus \{\Gamma^{o} \vdash s : \theta^{o}\}$
    if $s \notin \mathcal{N}$ **then** $W := W \cup \{$all new unjustified judgements$\}$
  end
end
return YES

Thus $\mathcal{D}^{j}$ is the largest initial subtree of $\mathcal{D}$ consisting only of justified nodes.

Let t be a labelled tree. As usual a subset $C \subseteq \mathsf{dom}(t)$ is a *cut* of t just if for every maximal path B of t, $B \cap C$ is a singleton set. An *interior node* n of C, written $n \prec C$, is just any node that is an ancestor of some node in C.

Definition 3.7. We say that a cut C of the tree $\mathcal{D}^{j}$ is *complete* if for every $c \in C$, either:

(i) c is a leaf-node. With open-type binding $F : \theta^{o}$, it follows that $\emptyset \vdash \mathcal{R}(F) : \widehat{\Theta}(\theta^{o})$ is valid (since c is justified and thanks to Lemma 3.2, Page 46); or

(ii) there is an interior node of C that has the same reified-type binding as c.

Given a complete cut C we say $n \preccurlyeq C$ if and only if $n \prec C$ or $n \in C$. Observe that unjustified$(\mathcal{D}) = \emptyset$ if, and only if, every node of $\mathcal{D}$ is closed. Hence, if unjustified$(\mathcal{D}) = \emptyset$, the set of its leaf-nodes is a complete cut. In this case note that $\mathcal{D}$ is finite, as briefly discussed on page 40.

Example 3.8 (Completion of analysing $\mathcal{G}_1$ against $\mathcal{A}_3$)**.** We will now pick up the analysis of this example where we left off (Example 3.2). Recall that we had started a typing derivation for $\mathcal{R}(F)$:

Rule	If	Then
A-App	(i) $\Gamma^o \vdash t\ u : \theta^o$ (ii) α fresh	(i) $\dfrac{\Gamma^o \vdash t : \alpha \to \theta^o}{\Gamma^o \vdash t\ u : \theta^o}$
A-Fun	(i) $\Gamma^o \vdash F : \theta^o \quad (\in \Delta^o)$ (ii) $\mathcal{R}(F) = \lambda x_1 \dots x_n.s$ (iii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$	(i) $\overline{\Gamma^o, F : \theta^o \vdash F : \theta^o}$ (ii) For $\Gamma_1^o \vdash s : \theta_1^o$ in Δ^o, $\Gamma_1^o := \Gamma_1^o \cup \{F : \theta^o\}$ (iii) Add new rightmost child $(F : \theta^o, \Delta_1^o)$ to Δ^o where: $\Delta_1^o = \dfrac{\dfrac{\{x_i : \alpha_i \mid 1 \leqslant i \leqslant n\} \vdash s : q}{\vdots}}{\emptyset \vdash \mathcal{R}(F) : \theta^o}$
A-Var	(i) $\Gamma^o, x : \alpha \vdash x : \theta^o$	(i) $\overline{\Gamma^o, x : \alpha \vdash x : \theta^o}$ (ii) $\Theta(\alpha) := \Theta(\alpha) \cup \{\theta^o\}$
A-Cst	(i) $\Gamma^o \vdash a : \theta^o$ (ii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$ (iii) $\delta(q, a) = q_1 \cdots q_n$	(i) $\overline{\Gamma^o \vdash a : \theta^o}$ (ii) $\forall i \in [1..n] \cdot \Theta(\alpha_i) := \Theta(\alpha_i) \cup \{q_i\}$
A-α	(i) $\dfrac{\overline{\Gamma^o \vdash t : \alpha \to \theta_1^o} \quad \cdots}{J = \Gamma^o \vdash t\ u : \theta_1^o}$ (ii) $\theta^o \in \Theta(\alpha)$ (iii) $\Gamma^o \vdash u : \theta^o$ not a child of J	(i) $\dfrac{\overline{\Gamma^o \vdash t : \alpha \to \theta_1^o} \quad \Gamma^o \vdash u : \theta^o \quad \cdots}{\Gamma^o \vdash t\ u : \theta_1^o}$

Table 3.2: Rules of algorithm

$$\dfrac{\dfrac{\dfrac{\overline{\{x : \alpha_1\} \vdash a : \alpha_3 \to \alpha_2 \to q_0} \quad \overline{\{x : \alpha_1\} \vdash x : q_1}}{\{x : \alpha_1\} \vdash a\ x : \alpha_2 \to q_0} \quad \{x : \alpha_1\} \vdash F\ (b\ x) : q_0}{\{x : \alpha_1\} \vdash a\ x\ (F\ (b\ x)) : q_0}}{\emptyset \vdash \lambda x.a\ x\ (F\ (b\ x)) : \alpha_1 \to q_0}$$

At this point we have a single unjustified judgement in the top right. Looking at the form of the term, $F\ (b\ x)$, we must apply A-App, creating a fresh open judgement. This new judgement will have term F (and type $\alpha_4 \to q_0$) so we apply A-Fun, which will add the type binding $F : \alpha_4 \to q_0$ to the open type environment used in this derivation and create a new derivation as a child of this one, again for the right-hand side of F. After the initial uses of A-App, this new derivation will be:

$$\dfrac{\dfrac{\dfrac{\{x : \alpha_4\} \vdash a : \alpha_6 \to \alpha_5 \to q_0}{\{x : \alpha_4\} \vdash a\ x : \alpha_5 \to q_0}}{\{x : \alpha_4\} \vdash a\ x\ (F\ (b\ x)) : q_0}}{\emptyset \vdash \lambda x.a\ x\ (F\ (b\ x)) : \alpha_4 \to q_0}$$

and as the term in the unjustified judgement is just the terminal a we continue with A-Cst. In the property automaton, $\mathcal{A}_3$, $\delta(q_0, a) = q_1\ q_0$ and so we must update the reification map: $\Theta(\alpha_6) := \{q_1\}$ and $\Theta(\alpha_5) := \{q_0\}$. These updates then trigger two corresponding uses of A-α for the arguments of a. In the first case $q_1 \in \Theta(\alpha_6)$ but there is no judgement for $x : q_1$ and in the second $q_0 \in \Theta(\alpha_5)$ but there is no judgement for $F\ (b\ x) : q_0$. The addition of these gives:

$$\dfrac{\dfrac{\dfrac{\overline{\{x : \alpha_4\} \vdash a : \alpha_6 \to \alpha_5 \to q_0} \quad \{x : \alpha_4\} \vdash x : q_1}{\{x : \alpha_4\} \vdash a\ x : \alpha_5 \to q_0} \quad \{x : \alpha_4\} \vdash F\ (b\ x) : q_0}{\{x : \alpha_4\} \vdash a\ x\ (F\ (b\ x)) : q_0}}{\emptyset \vdash \lambda x.a\ x\ (F\ (b\ x)) : \alpha_4 \to q_0}$$

Lastly we will demonstrate A-Var, which will be used to justify the judgement '$\{x : \alpha_4\} \vdash x : q_1$' above. First we must again modify the reification map, with $\Theta(\alpha_4) := \{q_1\}$, and this triggers another use of A-α in the previous derivation for the argument of F: 'bx'. This can be seen in Table 3.3 at the top-left of Δ_2^o, which shows the final state of the algorithm, including the state tree and reification map. In this table the derivations we have just considered are Δ_2^o and Δ_3^o, while Δ_4^o is a further trivial open derivation for the right-hand side of F.

Upon reifying the open types in this state, the nodes labelled $(F : \alpha_1 \to q_0, \Delta_2^o)$ and $(F : \alpha_4 \to q_0, \Delta_3^o)$ both have the same reified type binding: $F : q_1 \to q_0$. These nodes are both in $\mathcal{D}^j$ as they are justified, and therefore the latter constitutes a singleton complete cut. As a result the environment we obtain from the three nodes after reification — $\{S : q_0, F : q_1 \to q_0\}$ — is guaranteed to be a witness to $[\![\mathcal{G}_1]\!] \in \mathcal{L}(\mathcal{A}_3^\perp)$.

$$\mathcal{D} = (S : q_0, \Delta_1^o) \rightarrow (F : \alpha_1 \to q_0, \Delta_2^o) \rightarrow (F : \alpha_4 \to q_0, \Delta_3^o) \rightarrow (F : \alpha_8 \to q_0, \Delta_4^o)$$

$$\Delta_1^o = \dfrac{\dfrac{}{\{F : \alpha_1 \to q_0\} \vdash F : \alpha_1 \to q_0} \quad \dfrac{}{\{F : \alpha_1 \to q_0\} \vdash c : q_1}}{\{F : \alpha_1 \to q_0\} \vdash F\,a : q_0}$$

$$\Delta_2^o = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2^o \vdash a : \alpha_3 \to \alpha_2 \to q_0} \quad \dfrac{}{\Gamma_2^o \vdash x : q_1}}{\Gamma_2^o \vdash a\,x : \alpha_2 \to q_0} \quad \dfrac{\dfrac{}{\Gamma_2^o \vdash F : \alpha_4 \to q_0} \quad \dfrac{\dfrac{}{\Gamma_2^o \vdash b : \alpha_7 \to q_1} \quad \dfrac{}{\Gamma_2^o \vdash x : q_1}}{\Gamma_2^o \vdash b\,x : q_1}}{\Gamma_2^o \vdash F\,(b\,x) : q_0}}{\Gamma_2^o \vdash a\,x\,(F\,(b\,x)) : q_0}}{\{F : \alpha_4 \to q_0\} \vdash \lambda x.a\,x\,(F\,(b\,x)) : \alpha_1 \to q_0}$$

$$\Delta_3^o = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3^o \vdash a : \alpha_6 \to \alpha_5 \to q_0} \quad \dfrac{}{\Gamma_3^o \vdash x : q_1}}{\Gamma_3^o \vdash a\,x : \alpha_5 \to q_0} \quad \dfrac{\dfrac{}{\Gamma_3^o \vdash F : \alpha_8 \to q_0}}{\Gamma_3^o \vdash F\,(b\,x) : q_0}}{\Gamma_3^o \vdash a\,x\,(F\,(b\,x)) : q_0}}{\{F : \alpha_8 \to q_0\} \vdash \lambda x.a\,x\,(F\,(b\,x)) : \alpha_4 \to q_0}$$

$$\Delta_4^o = \emptyset \vdash \lambda x.a\,x\,(F\,(b\,x)) : \alpha_8 \to q_0$$

$$\begin{aligned}\Theta &= \{\alpha_1 \mapsto \{q_1\}, \alpha_2 \mapsto \{q_0\}, \alpha_3 \mapsto \{q_1\}, \alpha_4 \mapsto \{q_1\},\\ &\quad \alpha_5 \mapsto \{q_0\}, \alpha_6 \mapsto \{q_1\}, \alpha_7 \mapsto \{q_1\}, \alpha_8 \mapsto \emptyset\}\\ \Gamma_2^o &= \{F : \alpha_4 \to q_0, x : \alpha_1\}\\ \Gamma_3^o &= \{F : \alpha_8 \to q_0, x : \alpha_4\}\end{aligned}$$

Table 3.3: A final state of the algorithm (Example 3.8)

Correctness

Our approach to correctness falls into two parts. We start with the simpler part, which is to establish that the tree of open derivations $\mathcal{D}$ is consistent, in that (i) the reification of justified derivations yields valid concrete derivations in $\vdash_{\mathcal{A}}$, and (ii) the children of a node with environment Γ^o are labelled with derivations that establish consistency for each non-terminal type binding in Γ^o. This property allows us to establish soundness for Yes instances, but for No instances and for completeness we require some more intricate reasoning. For these the reduction of the HORS and partial runs of the automaton must be related to the computation of the algorithm.

Lemma 3.2 (Consistency of typing derivations)**.** *Let $(\mathcal{D}, \Theta)$ be a state of the algorithm,*

n *be a node of* $\mathcal{D}$*, and* $\mathcal{D}(n) = (F : \theta^o, \Delta^o)$ *where the judgement at the root of the open derivation* Δ^o *is* $\Gamma^o \vdash \mathcal{R}(F) : \theta^o$*.*

(i) Every justified judgement of $\widehat{\Theta}(\Delta^o)$ *is an instance of a rule or axiom of* $\vdash_{\mathcal{A}}$*. Hence, if* Δ^o *is justified then* $\widehat{\Theta}(\Delta^o)$ *is a valid type derivation, witnessing* $\widehat{\Theta}(\Gamma^o) \vdash \mathcal{R}(F) : \widehat{\Theta}(\theta^o)$*.*

(ii) Let $\Gamma^o = \{F_1 : \theta_1^o, \cdots, F_l : \theta_l^o\}$*. Then* $\{n_1, \cdots, n_l\}$ *is the set of successor nodes of* n*, where* $\mathcal{D}(n_i) = (F_i : \theta_i^o, \Delta_i^o)$ *for each* i*.*

Proof. As defined in Algorithm 1, a step of the algorithm comprises an application of one of the A-Ξ rules ($\Xi \in \{\textsc{App}, \textsc{Cst}, \textsc{Var}, \textsc{Fun}\}$), followed by any matching A-α. The base case is vacuously true. We show that such a step preserves the required property. Let Δ^o be an open 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^o in turn.

- A-App; $u = s\,t$. Δ_1^o is obtained from Δ^o by an application of the App rule where $n = 0$.
- A-Fun; $u = F$. After augmenting the typing environment of every judgement in Δ^o by the binding $F : \theta^o$, the original judgement J is closed by the Var rule. The new node that is created is labelled by an open derivation which is obtained by n applications of the Abs rule to $\emptyset \vdash \mathcal{R}(F) : \theta^o$, as required.
- A-Var; $u = x$. 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 the (different) J in A-α. The use of App in the right-hand derivation fragment in A-α remains valid as the addition of Θ^o to the intersection represented by α is matched exactly by the introduction of the open node $\Gamma^o \vdash u : \theta^o$.
- A-Cst; $u = a$. The judgement J is closed using the Term rule. As shown in the A-Var case, A-α preserves the validity of the open derivations by adding a new derivation for each extension of a type variable. Setting each α_i to $\{q_i\}$ gives us a valid use of Term.

□

With this lemma in place, we can show that using a complete cut to build a witnessing type environment is sound. The requirement that every type binding for non-leaf nodes in the cut must also be found in the interior conceptually allows us to use the typing derivation for the interior node to satisfy $(\mathcal{G}, \mathcal{A})$-consistency.

Lemma 3.3 (Complete cuts yield witnesses)**.** *Fix a problem instance of a HORS* $\mathcal{G}$ *and a DTT* $\mathcal{A}$*. Let* $(\mathcal{D}, \Theta)$ *be a state of the algorithm. Suppose there is a complete cut* C *of* $\mathcal{D}^j$*. Define* Γ *to be the set:*

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

Then Γ is $(\mathcal{G}, \mathcal{A})$-consistent (in the sense of Theorem 2.2).

Proof. Take arbitrary $n \preccurlyeq C$ with $\mathcal{D}(n) = (F : \theta^o, \Delta^o)$. We need to show that there exists some $\Gamma' \subseteq \Gamma$ such that $\Gamma' \vdash_{\mathcal{A}} \mathcal{R}(F) : \widehat{\Theta}(\theta^o)$. We assume first that $n \prec C$ or that n is a leaf of $\mathcal{D}$, otherwise by the definition of a complete cut there must exist some interior node n' of C having the same reified type binding as n from which to construct Γ' and we take n' rather than n. Since $n \in \mathcal{D}^j$, from Lemma 3.2 $\widehat{\Theta}(\Delta^o)$ is a valid type derivation witnessing:

$$\{F_1 : \widehat{\Theta}(\theta_1^o), \ldots, F_l : \widehat{\Theta}(\theta_l^o)\} \vdash \mathcal{R}(F) : \widehat{\Theta}(\theta^o)$$

where n has children $\{n_1, \cdots, n_l\}$ and $\mathcal{D}(n_i) = (F_i : \theta_i^o, \Delta_i^o)$ for each i. It remains to show that for each $i \in [1..l]$, $F_i : \widehat{\Theta}(\theta_i^o) \in \Gamma$. If n is a leaf of $\mathcal{D}$, then $l = 0$ and we are done. Otherwise, choose an arbitrary n_i. As $n \prec C$, necessarily $n_i \preccurlyeq C$ and so from the definition of Γ, $F_i : \widehat{\Theta}(\theta_i^o) \in \Gamma$. □

The second phase of the proof requires several intermediate lemmas. We first introduce a notion of history in order to talk about the computation of the algorithm.

Definition 3.9 (Computation history)**.** Each judgement J has associated with it a notion of a *computation history* written $\mathsf{history}(J)$. This is the sequence of judgements (excluding itself) that were created by application of the rules of the algorithm that led to the creation of J. The initial judgement has $\mathsf{history}(\Gamma^o \vdash \mathcal{R}(S) : q_0) = \emptyset$. When a rule of the algorithm justifies some judgement J' by drawing a line over it and adding some new judgement J, then $\mathsf{history}(J) = \mathsf{history}(J') \cdot J'$.

Definition 3.10 (Computation paths)**.** A *path of computation* is a sequence of judgements that are added sequentially by the algorithm. Formally, any sequence of judgements $\langle J_i \rangle_{0 \leqslant i \leqslant N}$ (where $N \leqslant \omega$) such that:

1. $J_0 = \Gamma^o \vdash \mathcal{R}(S) : q_0 = \mathcal{D}(\varepsilon)(\varepsilon)$ i.e. the sequence starts at the initial judgement.
2. For all $0 \leqslant i \leqslant N$, $\mathsf{history}(J_i) = \langle J_j \rangle_{0 \leqslant j < i}$.

The next lemma is the key in relating the computation of the algorithm to running the automaton over the currently explored prefix of the value tree. Intuitively, if we reach a judgement $\Gamma^o \vdash a : \theta^o$ for some terminal a where $(\mathsf{state}(\theta^o), a) \notin \delta$ this should correspond to a point in the value tree labelled a where the automaton rejects. Computation histories are sequences of judgements, and to show this property we consider the subsequences of these finite histories containing only judgements $\Gamma^o \vdash \xi : \theta^o$ where ξ is a symbol in Σ (the Σ-projection) or $\mathcal{N}$ ($\mathcal{N}$-projection) respectively. In the following lemma, the first clause establishes

the relationship between the Σ-projection and a gradually increasing prefix of the value and run-trees. This requires the second clause, which in turn establishes the relationship between the $\mathcal{N}$-projection and a particular redex encountered after some steps of reduction from the start symbol.

Lemma 3.4 (Reduction). *In state* $(\mathcal{D}, \Theta)$*, for every judgement* $J = \Gamma^o \vdash t : \theta^o \in \mathcal{D}$ *where* $\mathsf{state}(\theta^o) = q$*:*

(i) The Σ*-projection of* $\mathsf{history}(J)$ *is a sequence* $J_0 \cdots J_n$ $(\forall i \in [0..n] \cdot J_i = \Gamma_i^o \vdash a_i : \theta_i^o)$ *such that there exist words* β, γ *of length* n *and if there exists a run-tree* R*, then* $R(\beta) = (\gamma, q)$*. Furthermore, if for each* $i \in [0..n]$ *we let* β_i *(resp.* γ_i*) be the prefix of* β *(resp.* γ*) of length* i*, then* $R(\beta_i) = (\gamma_i, q_i)$ *and* $[\![\mathcal{G}]\!](\gamma_i) = a_i$*.*

(ii) The $\mathcal{N}$*-projection of* $\mathsf{history}(J)$ *is a sequence* $K_1 \cdots K_{m-1}$ *such that if* $J = K_m$ *then each* $K_j (j \in [1..m])$ *is related to a subterm* t_j *of some* u_j *where* $S \to u_1 \to \cdots \to u_m$*. In particular,* $u_m|_\gamma = t_m$ *and where there are* i *judgements of the form* $\Gamma^o \vdash a : \theta^o \in \mathsf{history}(K_j)$*,* $u_j|_{\gamma_i} = t_j$*.*

Proof. For each $\Gamma^o \vdash t : \theta^o$, the corresponding t_m can be defined by a function reify defined as follows:

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

$$\mathsf{intro}(\alpha) := \mathsf{resolve}(\Gamma^o, u) \text{ where } \quad \frac{\overline{\Gamma^o \vdash t : \alpha \to \theta^o} \quad \cdots}{\Gamma^o \vdash t\, u : \theta^o}$$

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

It is not possible to change either the term, or the variable bindings in the environment of a judgement, and so existing relationships between judgements and subterms are not broken. We show that for the current judgement, $t_m := \mathsf{reify}(\Gamma^o \vdash t : \theta^o)$ 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 $t_1 = \mathsf{reify}(\emptyset \vdash \mathcal{R}(S) : q_0) = \mathcal{R}(S)$. This is certainly a subterm of $u_1 = \mathcal{R}(S)$ where $S \to \mathcal{R}(S)$. Given that $t_1 = u_1$, we have $t_1 = u_1|_\varepsilon$ and as required $R(\varepsilon)$ must necessarily be q_0.

(ii) A-App. In this case t_m, β and γ are unchanged. From the definition of reify we can see that the fresh α will be resolved to the operand term that was removed: $\mathsf{reify}(\Gamma^o \vdash t\, u : \theta^o) = \mathsf{reify}(\Gamma^o \vdash t : \alpha \to \theta^o)$.

(iii) A-Var. This rule is necessarily followed by a single instance of A-α and then as above the inductive case gives the same term as the new case:

$$\begin{aligned} t_m &= \mathsf{reify}(\Gamma^o \vdash x : \theta^o) \\ &= \mathsf{resolve}(\Gamma^o, x) \\ &= x[\mathsf{intro}(\Gamma^o(x))/x] \\ &= v \\ &= \mathsf{reify}(\Gamma_1^o \vdash v : \theta^o). \end{aligned}$$

(iv) A-Fun. From the induction hypothesis there is a witness of the form $t_m = F\,\tilde{v}$. The new t_{m+1} is

$$\mathsf{reify}(\{x_v : \alpha_v \mid v \in \tilde{v}\} \vdash s : q) = s[\tilde{v}/\tilde{x}]$$

where $F\,\tilde{v} \to s[\tilde{v}/\tilde{x}]$. The inductive $t_m = u_m|_\gamma$ and t_{m+1} is a subterm of u_{m+1} (where $u_m \to u_{m+1}$ by reducing the redex t_m). Then if $S \to^m u_m$, $S \to^{m+1} u_{m+1}$ and $u_{m+1}|_\gamma = t_{m+1}$. As we are still talking about the same location in the run tree (β), it must still be labelled (γ, q), where q is the type in the new judgement.

(v) A-Cst. Here we progress the constraints on the run tree. The induction hypothesis gives us $t_m = \mathsf{reify}(\Gamma^o \vdash a : \alpha_1 \to \cdots \to \alpha_k \to q)$ as a witness. If $(q, a) \notin \delta$ we are trivially done. Otherwise $\delta(q, a) = q_1 \cdots q_k$ and A-α creates k new judgements of the form $\Gamma_1^o \vdash \mathsf{resolve}(\Gamma^o, \alpha_i) : q_i$; applying reify to each gives a subterm of t_m which will be $u_m|_{\gamma i}$. From the definition of a value tree and $S \to^* u_m$, $[\![\mathcal{G}]\!](\gamma) = a$. From the definition of a run tree, necessarily for each $i \in \{1, \cdots, k\}$, $R(\beta\, i) = (\gamma\, i, q_i)$ and we are done.

□

To ensure termination of the algorithm we need to show that the tree of open derivations $\mathcal{D}$ does not have an infinite width. If this holds then it always increases in depth and so every path in the tree will eventually see a duplicate reified type binding as required for a complete cut. To achieve this we first show that we always make progress, either in uncovering more of the known prefix of the value tree or in reduction of the HORS.

Lemma 3.5 (Eventually A-Cst or A-Fun)**.** *From any open judgement* $J = \Gamma^o \vdash t : \theta^o$*, continuing the algorithm eventually leads to using either* A-Cst *or* A-Fun*.*

Proof. We observe that the other two operations reduce the size of judgement for a particular notion of size. Given J, we set J' to be closest ancestor of J such that the parent of J' is not the leftmost premise in a use of App. Note that this just moves back over the most recent uses of A-App and it is possible that $J = J'$. Then $\mathsf{reifycost}(J')$ is the number of function invocations used to calculate $\mathsf{reify}(J')$. Using a lexicographic ordering over pairs, the size of J is defined as:

$$|J| := (\mathsf{reifycost}(J'), |\mathsf{tm}(J)|)$$

(i) A-App: The original judgement is $J = \Gamma^o \vdash t\,u : \theta^o$ and the new judgement is $J_1 = \Gamma^o \vdash t : \alpha \to \theta^o$. J' is unchanged, and therefore so is $\mathsf{reifycost}(J')$. The size of the term is reduced however and so $|J_1| < |J|$.

(ii) A-Var: The original judgement is $J = \Gamma^o \vdash x : \theta^o$ and the new judgement is $J_1 = \Gamma_1^o \vdash u : \theta^o$. From the definition of reify, we can see that one fewer function invocation is required to calculate $\mathsf{reify}(J_1)$ (that required to look up $\mathsf{intro}(\Gamma^o(x))$) and again $|J_1| < |J|$.

The order is well-founded, so we are done. □

The final piece required to show completeness is to ensure that the tree is of finite width. If this were not the case, then we would not be able to use the finiteness of the space of intersection types to guarantee that we would eventually find a complete cut. This requires a tricky argument and the use of a technical lemma due to Kobayashi and Ong [2009, Lemma 4.8, Appendix B], which states that any infinite sequence of reduction in a HORS must contain an infinite chain of reduction of non-terminals that have been introduced by the reduction of a non-terminal earlier in the chain.

Lemma 3.6. *If there exists a finite* N *such that all paths in the state tree* $\mathcal{D}$ *induced by a path of computation are bounded by* N*, then this path of computation is finite.*

Proof. We first require a notion of well-foundedness of reduction sequences. We take $\twoheadrightarrow$ to be the union of the standard HORS reduction $\rightarrow_{\mathcal{G}}$ with the relation

$$\{(a\, s_1 \dots s_{\mathsf{arity}(a)}, s_i) \mid a \in \Sigma, 1 \leqslant i \leqslant \mathsf{arity}(a)\}.$$

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

$$\tau \,:\, S = u_1 \twoheadrightarrow u_2 \twoheadrightarrow u_3 \twoheadrightarrow \cdots$$

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

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

$$F_i^\ell > F_j^{\ell'} \text{ just if the head symbol of } u_{\ell'} \text{ is } F_i^\ell$$

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

$$\underbrace{(F^\ell \tilde{u})}_{u^{\ell_1}} \twoheadrightarrow (C_1[F^{\ell_1}])[\tilde{u}/\tilde{x}] \twoheadrightarrow^* \underbrace{(F^{\ell_1} \tilde{v})}_{u^{\ell_2}} \twoheadrightarrow (C_2[F^{\ell_2}])[\tilde{v}/\tilde{x}] \twoheadrightarrow \dots$$

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

$$S = F_1 > F_{i_1}^{\ell_1} > F_{i_2}^{\ell_2} > \cdots$$

We now take a particular (possibly infinite) path of computation $\tau' = J_1 J_2 \cdots$. By assumption, τ' does not induce any infinite paths in the state tree $\mathcal{D}$. From Lemma 3.4, τ' witnesses a reduction sequence $\tau \,:\, S = u_1 \twoheadrightarrow u_2 \twoheadrightarrow \cdots$.

Notice that for each pair $F_i^\ell > F_j^{\ell'}$ from τ where $u_{\ell'} = F_i^\ell\ \tilde{s}$ then there is some judgement J_k such that $\mathsf{reify}(J_k) = u_{\ell'}$ (modulo colouring). Then J_{k+1} is necessarily the root of a new typing derivation so that there also exists a pair of nodes of $\mathcal{D}$: $n < n\,i$ (so '$n\,i$' is the i'th child of 'n'). The reverse also holds. Critically, 'n' is the parent of '$n\,i$,' so that any infinite sequence $S = F_1 > F_{i_1}^{\ell_1} > F_{i_2}^{\ell_2} > \cdots$ would imply the existence of an infinite path in $\mathcal{D}$. As a result τ' being well-founded implies that τ also is.

Kobayashi and Ong [2009, Lemma 4.8, Appendix B] showed that if τ is well founded, then necessarily it is finite. We can see that τ, being obtained from τ' via Lemma 3.4, has length determined by the number of uses of A-Fun and A-Cst. As τ is finite, if τ' were infinite it would end with an infinite sequence of uses of A-Var and A-App. This would contradict Lemma 3.5 and so τ' must be finite. □

The correctness of the algorithm now follows as a natural result of the preceding lemmas.

Theorem 3.7 (Correctness)**.** *Let $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$ be a HORS and $\mathcal{A} = \langle \Sigma, Q, \delta, q_0\rangle$ a DTT.*

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

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

(iii) Algorithm 1 terminates on every input.

Proof. (i) Immediate from Lemma 3.3.

(ii) Assume for contradiction that the algorithm returns No, but $\mathcal{A}^\perp$ accepts $[\![\mathcal{G}]\!]$ so that there must exist a run-tree R. As the algorithm returns No, there must have been a judgement $\Gamma^o \vdash a : \theta^o$ where $a \in \Sigma$, $\mathsf{state}(\theta^o) = q$ and $(q, a) \notin \delta$. From Lemma 3.4, we know that there is a word ζ, γ such that $[\![\mathcal{G}]\!](\gamma) = a$ and $R(\zeta) = (\gamma, q)$. The definition of a run tree requires that there must be a satisfying assignment for $\delta(q, a)$; as there is not we have our contradiction.

(iii) Let N be the product of the number of non-terminals of $\mathcal{G}$ and the total number of types (of the corresponding kinds) of $\vdash_{\mathcal{A}}$. By following the computation of the algorithm until every open judgement is at the end of a path in $\mathcal{D}$ of length N, we are guaranteed that every path either ends in a closed leaf or contains a recurrence of a reified type binding. As such, there must be a complete cut. Termination of this process is guaranteed by Lemma 3.6, an analogue of the *spinal decomposition* lemma of Ong [2006, Lemma 14 (long version)].

□

3.3 An Exact Correspondence

In this section we will show that the sequence of instructions carried out by the algorithm are isomorphic to traversals, and that a cosmetically modified version of the algorithm produces a justified judgement tree that is isomorphic to the traversal tree with respect to both the successor and pointer relations. We will formalise the definition of traversals and give another example to aid the reader's intuition.

We start with the *long transform*, shown in Example 3.1.

Definition 3.11 (Long transform)**.** The *long transform* $\overline{\mathcal{G}}$ of HORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ is a regular grammar (equivalently an order-0 HORS) that generates the possibly infinite unfolding of the syntax of the η-long normal form of $\mathcal{G}$.

First we inductively define an η-long transform, expanding only those terms in *operand position* by $\ulcorner \cdot \urcorner$:

$$\ulcorner \xi\ s_1 \cdots\ s_n \urcorner = \lambda x_1 \cdots x_m . X \ulcorner s_1 \urcorner \cdots \ulcorner s_n \urcorner \ulcorner x_1 \urcorner \cdots \ulcorner x_m \urcorner$$

where

$$\xi\ s_1 \cdots\ s_n : \kappa_1 \to \cdots \to \kappa_m \to o$$

$$X = \begin{cases} @_\kappa\ \xi & \xi : \kappa \in \mathcal{N} \\ \xi & \text{otherwise} \end{cases}$$

In the image of the long transform, '$\lambda x_1 \cdots x_n .$' is *not* an abstraction, but rather an atomic terminal symbol representing the syntactic occurrence of an abstraction. We require that all bound variables are initially distinct, and that $\ulcorner \cdot \urcorner$ only introduces fresh variables. Then $\overline{\mathcal{G}} = \langle \Sigma', \mathcal{N}', \mathcal{R}', S \rangle$ where:

$$\begin{aligned}
\mathcal{N}' &= \{F : o \mid F : \kappa \in \mathcal{N}\} \\
\mathcal{R}' &= \{(F, \lambda x_1 \cdots x_n . \ulcorner t \urcorner) \mid \mathcal{R}(F) = \lambda x_1 \cdots x_n . t\} \\
\Sigma' &\subseteq \Sigma \cup \{x_i \mapsto \mathsf{arity}(\kappa_i) \mid x_i \in \mathcal{V}, \mathcal{R}(F) = \lambda x_1 \cdots x_n . t, \\
&\qquad\qquad F : \kappa_1 \to \cdots \to \kappa_n \to o \in \mathcal{N}\} \\
&\quad \cup \{y_j \mapsto \mathsf{arity}(\kappa_j) \mid y_j \text{ introduced in } \ulcorner t \urcorner, t : \kappa_1 \to \cdots \to \kappa_m \to o\} \\
&\quad \cup \{\lambda x_1 \cdots x_n . \mapsto 1 \mid \{x_1, \cdots, x_n\} \subseteq \mathcal{V}\} \\
&\quad \cup \{@_\kappa \mapsto \mathsf{arity}(\kappa) + 1 \mid F : \kappa \in \mathcal{N}\}
\end{aligned}$$

and we take Σ' to be the smallest set satisfying the above inequality such that every symbol in $\mathcal{R}$ may be found in $\Sigma \uplus \mathcal{N}$. For brevity we tend to elide the type tag κ from @.

Example 3.12 (Examples of long transforms)**.** We first recapitulate the transform of $\mathcal{G}_1$:

$$\mathcal{G}_1 \begin{cases} S = F\ c \\ F = \lambda x.a\ x\ (F\ (b\ x)) \end{cases} \quad \mapsto \quad \overline{\mathcal{G}_1} \begin{cases} S = \boldsymbol{\lambda}.@\ F\ (\boldsymbol{\lambda}.c) \\ F = \lambda x.a\ (\boldsymbol{\lambda}.x)\ (\boldsymbol{\lambda}.@\ F\ (\boldsymbol{\lambda}.b\ (\boldsymbol{\lambda}.x))) \end{cases}$$

We will also give the long transform for an order-2 scheme. The behaviour of traversals (and by extension, Algorithm 1) is much more interesting above order-1. Given $\Sigma_3 = \{a \mapsto 0, g \mapsto 2, h \mapsto 1\}$ we define $\mathcal{G}_3$ and its long transform as:

$$\mathcal{G}_3 = \langle \Sigma_3, \{S : o, H : o \to o, F : (o \to o) \to o\}, \mathcal{R}_3, S \rangle$$

$$\mathcal{R}_3 \begin{cases} S = H\ a \\ H = \lambda z.F\ (g\ z) \\ F = \lambda w.w\ (w\ (F\ h)) \end{cases} \quad \mapsto \quad \overline{\mathcal{G}_3} \begin{cases} S = \boldsymbol{\lambda}.@\ H\ (\boldsymbol{\lambda}.a) \\ H = \lambda z.@\ F\ (\lambda y.g\ (\boldsymbol{\lambda}.z)\ (\boldsymbol{\lambda}.z)) \\ F = \lambda w.w\ (\boldsymbol{\lambda}.w\ (\boldsymbol{\lambda}.@\ F(\lambda x.h\ (\boldsymbol{\lambda}.x)))) \end{cases}$$

Definition 3.13 (Computation tree)**.** Given a HORS $\mathcal{G}$, the *computation tree* $\lambda(\mathcal{G})$ is a regular tree generated by unfolding the rules of the long transform $\overline{\mathcal{G}}$ *ad infinitum*. As noted above, $\overline{\mathcal{G}}$ is an order-0 HORS and $\lambda(\mathcal{G}) = [\![\overline{\mathcal{G}}]\!]$.

The computation tree of $\mathcal{G}_1$ can be seen in Figure 3.2 and the value tree and computation tree of $\mathcal{G}_3$ in Figure 3.3. Notice how the structure of the tree mirrors the syntax of the terms in the rewrite rules.

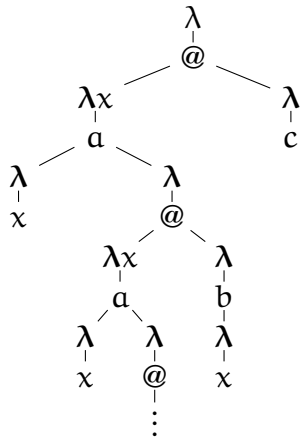


Figure 3.2: Computation tree $\lambda(\mathcal{G}_1)$

In the computation tree, the root, '@' nodes, and $a\ (\in \Sigma)$ nodes are considered to be *initial*. Every non-initial node is *enabled* by some other node:

- each variable node x_i is said to be i-enabled by its *binder*, labelled $\boldsymbol{\lambda}\widetilde{x}$, where x is the i-th element of the list $\widetilde{x}$.

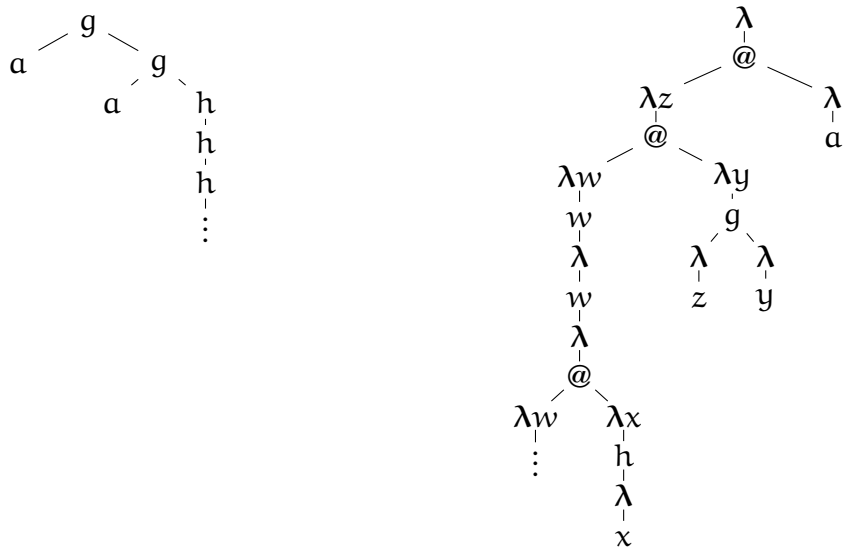


Figure 3.3: The value tree $[\![\mathcal{G}_3]\!]$ and computation tree $\lambda(\mathcal{G}_3)$

- any other non-initial node is said to be i-enabled by its parent, just if it is the i-th child.

In order to describe traversals over the computation tree, we require a notion of *justified sequences*. If a sequence contains m, i-enabled by some earlier node n, then we write:

$$w\ n \overset{i}{\leftarrow} m\ v$$

A justified sequence must satisfy

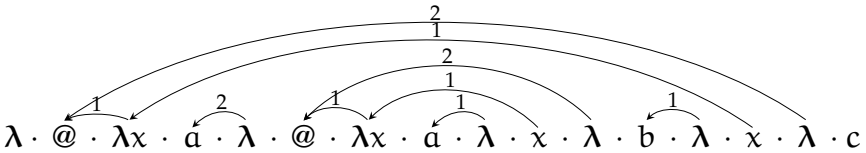
(i) *alternation*: the sequence must alternate between lambda and non-lambda nodes, and

(ii) the *pointer condition*: each non-initial node n must have a pointer to an earlier node n_0 such that n_0 j-enables n, for some j.

Recall the second traversal over $\lambda(\mathcal{G}_1)$ we considered at the start of the chapter:

$$\lambda \cdot @ \cdot \lambda x \cdot a \cdot \lambda \cdot @ \cdot \lambda x \cdot a \cdot \lambda \cdot x \cdot \lambda \cdot b \cdot \lambda \cdot x \cdot \lambda \cdot c$$

This clearly satisfies alternation; for the pointer condition we now add the pointers to non-initial nodes as determined by the definition of i-enabling above. Note that these are completely determined by the computation tree.



Definition 3.14 (P-view)**.** The *P-view* of a justified sequence w is a subsequence of w defined recursively:

$$\begin{aligned} \ulcorner \mathbf{\lambda} \urcorner &= \mathbf{\lambda} \\ \ulcorner w\, n \cdots \overset{i}{\leftarrow} \mathbf{\lambda}\tilde{x} \urcorner &= \ulcorner w \urcorner\, n \overset{i}{\leftarrow} \mathbf{\lambda}\tilde{x} \\ \ulcorner w\, \mathbf{\lambda}\tilde{x}\, n \urcorner &= \ulcorner w\, \mathbf{\lambda}\tilde{x} \urcorner\, n \end{aligned}$$

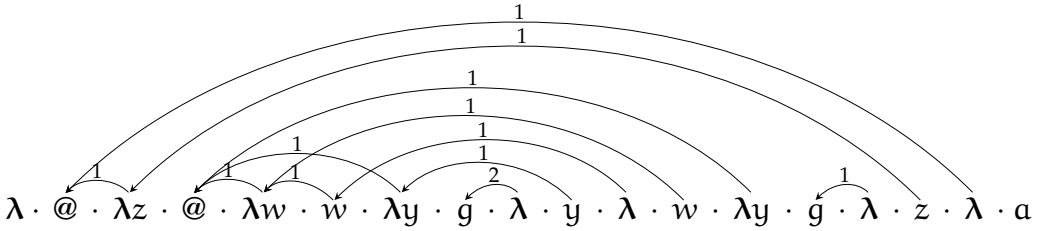
Note that in the last two rules n may have a pointer to a node in w (resp. $w\,\mathbf{\lambda}\tilde{x}$). If this node is found in $\ulcorner w \urcorner$ (resp. $\ulcorner w\,\mathbf{\lambda}\tilde{x} \urcorner$) then the pointer is preserved, otherwise it is removed.

The P-view captures the scope at a given point in the traversal. For example, the traversal above has a P-view of just $\mathbf{\lambda} \cdot c$, as none of the earlier computation is relevant once we have returned to the calling scope.

Definition 3.15 (Traversals of computation trees)**.** Traversals are defined by induction over the following rules:

- (Root). ε is a traversal.
- (App). If $w\,@$ is a traversal, then so is $w \overset{0}{\leftarrow} @\ \mathbf{\lambda}\tilde{x}$.
- (Sig). If $w\,a$ is a traversal, then so is $w \overset{i}{\leftarrow} a\ \mathbf{\lambda}$ for each $i \in [1..\mathsf{arity}(a)]$.
- (Var). If $w\,n\,\mathbf{\lambda}\tilde{x} \overset{i}{\leftarrow} \cdots x$ is a traversal, then so is $w\,n\,\mathbf{\lambda}\tilde{x} \overset{i}{\leftarrow} \cdots x\,\mathbf{\lambda}\tilde{y}$.
- (Lam). If $w\,\mathbf{\lambda}\tilde{x}$ is a traversal and $\ulcorner w\,\mathbf{\lambda}\tilde{x}\, n \urcorner$ is a path in $\lambda(\mathcal{G})$, then $w\,\mathbf{\lambda}\tilde{x}\, n$ is a traversal.

Example 3.16 (Traversal over $\lambda(\mathcal{G}_3)$)**.** We will now give a full example of an order-2 traversal, which exposes more of the structure of traversals. In the value tree, $[\![\mathcal{G}_3]\!]$, there is a path $g\ g\ a$ and we consider the corresponding traversal in Figure 3.4 (described by the annotations 0–18). We can see that the traversal satisfies alternation and the pointer condition from the flattened version below:



The reduction here gives rise to more complex "jumping". For example, when the order-1, arity 2 variable w is applied, we jump back to the argument of F, which is $\mathsf{g}\ \mathsf{z}$ in the original scheme, now represented by the subtree rooted at $\lambda\mathsf{y}$. The η-expansion has caused the second argument of w to be represented explicitly by the new variable y, and so if we choose to inspect the second argument of g we must jump back to the child of w, which represents the operand $w\ (\mathsf{F}\ \mathsf{h})$.

P-views are correspondingly more complex: an example is that of the prefix of this traversal up to and including the node labelled y and annotated with 10 in Figure 3.4:

$$\ulcorner \lambda \cdot @ \cdot \lambda z \cdot @ \cdot \lambda w \cdot w \cdot \lambda y \cdot g \cdot \lambda \cdot y \urcorner = \lambda \cdot \overset{1}{@} \cdot \lambda z \cdot \overset{1}{@} \cdot \lambda y \cdot \overset{2}{g} \cdot \lambda \cdot \overset{1}{y}$$

Notice that both the P-views considered so far constitute paths in the computation tree, and in fact this holds in general.

Proposition 3.8 ([Ong, 2006, Proposition 6])**.** *A sequence satisfies* P-visibility *just if every non-lambda node is justified by a lambda node in its P-view. Among other properties, traversals*

(i) are well-defined justified sequences that satisfy P-visibility, and
(ii) have a P-view that is a path in the computation tree.

To ease comparison between the algorithm computation and this traversal, a partial computation of the algorithm is shown in Figure 3.5. For simplicity a single-state automaton is used ($\langle \Sigma_3, \{q\}, \{(q, a) \mapsto q \cdots q \mid a \in \Sigma_3\}, q\rangle$) and non-terminal types are elided from the open type environments. This shows just the control flow of the algorithm (again annotated by ascending integers), which is not affected by the non-terminal types, and the structure of the open type derivations. In the figure, straight lines are used where the A-Fun rule has been used and curved lines where A-α has been used. We can see certain features of the traversal clearly in the computation of the algorithm, such as the two uses of the w variable, and the corresponding jumps back to the operand $(\mathsf{g}\ \mathsf{z})$ in the right-hand side of H. Where traversals pass to the left child of an @-labelled node, the model checking algorithm correspondingly creates a new typing derivation. There is a difference due to the η-expansion, however, which is handled implicitly. The direct jump between the judgements labelled 10 and 11 in Figure 3.5 corresponds to the sequence 8–12 in the traversal in Figure 3.4 due to the extra lambdas. Defining the model checking algorithm with strict alternation over η-expanded terms would impair the performance.

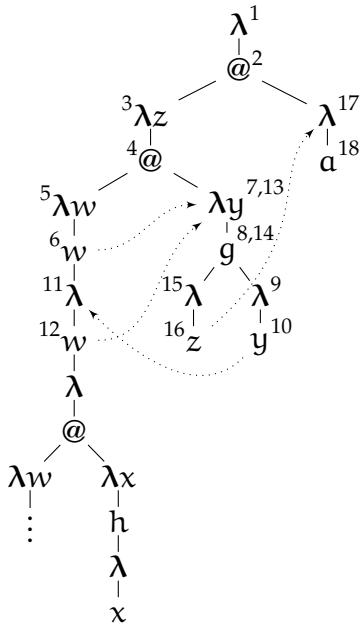


Figure 3.4: A traversal over $\lambda(\mathcal{G}_3)$

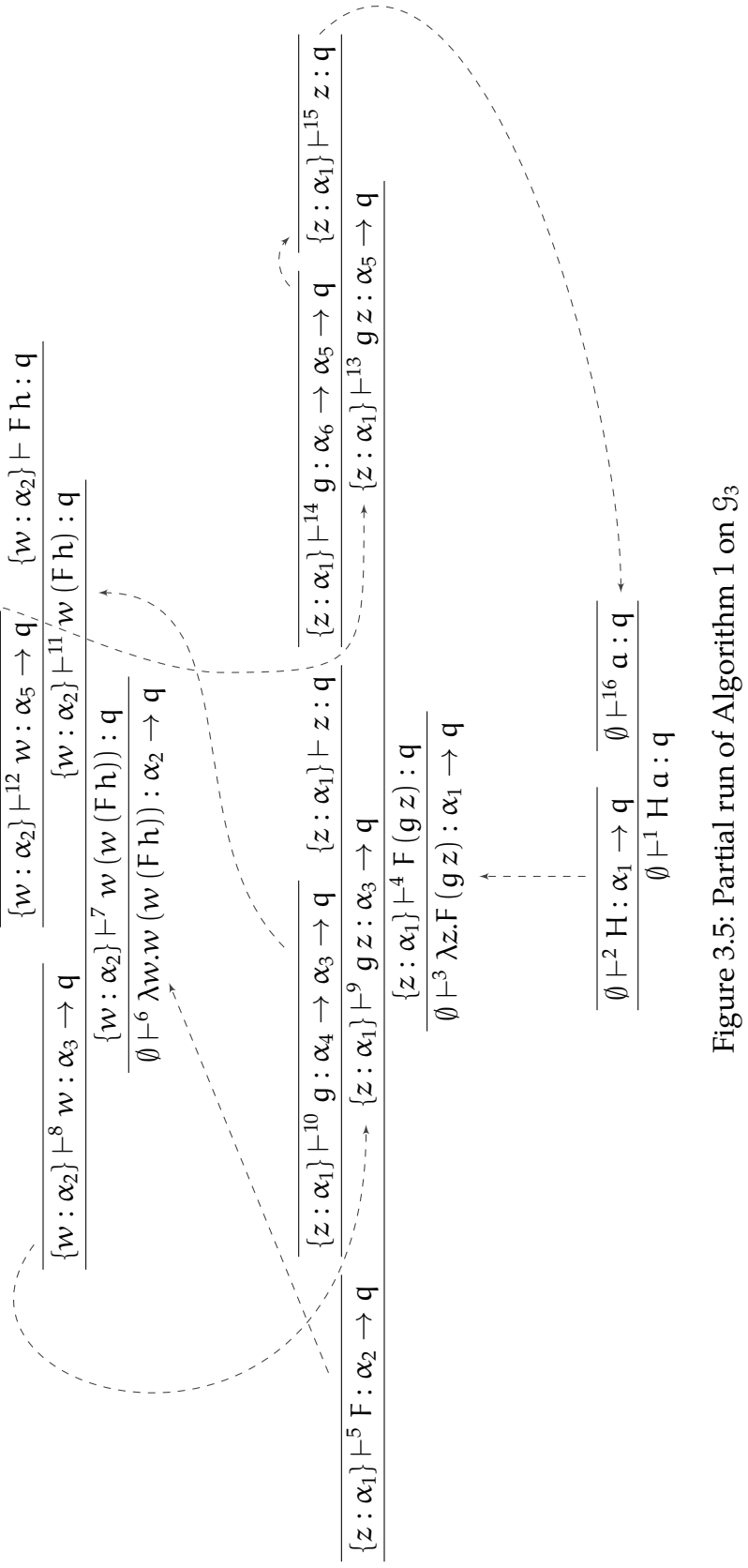


Figure 3.5: Partial run of Algorithm 1 on $\mathcal{G}_3$

In order to show a precise correspondence between traversals and the computation of the algorithm, we consider an equivalent formulation of the algorithm rules that operates on HORS in η-long normal form. In Table 3.4 the head symbol of a term, ξ, ranges over terminals (Σ), non-terminals ($\mathcal{N}$) and variables ($\mathcal{V}$). Long abstraction and application rules are used in this new set of rules for uniformity and are simply derived from multiple uses of the standard rules.

As well as the derivation tree structure of $\mathcal{D}$, the computation of the algorithm can be considered to define a tree of judgements. For this purpose we consider $\mathsf{history}(J)$ to include only those judgements that were at some time unjustified. In this presentation of the algorithm this excludes the intermediate judgement with term $\xi\, t_1 \cdots t_n$ in rule A-Abs.

Notice that in η-long normal form, the algorithm alternates between judgements having terms in *Lambda form* ($\lambda\widetilde{x}.t$) and *Non-Lambda form* (ξ). This is a natural consequence of the transform presented in Definition 3.11, which uses 'dummy lambdas' to ensure this property so that an order-0 term t will be transformed to $\lambda.t$. For any history sequence, the first element (being the root judgement of $\mathcal{D}$) has no pointer. For the rest of the sequence, we use a case analysis over the form of the judgement and its head symbol:

1. Lambda form: $J_i = \Gamma^o \vdash \lambda\widetilde{x}.\xi\, t_1 \cdots t_n : \theta^o$, so that $J_{i+1} = \Gamma_1^o \vdash \xi : \theta_1^o$.
   a) If $\xi \in \mathcal{N} \cup \Sigma$, then J_{i+1} has no pointer.
   b) If $\xi \in \mathcal{V}$, say $\xi = y_j$, then J_{i+1} points to $J' = \Gamma_2^o \vdash \lambda y_1 \cdots y_m \cdot s : \theta_2^o$ in the same open derivation.
2. Non-Lambda form: $J_i = \Gamma^o \vdash \xi : \Theta^o$
   a) If $\xi \in \mathcal{N}$, then $J_{i+1} = \emptyset \vdash \mathcal{R}(F) : \theta^o$ and points to J_i.
   b) If $\xi \in \Sigma \cup \mathcal{V}$, then J_{i+1} will be created by A-α and will point to its leftmost sibling judgement. Note that this judgement was created at the same time as the α that whose modification triggered the A-α rule.

We aim to show that the tree induced in this way by the computation of the algorithm is isomorphic to the traversal tree with respect to the successor and the pointer relations.

Theorem 3.9 (Correspondence)**.** *In any state of the algorithm, for each unjustified judgement* J, $\mathsf{history}(J)$ *determines a unique traversal over the computation tree* $\lambda(\mathcal{G})$.

Proof. We proceed by induction. In the base case $J = \emptyset \vdash \mathcal{R}(S) : q_0$, the corresponding traversal is just $\mathbf{\lambda}$ and neither node has a pointer. For the inductive step we case split as before:

Rule	If	Then
L-Abs	(i) $\Gamma^o \vdash \lambda x_1 \cdots x_m.\xi\, t_1 \cdots t_n : \theta^o$ (ii) $\theta^o = \alpha_1 \to \cdots \to \alpha_m \to q$ (iii) $\forall i \in [1..n] \cdot \beta_i$ fresh	(i) $\dfrac{\Gamma_1^o \vdash \xi : \beta_1 \to \cdots \to \beta_n \to q}{\dfrac{\Gamma_1^o \vdash \xi\, t_1 \cdots t_n : q}{\Gamma^o \vdash \lambda x_1 \cdots x_m.\xi\, t_1 \cdots t_n : \theta^o}}$ (ii) $\Gamma_1^o = \Gamma^o \cup \{x_i : \alpha_i \mid 1 \leqslant i \leqslant m\}$
L-Fun	(i) $J = \Gamma^o \vdash F : \theta^o \quad (\in \Delta^o)$ (ii) $\mathcal{R}(F) = \lambda x_1 \ldots x_n.s$ (iii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$	(i) $\dfrac{}{\Gamma^o, F : \theta^o \vdash F : \theta^o}$ (ii) For $\Gamma_1^o \vdash s : \theta_1^o$ in Δ^o, $\Gamma_1^o := \Gamma_1^o \cup \{F : \theta^o\}$ (iii) Add new rightmost child to Δ^o: $\emptyset \vdash \mathcal{R}(F) : \theta^o$
L-Var	(i) $\Gamma^o, x : \alpha \vdash x : \theta^o$	(i) $\dfrac{}{\Gamma^o, x : \alpha \vdash x : \theta^o}$ (ii) $\Theta(\alpha) := \Theta(\alpha) \cup \{\theta^o\}$
L-Cst	(i) $\Gamma^o \vdash a : \theta^o$ (ii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$ (iii) $\delta(q, a) = q_1 \cdots q_n$	(i) $\dfrac{}{\Gamma^o \vdash a : \alpha_1 \to \cdots \to \alpha_n \to q}$ (ii) $\forall i \in [1..n] \cdot \Theta(\alpha_i) := \Theta(\alpha_i) \cup \{q_i\}$
L-α	(i) $\dfrac{\dfrac{}{\Gamma^o \vdash \xi : \theta_1^o} \quad \cdots}{J = \Gamma^o \vdash \xi\, t_1 \cdots t_n : \theta_1^o}$ (ii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$ (iii) $\exists i \in [1..n] \cdot \theta^o \in \Theta(\alpha_i)$ (iv) $\Gamma^o \vdash t_i : \theta^o$ not a child of J	(i) $\dfrac{\dfrac{}{\Gamma^o \vdash \xi : \theta_1^o} \quad \Gamma^o \vdash t_i : \theta^o \quad \cdots}{\Gamma^o \vdash \xi\, t_1 \cdots t_n : q}$

Table 3.4: Rules of algorithm

1. Lambda form: $J_i = \Gamma^o \vdash \lambda\widetilde{x}.\xi\, t_1 \cdots t_n : \theta^o$, so that $J_{i+1} = \Gamma_1^o \vdash \xi : \theta_1^o$. The determined traversal node is $\boldsymbol{\lambda}\widetilde{x}$.

   a) If $\xi \in \mathcal{N}$ (where $\xi : \kappa$) then J_{i+1} determines the traversal node $@_\kappa$, which does not have a pointer.

   b) If $\xi \in \Sigma$, then J_{i+1} determines the traversal node ξ, which has no pointer.

   c) If $\xi \in \mathcal{V}$, say $\xi = y_j$, then J_{i+1} determines traversal node ξ, which points to the traversal node determined by $J' = \Gamma_2^o \vdash \lambda y_1 \cdots y_m \cdot s : \theta_2^o$ in the same open derivation. Notice that this will be $\boldsymbol{\lambda} y_1 \cdots y_m.$, i.e. the binder of y_j.

2. Non-Lambda form: $J_i = \Gamma^o \vdash \xi : \theta^o$, which determines the traversal node $@_\kappa$ if $\xi : \kappa \in \mathcal{N}$ and ξ otherwise.

   a) If $\xi \in \mathcal{N}$, then $J_{i+1} = \emptyset \vdash \mathcal{R}(F) : \theta^o$ where $\mathcal{R}(F) = \lambda\widetilde{x}.t$ determines the traversal node $\boldsymbol{\lambda}\widetilde{x}.$, which points to the node determined by J_i ($@_\kappa$) as required.

   b) If $\xi \in \Sigma$, then J_{i+1} can be one of $\mathsf{arity}(a)$ judgements, just as the traversal node ξ has $\mathsf{arity}(a)$ possible extensions. As each argument of a terminal must be order 0, $J_{i+1} = \Gamma^o \vdash \lambda.t : q$, and points at J, just as the determined traversal node $\boldsymbol{\lambda}$ points at ξ.

   c) If $\xi \in \mathcal{V}$, then J_i points to the judgement J' containing ξ's binder in the same open derivation. From the definition of A-Abs J' will be the successor of the leftmost sibling of J_{i+1}, which J_{i+1} will point to. $J_{i+1} = \Gamma_1^o \vdash \lambda\widetilde{x}.s : \theta^o$ determines the traversal node $\boldsymbol{\lambda}\widetilde{x}.$, which by definition points to the predecessor of that determined by J' as required.

□

Corollary 3.10 (Tree isomorphism)**.** *The trees induced by traversals over $\lambda(\mathcal{G})$ and histories of computation of the algorithm are isomorphic.*

Chapter 4

Recursion Schemes with Cases

We have shown how the idea of traversals gives rise to a model checking algorithm for HORS. In this chapter we consider a richer model of computation, *higher-order recursion schemes with cases* (HORSC), and argue that they are more suitable as a target of abstraction than plain HORS. HORSC extend HORS with a definition-by-cases construct (to express program control flow based on data) and non-determinism (to express abstraction of behaviours). We will introduce a novel intersection and union type system for characterising the HORSC model checking problem, and an extension to our traversal-based algorithm from Chapter 3. Finally, we test our claim that HORSC are a suitable target for abstraction using an implementation of our algorithm — TravMC — to model check the output of an abstraction-refinement procedure [Ong and Ramsay, 2011] and compare with other tools for HORS model checking. This work was carried out jointly with Ong and Ramsay and published in the proceedings of ICFP 2012 [Neatherway et al., 2012]. I was responsible for the development of HORSC, the type system that characterises the model checking problem, the algorithm and its implementation. Ong again helped with the proof effort and Ramsay's implementation of his abstraction-refinement procedure was used to generate large input instances for benchmarking.

4.1 Abstract Models with Data

We are interested in providing backend model checking algorithms for procedures that compute sound abstractions of higher-order functional programs. In the first-order world, abstraction procedures for programs written in imperative languages such as C will often produce over-approximations in the form of Boolean programs with non-deterministic choice. HORS do not include a native notion of data (although it can be simulated using the power of higher-order functions), and

so far HORS model checkers in the literature consider only deterministic HORS. Encoding these features into plain HORS causes a number of problems including raising the order and arity of the scheme and obfuscating the control flow and structure of the abstracted program.

Example 4.1. The *Risers* program from Mitchell and Runciman [2008] provides an interesting example of a program with partial pattern matching that cannot crash:

$$
\begin{aligned}
&\mathsf{risers}\ [] = [] \\
&\mathsf{risers}\ [x] = [[x]] \\
&\mathsf{risers}\ (x : y : etc) = \mathbf{if}\ x \leqslant y\ \mathbf{then}\ (x : s) : ss\ \mathbf{else}\ [x] : (s : ss) \\
&\qquad \mathbf{where}\ (s : ss) = \mathsf{risers}\ (y : etc)
\end{aligned}
$$

In this program the partial pattern match is found in the left-hand side of the **where** clause where it is assumed that risers will not return an empty list. The property of *pattern-match safety* is a general one that we would like to hold of any program we write – that any partial pattern match will never be applied to an argument corresponding to one of the missing cases. It is not obvious at a glance that this program is safe in this sense from the definition, but if we look carefully we can see that both branches of the **if** expression in risers return a non-empty list as required by the destruction in the **where** clause.

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). To ease comparison with the original program, we represent the abstraction here using a Haskell-like syntax with non-deterministic choice. We treat the **if** expression non-deterministically and the pattern-matching is applied to a single data type that could be declared as $\mathsf{data\ Bool} = \mathsf{Nil} \mid \mathsf{Cons}_1 \mid \mathsf{Cons}_2$:

$$
\begin{aligned}
&\mathsf{risers\ Nil} = \mathsf{Nil} \\
&\mathsf{risers\ Cons}_1 = \mathsf{Cons}_1 \\
&\mathsf{risers\ Cons}_2 = \mathsf{ifthenelse\ (destruct\ Cons}_2) \\
&\mathsf{ifthenelse}\ yetc = \mathsf{cons\ (where}\ yetc) \\
&\mathsf{ifthenelse}\ yetc = \mathsf{cons\ (cons\ (where}\ yetc)) \\
&\mathsf{where}\ yetc = \mathsf{destruct\ (risers}\ yetc) \\
&\mathsf{destruct\ Nil} = \mathsf{error} \\
&\mathsf{destruct\ Cons}_1 = \mathsf{Nil} \\
&\mathsf{destruct\ Cons}_2 = \mathsf{Cons}_1 \\
&\mathsf{destruct\ Cons}_2 = \mathsf{Cons}_2 \\
&\mathsf{cons\ Nil} = \mathsf{Cons}_1 \\
&\mathsf{cons\ Cons}_1 = \mathsf{Cons}_2 \\
&\mathsf{cons\ Cons}_2 = \mathsf{Cons}_2
\end{aligned}
$$

The pattern match error is preserved – it occurs if the where function tries to destruct an empty list returned by risers. Furthermore, the pattern-match safety of the original program has been preserved by the abstraction. In the following material, we will use a new syntax that draws on the existing HORS theory for convenience rather than one inspired by Haskell.

Syntax of HORSC

As we only deal with atomic data items (order-0 constructors), rather than using pattern matching as above, we introduce a case construct.

Definition 4.2 (Terms with cases)**.** In the context of a finite domain of size n, we extend our standard grammar with an additional case clause. The notions of free variables and of concrete prefixes of terms must each be correspondingly updated with an extra rule:

$$s, t ::= a \mid x \mid F \mid s\ t \mid \lambda x.t \mid \mathsf{case}(e, e_1, \cdots, e_n)$$

$$\mathsf{FV}(\mathsf{case}(e, e_1, \cdots, e_n)) = \mathsf{FV}(e) \cup \bigcup_{1 \leqslant i \leqslant n} \mathsf{FV}(e_i)$$

$$\mathsf{case}(e, e_1, \cdots, e_n)^{\perp} = \perp$$

We refer to the first argument of a case clause (e in the grammar above) as the *scrutinee* and the remaining arguments as the *choice terms*.

Recall that for HORS we use only a single base kind 'o'. To distinguish between elements of our finite data domain and standard tree constructors, when defining HORSC we additionally use 'd':

$$\frac{\Gamma \vdash e : d \qquad \Gamma \vdash e_1 : \kappa \qquad \cdots \qquad \Gamma \vdash e_n : \kappa \qquad \mathsf{order}(\kappa) = 0}{\Gamma \vdash \mathsf{case}(e, e_1, \cdots, e_n) : \kappa}\ (\text{K-Case})$$

Definition 4.3. A *(non-deterministic) higher-order recursion scheme with cases* (HORSC) is a quadruple $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ similarly to a HORS where:

(i) Σ now contains a distinguished subset of d-kinded symbols, $B = \{ b_1, \ldots, b_n \}$; if $a \in (\Sigma \setminus B)$ then a does not have kind d.

(ii) As for HORS, $\mathcal{N}$ is just a kind environment of non-terminal symbols.

(iii) In order to allow the expression of non-determinism, $\mathcal{R}$ is now a function from non-terminals to a set of closed terms. Note that each non-terminal $F : \kappa \in \mathcal{N}$ has a fixed arity (determined by κ) and so the number of outer abstractions in the right-hand sides of F, $\mathcal{R}(F)$ will be invariant among that set.

(iv) $S \in \mathcal{N}$ is a distinguished 'start' symbol of ground kind, and without loss of generality we assume that $\mathcal{R}(S)$ is a singleton set. By abuse of notation we sometimes write $\mathcal{R}(S)$ 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.

$$\frac{\lambda x_1 \ldots x_m.t \in \mathcal{R}(F)}{F\, s_1 \ldots s_m \to t[\tilde{s}/\tilde{x}]} \qquad \frac{1 \leqslant i \leqslant n}{\mathsf{case}(b_i, s_1, \ldots, s_n) \to s_i} \qquad \frac{s \to s'}{C[s] \to C[s']}$$

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

$$C \quad ::= \quad [\,] \mid C\, s \mid s\, C \mid \mathsf{case}(C, t_1, \ldots, t_n)$$

As before, we write $\to_{\mathcal{G}}^*$ for the reflexive transitive closure. Subterms of the form $F\, t_1 \cdots t_n$ and $\mathsf{case}(s, t_1, \cdots, t_n)$ are *redexes*.

Now that we allow non-determinism, a HORSC will define a *tree language* rather than a single tree. 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 $i' \geqslant i$ such that Δ is contracted in $t_{i'} \to t_{i'+1}$.

Revisiting the *Risers* example, we could express the abstract model above using a HORSC $\mathcal{H}_1 = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where the data items Nil, Cons_1, Cons_2 and an error value (for a pattern match failure) are represented by b_1 to b_4.

$$\Sigma = B = \{b_i \mapsto 0 \mid 1 \leqslant i \leqslant 4\}$$

The functions are represented directly as non-terminals that operate on the finite data domain:

$$\mathcal{N} = \{S : d, \mathit{Risers} : d \to d, \mathit{Ifthenelse} : d \to d, \mathit{Where} : d \to d,$$
$$\mathit{Destruct} : d \to d, \mathit{Cons} : d \to d, \mathit{Input} : d\}$$

and the rewrite rules ($\mathcal{R}$) can now be given in HORSC syntax. The reader may find it a useful exercise to compare with the Haskell-like version in Example 4.1.

$$
\begin{aligned}
S &= \mathit{Risers}\ \mathit{Input} \\
\mathit{Risers}\ \mathtt{xs} &= \mathsf{case}(\mathtt{xs}, \mathsf{b}_1, \mathsf{b}_2, (\mathit{Ifthenelse}\ (\mathit{Destruct}\ \mathsf{b}_3)), \mathsf{b}_4) \\
\mathit{Ifthenelse}\ \mathtt{yetc} &= \mathit{Cons}\ (\mathit{Where}\ \mathtt{yetc}) \\
\mathit{Ifthenelse}\ \mathtt{yetc} &= \mathit{Cons}\ (\mathit{Cons}\ (\mathit{Where}\ \mathtt{yetc})) \\
\mathit{Where}\ \mathtt{yetc} &= \mathit{Destruct}\ (\mathit{Risers}\ \mathtt{yetc}) \\
\mathit{Destruct}\ \mathtt{xs} &= \mathsf{case}(\mathtt{xs}, \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4) \\
\mathit{Destruct}\ \mathtt{xs} &= \mathsf{case}(\mathtt{xs}, \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_3, \mathsf{b}_4) \\
\mathit{Cons}\ \mathtt{xs} &= \mathsf{case}(\mathtt{xs}, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
\mathit{Input} &= \mathsf{b}_1 \\
\mathit{Input} &= \mathsf{b}_2 \\
\mathit{Input} &= \mathsf{b}_3
\end{aligned}
$$

The *Input* non-terminal chooses non-deterministically between empty lists, lists of length one and lists of two or more elements. This allows us to work with a closed program while still modelling all possible ways that the *Risers* function might be invoked. The error atom b_4, representing a pattern-match failure, can only be introduced if *Destruct* is applied to an empty list b_1. As errors are propagated by case expressions, any pattern match error will be visible at the outer level. One possible reduction sequence of $\mathcal{H}_1$ can be seen in Figure 4.1, witnessing the fact that $\mathsf{b}_2 \in [\![\mathcal{H}_1]\!]$.

Alternative representations of data

As we mentioned above, there are other ways to represent data. We mentioned one such in Section 2.3, *recursion schemes with finite data domains* (RSFD). RSFD were introduced by Kobayashi in his work on higher-order, multi-parameter, tree transducers. HORSC extends RSFD in several ways:

(i) The b_i's of HORSC are terminal symbols and may be used freely, whereas the d_i's of RSFD are specifically *data*, distinct from variables, non-terminals and terminals,

(ii) RSFD does not allow non-determinism, and

(iii) In RSFD the return kind of both non-terminals and the case construct must be o.

$$
\begin{aligned}
S &\to \underline{\mathit{Risers}\ \mathit{Input}} \\
&\to \mathsf{case}(\underline{\mathit{Input}}, \mathsf{b}_1, \mathsf{b}_2, (\mathit{Ifthenelse}\ (\mathit{Destruct}\ \mathsf{b}_3)), \mathsf{b}_4) \\
&\to \underline{\mathit{Ifthenelse}\ (\mathit{Destruct}\ \mathsf{b}_3)} \\
&\to \underline{\mathit{Cons}\ (\mathit{Where}\ (\mathit{Destruct}\ \mathsf{b}_3))} \\
&\to \mathsf{case}(\underline{\mathit{Where}\ (\mathit{Destruct}\ \mathsf{b}_3)}, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \mathsf{case}(\underline{\mathit{Destruct}\ (\mathit{Risers}\ (\mathit{Destruct}\ \mathsf{b}_3))}, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \mathsf{case}(\mathsf{case}(\underline{\mathit{Risers}\ (\mathit{Destruct}\ \mathsf{b}_3)}, \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4), \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \mathsf{case}(\mathsf{case}(\mathsf{case}(\underline{\mathit{Destruct}\ \mathsf{b}_3}, \mathsf{b}_1, \mathsf{b}_2, (\mathit{Ifthenelse}\ (\mathit{Destruct}\ \mathsf{b}_3)), \mathsf{b}_4) \\
&\qquad\qquad \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4), \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \mathsf{case}(\mathsf{case}(\mathsf{case}(\underline{\mathsf{case}(\mathsf{b}_3, \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4)}, \mathsf{b}_1, \mathsf{b}_2, (\mathit{Ifthenelse}\ (\mathit{Destruct}\ \mathsf{b}_3)), \mathsf{b}_4), \\
&\qquad\qquad \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4), \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \mathsf{case}(\mathsf{case}(\underline{\mathsf{case}(\mathsf{b}_2, \mathsf{b}_1, \mathsf{b}_2, (\mathit{Ifthenelse}\ (\mathit{Destruct}\ \mathsf{b}_3)), \mathsf{b}_4)}, \\
&\qquad\qquad \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4), \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \mathsf{case}(\underline{\mathsf{case}(\mathsf{b}_2, \mathsf{b}_4, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4)}, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4) \\
&\to \underline{\mathsf{case}(\mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_3, \mathsf{b}_4)} \\
&\to \mathsf{b}_2
\end{aligned}
$$

Figure 4.1: Reduction of $\mathcal{H}_1$

A result of the kinding restriction is that a case construct in an RSFD must take an atomic data item d_i or a variable as its first argument, while in HORSC there is no such restriction. The scrutinee in a HORSC may be any term of kind d, which may reduce to a number of elements of B, or even diverge.

To encode abstractions such as $\mathcal{H}_1$ as an RSFD would require performing a continuation-passing-style (CPS) transform so that terms of kind d would be replaced by terms of kind $(\mathsf{d} \to \mathsf{o}) \to \mathsf{o}$. Non-determinism could also then be removed using an arity-2 br terminal symbol. However, the CPS transform raises the order and arity of the problem instance as described, for example, by Meyer and Wand [1985]. In fact in the worst case, for an order-n HORSC with maximum arity k, the image of the transform will have order-$(2nk - n + 1)$. As explained in Section 2.2 the HORS model checking problem is n-ExpTime complete for order-n schemes, and so we wish to avoid raising the order where possible.

Another possibility is a Church-style encoding, representing the data elements using projection functions. Given a finite domain of size n, element i (where

$1 \leqslant i \leqslant n$) will be represented as non-terminal D_i with kind and rewrite rule:

$$D_i : \underbrace{o \to \cdots \to o}_{n} \to o$$
$$D_i\ x_1 \cdots x_n = x_i$$

This in turn necessitates increasing the order and arity of all the non-terminals in the pre-transform scheme. Where a non-terminal previously had d as part of its kinding e.g. $F : \kappa \to d \to d \in \mathcal{N}$, it must be expanded to $\mathcal{N}(D_i)$:

$$F' : \kappa \to (\underbrace{o \to \cdots \to o}_{n} \to o) \to \underbrace{o \to \cdots \to o}_{n} \to o$$

The raising of the order and arity (which both appear in the hyper-exponent) in this way is also undesirable for a practical solution.

It is possible that a model checking algorithm for HORS may not be vulnerable to the increase in the number of types in this particular way, as many of the new types represent overly conservative or spurious behaviours. However, it is by no means clear, and would require an in-depth analysis for each algorithm. We take a cue from best practice in the design of functional programs and seek to eliminate redundant and illegal states from our representation, thus guaranteeing they will not occur.

Universal HORSC Model Checking Problem

Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ be a DTT. Given a HORSC $\mathcal{G}$ that defines a tree language $[\![\mathcal{G}]\!]$, we say that this language 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 $\mathcal{A}^\perp$. In this work, we are interested in model checking sound approximations of the behaviour of functional programs, and so the universal interpretation is more appropriate. We define the *Universal HORSC Model Checking Problem for DTT* to be to check whether the language $[\![\mathcal{G}]\!]$ is universally accepted by $\mathcal{A}^\perp$ ($[\![\mathcal{G}]\!] \subseteq \mathcal{L}(\mathcal{A}^\perp)$). Henceforth, we will refer to this problem simply as the *HORSC Model Checking Problem*.

For exposition of our approach to the HORSC model checking problem, we will use a small example that nevertheless includes the interesting features of HORSC: data, non-determinism, divergence and higher-order functions.

Example 4.4. The HORSC $\mathcal{H}_2$ is specified by terminal symbols $b_1 :: d$, $b_2 :: d$, $\mathtt{zero} :: o$, $\mathtt{succ} :: o \to o$ and $\mathtt{pred} :: o \to o$; non-terminal symbols $S :: o$, $H :: d$ and

$G :: (o \to o) \to o$, start symbol S and rules:

$$\begin{aligned} S &= \mathsf{case}(H, G\ succ,\ G\ pred) \\ H &= b_1 \\ H &= H \\ G\ g &= g\ zero \end{aligned}$$

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

We will check that the trees generated by $\mathcal{H}_2$ specify positive integers using a simple DTT $\mathcal{A}_4 = \langle \{succ, pred, zero\}, \{q_0, q_1\}, \delta, q_0 \rangle$ where δ is the function:

$$\{(q_0, succ) \mapsto q_1, (q_1, zero) \mapsto \varepsilon\}$$

The run tree of $\mathcal{A}_4$ over $succ\ zero$ is very simple:

$$\begin{array}{l} (\varepsilon, q_0) \\ \quad | \\ (1, q_1) \end{array}$$

4.2 Types with Intersection and Union

In this section we develop a type system parameterised by a DTT that characterises the model checking problem. Here, in order to handle the data and the case construct, we will introduce a carefully restricted form of union types that shows a pleasant symmetry with intersections. Fix $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ with $B \subseteq \Sigma$. We extend our definition of types and *well-kinded types* by the following set of rules:

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

The leftmost rule here is new. Notice that the union operator can only be used at base kind to describe the possible data items a term of kind d may non-deterministically reduce to. The well-kinded types now include expressions such as the types below, with the kinds they refine on the right.

$$\begin{array}{c} (q_1 \wedge q_2 \to (b_1 \vee b_2)) \to q_0 \to b_1 :: (o \to d) \to o \to d \\ (b_1 \vee b_2) \to q_2 :: d \to o \end{array}$$

As before we write $\bigwedge_{i=1}^{k} \theta_i$ for $\bigwedge\{\theta_1, \cdots, \theta_k\}$, and $\top$ for $\bigwedge \emptyset$; we now similarly write $\bigvee_{j=1}^{l} b_{i_j}$ for $\bigvee\{b_{i_1}, \cdots, b_{i_l}\}$ and $\bot$ for $\bigvee \emptyset$. We will often abbreviate the singleton union $\bigvee\{b_i\}$ simply as b_i. The scrutinee of case terms will necessarily be assigned

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

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

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

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

$$\frac{\Gamma \vdash_{\mathcal{A}} s : \bigvee_{i \in I} b_i \qquad \Gamma \vdash_{\mathcal{A}} t_i : \theta \quad (i \in I)}{\Gamma \vdash_{\mathcal{A}} \mathsf{case}(s, t_1, \ldots, t_n) : \theta}\ \textsc{Case}$$

$$\frac{j \in I}{\Gamma \vdash_{\mathcal{A}} b_j : \bigvee_{i \in I} b_i}\ \textsc{Union}$$

Table 4.1: HORSC/DTT Intersection and Union type assignment system

some union type, as only a union can refine d according to the above definition of well-kinded types. Intuitively, a use of case where the scrutinee diverges will never be reduced, hence our natural usage of $\bot$ for the empty union.

We now present the type system itself. As before a typing for some term t describes the tree generated by t so that a term typed with an intersection of automaton states generates a tree accepted from *all* the states, while a term typed with a union of data types generates *some* of the corresponding data items. For example, the typing $\lambda x.s : (q_0 \wedge q_1) \to (b_0 \vee b_1)$ says that we have a function that takes a tree t 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.

The typing derivations are constructed using the rules in Table 4.1. Note in particular the final two (new) rules, which cover the addition of case to the term language. In Case each possible typing for s 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. Each $b \in \mathsf{B}$ can be typed using the Union rule by any disjunction containing a type of the same name.

Example 4.5. In order to see the new typing rules in action, we give a typing

derivation for a reduct of S that includes case. According to $\mathcal{R}$:

$$S \to \mathsf{case}(H, G\ \mathtt{succ}, G\ \mathtt{pred}) \to \mathsf{case}(b_1, G\ \mathtt{succ}, G\ \mathtt{pred})$$

We give one of the environments $\{G : \top \to q_0\}$ which can be used to assign q_0 to $\mathsf{case}(b_1, G\ \mathtt{succ}, G\ \mathtt{pred})$ in $\vdash_{\mathcal{A}_4}$. In this derivation we can see how the assumption that the scrutinee has type $\bigvee\{b_1\}$ requires us to check that the first choice term, $G\ \mathtt{succ}$, has type q_0.

$$\textsc{Case}\ \dfrac{\textsc{Union}\ \dfrac{}{G : \top \to q_0 \vdash_{\mathcal{A}_4} b_1 : \bigvee\{b_1\}} \quad \dfrac{\dfrac{}{G : \top \to q_0 \vdash_{\mathcal{A}_4} G : \top \to q_0}\textsc{Var}}{G : \top \to q_0 \vdash_{\mathcal{A}_4} G\ \mathtt{succ} : q_0}\textsc{App}}{G : \top \to q_0 \vdash_{\mathcal{A}_4} \mathsf{case}(b_1, G\ \mathtt{succ}, G\ \mathtt{pred}) : q_0}$$

Characterisation of the HORSC model checking problem

Following Kobayashi's technique, we characterise the HORSC model checking problem using a definition of consistent type environments.

Definition 4.6. Fix a HORSC $\mathcal{G} = \langle \Sigma, \mathcal{N}, q, S\rangle$ and a DTT $\mathcal{A}$. We say that a type environment Γ is $(\mathcal{G}, \mathcal{A})$-*consistent* just if

(i) $\Gamma :: \mathcal{N}$

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

We write $\Gamma \vdash_{\mathcal{A}} (\mathcal{G}, t) : \theta$ just if Γ is $(\mathcal{G}, \mathcal{A})$-consistent and $\Gamma \vdash_{\mathcal{A}} t : \theta$. We write $\vdash_{\mathcal{A}} (\mathcal{G}, t) : \theta$ if $\Gamma \vdash_{\mathcal{A}} (\mathcal{G}, t) : \theta$ for some Γ.

Theorem 4.1 (Characterisation)**.** *Given a HORSC $\mathcal{G}$ and a DTT $\mathcal{A}$, the tree language $[\![\mathcal{G}]\!]$ is included in the tree language of $\mathcal{A}^{\perp}$ if, and only if, there exists a $(\mathcal{G}, \mathcal{A})$-consistent type environment Γ such that $S : q_0 \in \Gamma$. Equivalently, $\vdash_{\mathcal{A}} (\mathcal{G}, S) : q_0$.*

Our proof proceeds in two parts, following that of Theorem 2.2. We deal with soundness first, and then completeness.

Soundness

Lemma 4.2 (Substitution)**.** *If $\Gamma, x : \theta_1, \cdots, x : \theta_n \vdash_{\mathcal{A}} t : \theta$ (where $x \notin \mathsf{dom}(\Gamma)$) and for all $i \in [1..n]$, $\Gamma \vdash_{\mathcal{A}} u : \theta_i$ then $\Gamma \vdash_{\mathcal{A}} t[u/x] : \theta$.*

Proof. Take Δ_i to be the derivation witnessing $\Gamma \vdash_{\mathcal{A}} u : \theta_i$, for each $i \in [1..n]$. Then the result follows by induction over the derivation of $\Gamma, x : \theta_1, \cdots, x : \theta_n \vdash_{\mathcal{A}} t : \theta$, substituting any leaf of the form $\Gamma, x : \theta_1, \cdots, x : \theta_n \vdash_{\mathcal{A}} x : \theta_i$ with Δ_i. □

Lemma 4.3 (Subject Reduction). *If* $\vdash_{\mathcal{A}} (\mathcal{G}, t) : q$ *and* $t \to_{\mathcal{G}} t'$ *then* $\vdash_{\mathcal{A}} (\mathcal{G}, t') : q$.

Proof. As $\vdash_{\mathcal{A}} (\mathcal{G}, t) : q$, there exists a specific Γ such that $\Gamma \vdash_{\mathcal{A}} (\mathcal{G}, t) : q$. We will show that also $\Gamma \vdash_{\mathcal{A}} (\mathcal{G}, t') : q$ by induction on the derivation for $t \to_{\mathcal{G}} t'$. Typically this derivation will involve a number of uses of the context rule followed by use of either the $\mathcal{R}$ axiom or the case axiom. The inductive step (inside a context) is uninteresting, so we consider here the two base cases:

(i) $t = F\, t_1 \cdots t_n$. Then $t' = s[\tilde{t}/\tilde{x}]$ where $\lambda\tilde{x}.s \in \mathcal{R}(F)$. The judgement $\Gamma \vdash_{\mathcal{A}} F\, t_1 \cdots t_n : q$ must use the App rule with antecedents:

$$\Gamma \vdash_{\mathcal{A}} F : \bigwedge_{j \in [1..m_1]} \theta_{1,j} \to \cdots \to \bigwedge_{j \in [1..m_n]} \theta_{n,j} \to q \quad (= \theta_F)$$

$$\Gamma \vdash_{\mathcal{A}} t_i : \theta_{i,j} (\text{for } i \in [1..n], j \in [1..m_i])$$

Hence $F : \theta_F \in \Gamma$ and as Γ is $(\mathcal{G}, \mathcal{A})$-consistent, we know that $\Gamma \vdash_{\mathcal{A}} \lambda\tilde{x}.s : \theta_F$ and via Abs that $\Gamma \cup \{x_i : \theta_{i,j} \mid i \in [1..n], j \in [1..m_i]\} \vdash_{\mathcal{A}} s : q$. We apply Lemma 4.2 n times to obtain $\Gamma \vdash_{\mathcal{A}} s[\tilde{t}/\tilde{x}] : q$ as required.

(ii) $t = \mathsf{case}(s, t_1, \cdots, t_n)$. From the derivation for $t \to_{\mathcal{G}} t'$, $s = b_i$ and $t = t_i$ for some $i \in [1..n]$. Necessarily the derivation of $\Gamma \vdash_{\mathcal{A}} t : q$ is:

$$\frac{\overline{\Gamma \vdash_{\mathcal{A}} b_i : \bigvee B} \qquad \Delta_i \qquad \cdots}{\Gamma \vdash_{\mathcal{A}} \mathsf{case}(b_i, t_1, \cdots, t_n) : q}$$

where Δ_i witnesses $\Gamma \vdash_{\mathcal{A}} t_i : q$. Then $\Gamma \vdash_{\mathcal{A}} (\mathcal{G}, t') : q$ as required.

□

Lemma 4.4 (Acceptance of typed concrete prefixes). *If* $\vdash_{\mathcal{A}} (\mathcal{G}, t) : q$, *then* $t^\perp$ *is accepted by* $\mathcal{A}^\perp$ *from state* q.

Proof. The extension of $(\cdot)^\perp$ to case terms is all that is required to lift this lemma from the HORS setting. We present Kobayashi's reasoning for completeness.

We will proceed by induction on the structure of $t^\perp$. If t is not headed by a terminal symbol, then $t = \perp$ and the result is immediate. Otherwise $t^\perp = a\, t_1^\perp \cdots t_n^\perp$. By assumption, $\Gamma \vdash_{\mathcal{A}} a\, t_1 \cdots t_n : q$, the proof of which must use the Term rule with $\delta(q, a) = q_1 \cdots q_n$ and contain $\Gamma \vdash_{\mathcal{A}} t_i : q_i$ for each $i \in [1..n]$. By the induction hypothesis, each $t_i^\perp$ is accepted by $\mathcal{A}^\perp$ from state q_i and so $t^\perp$ is accepted from q. □

Theorem 4.5 (Soundness). *If* $\vdash_{\mathcal{A}} (\mathcal{G}, S) : q_0$ *then* $[\![\mathcal{G}]\!] \subseteq \mathcal{L}(\mathcal{A}^\perp)$.

Proof. Assume that $\vdash_{\mathcal{A}} (\mathcal{G}, S) : q_0$ is witnessed by Γ and take an arbitrary element T of $[\![\mathcal{G}]\!]$. T is defined by a fair, outermost, maximal reduction sequence $\langle t_i \rangle_{i \in I}$ where I is a prefix of W such that $\bigsqcup_{i \in I} t_i^{\perp} = T$. We must show that for every $i \in I$ that $t_i^{\perp}$ is accepted by $\mathcal{A}^{\perp}$, and from Lemma 4.4 it suffices to show that $\Gamma \vdash_{\mathcal{A}} t_i : q_0$. By assumption $\Gamma \vdash_{\mathcal{A}} S : q_0$ and by subject reduction (Lemma 4.3) for any t such that $S \to_{\mathcal{G}}^{*} t$ necessarily $\Gamma \vdash_{\mathcal{A}} t : q_0$. □

Completeness

Lemma 4.6 (Inverse substitution)**.** *If* $\Gamma \vdash_{\mathcal{A}} t[u/x] : \theta$ *then there exists* $\Gamma' = \Gamma, x : \theta_1, \cdots, x : \theta_n$ *such that* $\Gamma' \vdash_{\mathcal{A}} t : \theta$ *and for each* $i \in [1..n]$, $\Gamma \vdash_{\mathcal{A}} u : \theta_i$.

Proof. By induction on the structure of t.

(i) $t = a \ (\in \Sigma)$. Then $n = 0$ and $\Gamma = \Gamma'$.

(ii) $t = y \ (\in \mathcal{V})$ and $y \neq x$. Then $n = 0$ and $\Gamma = \Gamma'$.

(iii) $t = x$. Then $\Gamma' = \Gamma, x : \theta$, $n = 1$, $\theta = \theta_1$ and $\Gamma \vdash_{\mathcal{A}} u : \theta$ by assumption.

(iv) $t = t_1\ t_2$. Then from the induction hypothesis we have:

$$\Gamma_1' \vdash_{\mathcal{A}} t_1 : \bigwedge_{i \in [1..n]} \theta_i \to \theta \text{ and } \forall i \in [1..n] \cdot \Gamma_{2,i}' \vdash_{\mathcal{A}} t_2 : \theta_i$$

where each environment is an extension of Γ with typings for x. By application of weakening, $\Gamma' = \Gamma_1' \cup \bigcup_{i \in [1..n]} \Gamma_{2,i}'$ can be substituted in the judgements above, and so $\Gamma' \vdash_{\mathcal{A}} t_1\ t_2 : \theta$. Every $x : \theta' \in \Gamma'$ comes from Γ_1' or some $\Gamma_{2,i}'$, and so from the induction hypothesis $\Gamma \vdash_{\mathcal{A}} u : \theta'$.

(v) $t = \mathsf{case}(s, t_1, \cdots, t_n)$. Necessarily $\theta = \beta$ for some ground kind β. From the induction hypothesis, we have $\Gamma_s' \vdash_{\mathcal{A}} s : \bigvee B$ and for each $b_i \in B$, $\Gamma_i' \vdash_{\mathcal{A}} t_i : \beta$. As for the previous case, the union environment $\Gamma' = \Gamma_s \cup \bigcup_{i \in [1..n]} \Gamma_i'$ can be used with weakening in these derivations and so used in $\Gamma' \vdash_{\mathcal{A}} \mathsf{case}(s, t_1, \cdots, t_n) : \beta$. Again every new type binding $x : \theta' \in \Gamma'$ can be proven for u via the induction hypothesis.

□

Lemma 4.7 (Subject expansion)**.** *Fix a term* t. *If for all* t' *such that* $t \to_{\mathcal{G}} t'$ *via outermost reduction,* $\Gamma \vdash_{\mathcal{A}} t' : \beta$ *for ground kind* β, *then* $\vdash_{\mathcal{A}} (\mathcal{G}, t) : \beta$.

Proof. We proceed by induction over the structure of t.

(i) $t = \mathsf{case}(s, t_1, \cdots, t_n)$. Then either the whole term is a redex and $s = b_i$, or it is not and s reduces.

a) $s = b_i$. Then the reduction is deterministic: $t' = t_i$. By assumption $\Gamma \vdash_{\mathcal{A}} t_i : \beta$ is witnessed by derivation Δ_i. Then

$$\frac{\overline{\Gamma \vdash_{\mathcal{A}} b_i : b_i} \qquad \Delta_i}{\Gamma \vdash_{\mathcal{A}} \mathsf{case}(s, t_1, \cdots, t_n) : \beta}$$

b) $s \neq b_i$. Then for every outermost reduction $s \to_{\mathcal{G}} s_j$ where $j \in [1..n]$ we have:

$$\frac{\dfrac{\Delta_j}{\Gamma \vdash_{\mathcal{A}} s_j : \bigvee B_j} \qquad \dfrac{\Delta_{j,i}}{\Gamma \vdash_{\mathcal{A}} t_i : \beta} \quad (\text{for each } b_i \in B_j)}{\Gamma \vdash_{\mathcal{A}} \mathsf{case}(s_j, t_1, \cdots, t_n) : \beta}$$

Let $B = \bigcup_{j \in J} B_j$, and by the induction hypothesis and union weakening, $\Gamma' \vdash_{\mathcal{A}} s : \bigvee B$. By assumption we have a proof of $\Gamma \vdash_{\mathcal{A}} t_i : \beta$ for every $b_i \in B$ and therefore $\Gamma \cup \Gamma' \vdash_{\mathcal{A}} \mathsf{case}(s, t_1, \cdots, t_n) : \beta$ as required.

(ii) $t = F\ t_1 \cdots t_n$ ($F \in \mathcal{N}$). Then let $\mathcal{R}(F) = \{\lambda\tilde{x}.s_1, \cdots, \lambda\tilde{x}.s_m\}$ so that for each $j \in [1..m]$, $t \to_{\mathcal{G}} s_j[\tilde{t}/\tilde{x}]$ and $\Gamma \vdash_{\mathcal{A}} s_j[\tilde{t}/\tilde{x}] : \beta$. From Lemma 4.6, for each $j \in [1..m]$, also $\Gamma'_j \vdash_{\mathcal{A}} s_j : \beta$, where each Γ'_j extends Γ with typings for $\tilde{x} = \{x_1, \cdots, x_n\}$ and for each $i \in [1..n]$ and each $x : \theta \in \Gamma_j$, $\Gamma \vdash_{\mathcal{A}} t_i : \theta$.

Let $\theta' = \bigwedge T_1 \to \cdots \to \bigwedge T_n \to \beta$ where $T_i = \{\theta \mid x_i : \theta \in \Gamma'_j, 1 \leqslant j \leqslant m\}$ and $\Gamma' = \Gamma \cup \{F : \theta'\}$. Then as noted above we can type each argument of F with every member of the corresponding intersection in θ' and so construct a derivation for $\Gamma' \vdash_{\mathcal{A}} F\ t_1 \cdots t_n : \beta$. Having introduced a new type into our environment ($F : \theta'$), we must reestablish $(\mathcal{G}, \mathcal{A})$-consistency of Γ'. The property follows by weakening the environment in $\Gamma'_j \vdash_{\mathcal{A}} s_j : \beta$ to $\bigcup_{j \in [1..m]} \Gamma'_j$ and use of the Abs rule at the bottom of each derivation to yield $\Gamma \vdash_{\mathcal{A}} s_j : \theta'$.

(iii) $t = a\ t_1 \cdots t_n$ ($a \in \Sigma$). Follows immediately from the induction hypothesis.

□

Lemma 4.8 (Typing of accepted concrete prefixes)**.** *If $t^\perp$ is accepted by $\mathcal{A}^\perp$ from state q, then $\emptyset \vdash_{\mathcal{A}^\perp} t^\perp : q$.*

Proof. The inverse of Lemma 4.4, and it follows by similar reasoning. We induct on the structure of $t^\perp$. If $t^\perp = \perp$, the result follows immediately as $\emptyset \vdash_{\mathcal{A}^\perp} \perp : q$ for all q. Otherwise $t^\perp$ has some concrete prefix i.e. $t^\perp = a\ t_1 \cdots t_n$. By assumption that $t^\perp$ is accepted from q, $\delta(q, a) = q_1 \cdots q_n$ and further $t_i^\perp$ is accepted from q_i. Then from the induction hypothesis, $\emptyset \vdash_{\mathcal{A}^\perp} t_i^\perp : q_i$ for every $i \in [1..n]$. Using App and Term, $\emptyset \vdash_{\mathcal{A}^\perp} t^\perp : q$ as required. □

Theorem 4.9 (Completeness). *If* $[\![\mathcal{G}]\!] \subseteq \mathcal{L}(\mathcal{A}^\perp)$ *then* $\vdash_{\mathcal{A}} (\mathcal{G}, S) : q_0$.

Proof. With the intermediate lemmas in place, accounting for non-determinism is sufficient to lift Kobayashi's proof technique and exhibit a witnessing type environment. We first recapitulate the definition of Shrink, adjusted for our non-deterministic setting. Remember that Shrink removes any type bindings from the environment for which the right-hand side cannot be proven to have the same type. In other words, it removes counter-examples to $(\mathcal{G}, \mathcal{A})$-consistency.

$$\mathsf{Shrink}_{(\mathcal{G},\mathcal{A})}(\Gamma) = \{F : \theta \mid F : \theta \in \Gamma, \forall \lambda\tilde{x}.s \in \mathcal{R}(F) \cdot \Gamma \vdash_{\mathcal{A}} \lambda\tilde{x}.s : \theta\}$$

In the lattice of type environments, Shrink is a monotone function, and therefore guaranteed to have fixed points, which coincide with $(\mathcal{G}, \mathcal{A})$-consistency. As such the greatest fixed point, being the largest, is our best chance at a witness of $\vdash_{\mathcal{A}} (\mathcal{G}, S) : q_0$. The greatest fixed point can be calculated by iteration from Γ_{max}, the top element of the lattice. Let $\nu\mathsf{Shrink}_{(\mathcal{G},\mathcal{A})} = \mathsf{Shrink}^m_{(\mathcal{G},\mathcal{A})}(\Gamma_{\mathsf{max}})$, for some finite m. Thus it suffices to show that $[\![\mathcal{G}]\!] \subseteq \mathcal{L}(\mathcal{A}^\perp)$ implies $S : q_0 \in \mathsf{Shrink}^m_{(\mathcal{G},\mathcal{A})}(\Gamma_{\mathsf{max}})$.

We next construct a new scheme $\mathcal{G}^m = \langle \Sigma^\perp, \mathcal{N}^m, \mathcal{R}^m, S^m \rangle$ with a finite unfolding of the rules of $\mathcal{G}$, where $\mathcal{N} = \{F^k : \kappa \mid F : \kappa \in \mathcal{N}, 0 \leqslant k \leqslant m-1\}$ and $\mathcal{R}^m$ is the smallest set that satisfies the following rules:

$$\frac{\lambda x.s \in \mathcal{R}(F) \qquad 0 \leqslant k \leqslant m-1}{\lambda x.s[G^k/G \mid G \in \mathcal{N}] \in \mathcal{R}^m(F^{k+1})} \qquad \frac{\lambda x.s \in \mathcal{R}(F)}{\lambda x.\perp \in \mathcal{R}(F^0)}$$

$\mathcal{G}^m$ is defined without recursion, and therefore its tree language is one of finite trees over $\Sigma^\perp$. $[\![\mathcal{G}^m]\!]$ approximates $[\![\mathcal{G}]\!]$ in the sense that every element is related to an element of $[\![\mathcal{G}]\!]$ by $\sqsubseteq$ (Definition 2.3) and so $[\![\mathcal{G}]\!] \subseteq \mathcal{L}(\mathcal{A}^\perp)$ definitely gives us $[\![\mathcal{G}^m]\!] \subseteq \mathcal{L}(\mathcal{A}^\perp)$.

It follows from Lemma 4.8 that $\emptyset \vdash_{\mathcal{A}^\perp} t : q_0$, for every $t \in [\![\mathcal{G}^m]\!]$. From Lemma 4.7, as $S \to^*_{\mathcal{G}^m} t$ for every $t \in [\![\mathcal{G}^m]\!]$, necessarily $\vdash_{\mathcal{A}^\perp} (\mathcal{G}^m, S^m) : q_0$, witnessed by Γ. Take the family of k-*flattenings* of Γ to be:

$$\Gamma_k = \{F : \theta \mid F^j : \theta \in \Gamma, j \geqslant k\}$$

where $\Gamma_0 = \Gamma$.

Next we show that each application of $\mathsf{Shrink}_{(\mathcal{G},\mathcal{A}^\perp)}$ only removes one "layer" of type bindings, meaning that the greatest fixed point reached after m iterations must contain $S : q_0$ as well as being $(\mathcal{G}, \mathcal{A}^\perp)$-consistent. This follows from $\Gamma_{k+1} \subseteq \mathsf{Shrink}_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_k)$ for every $k \in [0..m-1]$. Take any $F : \theta \in \Gamma_{k+1}$. Then for some $j \geqslant k$, $F^j : \theta \in \Gamma$ and for each $\lambda\tilde{x}.s \in \mathcal{R}^m(F^j)$, $\Gamma \vdash_{\mathcal{A}^\perp} \lambda\tilde{x}.s : \theta$. As s only contains non-terminals of the form G^{j-1}, Γ_k contains all the necessary bindings to prove for

each $\lambda\widetilde{x}.s \in \mathcal{R}(F)$, $\Gamma_k \vdash_{\mathcal{A}^\perp} \lambda\widetilde{x}.s : \theta$ by renaming each G^{j-1} to G. Therefore $F : \theta$ is preserved by $\mathsf{Shrink}_{(\mathcal{G},\mathcal{A}^\perp)}$.

From monotonicity of Shrink, $\Gamma_{k+1} \subseteq \mathsf{Shrink}_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_k)$ implies that

$$\mathsf{Shrink}_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_{k+1}) \subseteq \mathsf{Shrink}^2_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_k)$$

which allows us to construct a sequence of inclusions:

$$\Gamma_m \subseteq \mathsf{Shrink}_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_{m-1}) \subseteq \cdots \subseteq \mathsf{Shrink}^m_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_0) \subseteq \mathsf{Shrink}^m_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_{\mathsf{max}})$$

Note that by definition Γ_m contains $S : q_0$, and this is preserved to the right. As $\perp$ is not present in the rules of $\mathcal{G}$, we can drop back to the standard type system $\vdash_{\mathcal{A}}$:

$$\mathsf{Shrink}^m_{(\mathcal{G},\mathcal{A}^\perp)}(\Gamma_{\mathsf{max}}) = \mathsf{Shrink}^m_{(\mathcal{G},\mathcal{A})}(\Gamma_{\mathsf{max}})$$

Thus our proposed witness contains $S : q_0$ as required and we are done. □

4.3 Model checking HORSC

Now that we have our characterisation in place, we will define a model checking algorithm that extends Algorithm 1. The main difficulty in dealing with case is that it behaves in a call-by-value style, requiring that the scrutinee term be reduced to a value before the reduction can continue any further. In contrast to the standard call-by-name behaviour, we cannot continue exploring with knowledge of what return type we require the term to have. Instead we need to determine which of the elements of the finite domain the scrutinee can reduce to in the worst case, so that we can limit the choice terms that we analyse to a minimum.

In keeping with the style of the original algorithm, we will ensure that the open derivations represent valid concrete derivations at all times. In Section 3.2 we gave an intuitive explanation of how the algorithm should behave when a terminal, non-terminal or variable is encountered and this remains unchanged. For the rest of this section we fix a HORSC $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ with $B \subseteq \Sigma$, and a DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$. Imagine that in the course of constructing the open derivations we encounter a subgoal of the form $\vdash_{\mathcal{A}} \mathsf{case}(s, t_1, \cdots, t_n) : q$. Syntactically we are limited to using the Case rule at this point, and if we assume initially that s has type $\perp = \bigvee \emptyset$ (i.e. that it diverges) then we are left with no other subgoals.

Further exploration of $\vdash_{\mathcal{A}} s : \bigvee \emptyset$ is very likely to show that assuming that s always diverges is incorrect. If we encounter a subgoal with term b_i, then we must ensure, similarly to encountering a variable, that in the original case judgement the scrutinee type is updated to include b_i, and the corresponding subgoal $\vdash_{\mathcal{A}} t_i : q$ is added. We will write type variables representing unions (of kind d) β (rather than α) and now illustrate how they keep open derivations consistent.

Example 4.7 (Building a derivation for "$[\![\mathcal{H}_2]\!] \subseteq \mathcal{L}(\mathcal{A}_4^{\perp})$")**.** We will now illustrate how to start the search for a type environment witnessing that our example HORSC generates trees accepted by $\mathcal{A}_4$. As before, the search is driven by the characterisation (Theorem 4.1), which requires that a witnessing environment be both consistent and contain $S : q_0$. If we start with $\{S : q_0\}$, then to ensure consistency we must immediately check $\vdash \mathcal{R}(S) : q_0$. Since the right-hand side of S is a case construct:

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

the simplest way to proceed is to assume that $H : \beta$ where β is a fresh type variable initialised to $\bigvee \emptyset = \perp$. As usual, $\perp$ 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 some b_i then β will be updated accordingly. However, we must check this by exploring $\mathcal{R}(H)$ to find if it can reduce to any members of the finite domain B. This leaves us with the derivation

$$\dfrac{\dfrac{}{\{H : \beta\} \vdash H : \beta}\ \textsc{Var}}{\{H : \beta\} \vdash \mathsf{case}(H, G\ \mathtt{succ}, G\ \mathtt{pred}) : q_0}\ \textsc{Case}$$

and as the non-terminal H has two possible rewrite rules, we must build two further derivations, which are rooted at the following judgements and correspond to the respective right-hand sides of H:

$$\emptyset \vdash b_1 : \beta \qquad\qquad \emptyset \vdash H : \beta$$

To build a derivation rooted at the left-hand judgement we use the Union rule (see the derivation on the left in the top row of Table 4.2). This requires β to contain b_1, causing an additional obligation to type the first choice term (G succ) of the case construct.

We will also continue the derivation from the new judgement for G succ in:

$$\dfrac{\textsc{Var}\ \dfrac{}{\{H : \beta\} \vdash H : \beta} \qquad \{H : \beta\} \vdash G\ \mathtt{succ} : q_0}{\{H : \beta\} \vdash \mathsf{case}(H, G\ \mathtt{succ}, G\ \mathtt{pred}) : q_0}\ \textsc{Case}$$

In order to apply the App rule (in a bottom-up fashion), we introduce another type variable, α_1. Dually to the use of Case, and as for the algorithm in Chapter 3, α_1 is initialised 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 4.2), 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 α_1 to include $\alpha_2 \to q_0$, and just as before, after adding the new type to α_1, the use of the App rule to justify the

$$\frac{b_1 \in \beta}{\emptyset \vdash b_1 : \beta}\,\textsc{Union} \qquad \frac{\dfrac{\dfrac{}{\{g : \alpha_1\} \vdash g : \alpha_2 \to q_0}\,\textsc{Var}}{\{g : \alpha_1\} \vdash g\ zero : q_0}\,\textsc{App}}{\emptyset \vdash \lambda g.g\ zero : \alpha_1 \to q_0}\,\textsc{Abs}$$

$$\frac{\dfrac{}{\Gamma^o \vdash H : \beta}\,\textsc{Var} \quad \dfrac{\dfrac{}{\Gamma^o \vdash G : \alpha_1 \to q_0}\,\textsc{Var} \quad \Gamma^o \vdash succ : \alpha_2 \to q_0}{\Gamma^o \vdash G\ succ : q_0}\,\textsc{App}}{\Gamma^o \vdash case(H, G\ succ, G\ pred) : q_0}\,\textsc{Case}$$

Table 4.2: Examples of open derivations ($\Gamma^o = \{H : \beta\}$)

judgement $\emptyset \vdash G\ succ : q_0$ is no longer valid. We must add an extra judgement for the operand (see the lower derivation in Table 4.2), which in turn can be justified by the Term rule.

Formal definition of the algorithm

We will now formally extend the machinery of open types and reification to account for union types. First we extend the set of open types to account for the introduction of the new base kind d. Open types may now also take the form $\alpha_1 \to \cdots \alpha_n \to \beta$, where β is a type variable of kind d rather than an automaton state q.

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

Recall that an instantiation map is a function $\Theta : A \to \mathcal{P}(\mathbb{P})$. We extend the definition of the reification map induced by Θ by interpreting type variables of kind d as defining a union, as follows:

$$\begin{aligned} \widehat{\Theta}_o(q) &:= q \\ \widehat{\Theta}_d(B) &:= \bigvee B \\ \widehat{\Theta}_{\kappa_1 \to \kappa_2}(\alpha \to \theta^o) &:= \left(\bigwedge_{\theta_1^o \in \Theta(\alpha)} \widehat{\Theta}_{\kappa_1}(\theta_1^o) \right) \to \widehat{\Theta}_{\kappa_2}(\theta^o) \end{aligned}$$

The reification map $\widehat{\Theta} : \mathbb{P} \to Type$ is then defined to be $\theta^o \mapsto \widehat{\Theta}_\kappa(\theta^o)$ for $\theta^o \in \mathbb{P}_\kappa$.

Example 4.8 (Reification of open types with unions)**.** Say $\kappa = (o \to o) \to (o \to d) \to d$ and an element of $\mathbb{P}_\kappa$ is $\theta^o = \alpha_1 \to \alpha_2 \to \beta_1$. If the instantiation map Θ is

defined by:

$$\begin{aligned}
\alpha_1 &\mapsto \{\alpha_3 \to q_0\} \\
\alpha_2 &\mapsto \{\alpha_4 \to \beta_2\} \\
\alpha_3 &\mapsto \{q_0, q_1\} \\
\alpha_4 &\mapsto \emptyset \\
\beta_1 &\mapsto \{b_1, b_2\} \\
\beta_2 &\mapsto \{b_1\}
\end{aligned}$$

Then the reified form of θ^o is:

$$\widehat{\Theta}(\theta^o) = (q_0 \wedge q_1 \to q_0) \to (\top \to b_1) \to b_1 \vee b_2$$

The definitions of open judgements, type environments and derivations are lifted directly to use this new notion of open types. States of the algorithm will be labelled additionally with the right-hand side of the rewrite rule that was used to create them, as a non-terminal is no longer sufficient to distinguish different reductions. $\mathcal{D}$ is therefore a $((\mathcal{R} \times \mathbb{P}) \times D)$-labelled tree, where each node n is labelled by a quadruple $(F, s : \theta^o, \Delta^o)$. The open derivation Δ^o will be rooted at an open judgement $\Gamma^o \vdash_{\mathcal{A}} \lambda\tilde{x}.s : \theta^o$, where $\lambda\tilde{x}.s \in \mathcal{R}(F)$. Distinguishing different non-deterministic choices in this way allows us to use the same definition termination in terms of finding a complete cut.

The top loop (Algorithm 2) is slightly adjusted; the main changes can be found in the rules of the algorithm, which can be seen in Table 4.3. The new rules A-Cas and A-Uni correspond to the typing rules Case and Union respectively as illustrated in Example 4.7, while the third addition A-β, keeps the open derivations consistent when updating union type variables, just as A-α does for intersection type variables.

Example 4.9 (Completion of analysis of "$[\![\mathcal{H}_2]\!] \subseteq \mathcal{L}(\mathcal{A}_4^{\perp})$?")**.** We are now in a position to finish the run of Algorithm 2 started in Example 4.7. Δ_1^o, Δ_2^o and Δ_3^o are the open derivations explored in the previous example, with Δ_2^o, Δ_3^o and Δ_4^o being required to prove Δ_1^o (see Table 4.4). Previously we did not explore Δ_4^o, but here we can see that doing so creates two new derivations with the same structure, as might be expected. An interesting feature of this example is the jumping between derivations caused by the higher-order computation. Observe in Δ_2^o how closing the top left judgement adds $\alpha_2 \to q_0$ to $\Theta(\alpha_1)$, then A-α adds a judgement for succ in Δ_1^o, and finally using A-Cst to close this judgement modifies α_2 and we return to Δ_1^o.

In searching for a complete cut we observe that Δ_5^o and Δ_6^o trivially have the same reified-type binding as Δ_3^o and Δ_4^o ($H, H : b_1$ and $H, b_1 : b_1$). The only unjustified derivation is Δ_6^o and thus the nodes labelled by Δ_2^o, Δ_3^o and Δ_4^o form a

Algorithm 2: Model Checking HORSC

input : HORSC $\mathcal{H} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$
output: Whether $[\![\mathcal{H}]\!] \subseteq \mathcal{L}(\mathcal{A}^{\perp})$
$\mathcal{D} :=$ singleton tree with label $(S : q_0, \emptyset \vdash \mathcal{R}(S) : q_0)$
$\Theta := \{\alpha \mapsto \emptyset \mid \alpha \in A\}$
while $\mathcal{D}^{j}$ does *not* have a complete cut **do**
  $W :=$ unjustified$(\mathcal{D})$
  while $W \neq \emptyset$ **do**
    $\Gamma^{o} \vdash s : \theta^{o} :=$ any element of W
    if state$(\theta^{o}) = q \wedge s \in \Sigma \wedge (q, s) \notin \delta$ **then**
      return No
    else
      Apply matching rule A-Ξ possibly followed by A-α, A-β
    end
    $W := W \setminus \{\Gamma^{o} \vdash s : \theta^{o}\}$
    if $s \notin \mathcal{N}$ **then** $W := W \cup \{$all new unjustified judgements$\}$
  end
end
return YES

complete cut. As a result the reification of the bindings $\{S : q_0, G : \alpha_1 \to q_0, H : \beta\}$ will be a $(\mathcal{H}_2, \mathcal{A}_4)$-consistent type environment:

$$\widehat{\Theta}(\{S : q_0, G : \alpha_1 \to q_0, H : \beta\}) = \{S : q_0, G : (q_1 \to q_0) \to q_0, H : b_1\}$$

Correctness

We now prove the correctness of the extended algorithm. Unsurprisingly the proof follows the same structure as in Section 3.2, and for inductions over term structure or rules of the algorithm we elide cases proved there if they are not affected. Throughout the rest of this section we fix a HORSC $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ with finite domain $B \subseteq \Sigma$ and DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$. We first prove soundness.

Lemma 4.10 (Consistency of typing derivations)**.** *Let $(\mathcal{D}, \Theta)$ be a state of the algorithm, n be a node of $\mathcal{D}$, and $\mathcal{D}(n) = (F, s : \theta^{o}, \Delta^{o})$ where the judgement at the root of the open derivation Δ^{o} is $\Gamma^{o} \vdash \lambda\tilde{x}.s : \theta^{o}$ where $\lambda\tilde{x}.s \in \mathcal{R}(F)$.*

(i) Every justified judgement of $\widehat{\Theta}(\Delta^{o})$ is an instance of a rule or axiom of $\vdash_{\mathcal{A}}$. Hence, if Δ^{o} is justified then $\widehat{\Theta}(\Delta^{o})$ is a valid type derivation, witnessing $\widehat{\Theta}(\Gamma^{o}) \vdash \mathcal{R}(F) : \widehat{\Theta}(\theta^{o})$.

Rule	If	Then
A-App	(i) $\Gamma^o \vdash t\ u : \theta^o$ (ii) α fresh	(i) $\dfrac{\Gamma^o \vdash t : \alpha \to \theta^o}{\Gamma^o \vdash t\ u : \theta^o}$
A-Fun	(i) $J = \Gamma^o \vdash F : \theta^o \quad (\in \Delta^o)$ (ii) $\mathcal{R}(F) = \{\lambda\tilde{x}.s_1, \cdots, \lambda\tilde{x}.s_m\}$ (iii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$	(i) $\overline{\Gamma^o, F : \theta^o \vdash F : \theta^o}$ (ii) For $\Gamma_1^o \vdash s : \theta_1^o$ in Δ^o, $\Gamma_1^o := \Gamma_1^o \cup \{F : \theta^o\}$ (iii) For each $j \in [1..m]$ add new rightmost child $(F, s_j : \theta^o, \Delta_j^o)$ to Δ^o: $\Delta_j^o = \dfrac{\{x_i : \alpha_i \mid 1 \leqslant i \leqslant n\} \vdash s : q}{\vdots}\ \dfrac{}{\emptyset \vdash \lambda\tilde{x}.s_j : \theta^o}$
A-Var	(i) $\Gamma^o, x : \alpha \vdash x : \theta^o$	(i) $\overline{\Gamma^o, x : \alpha \vdash x : \theta^o}$ (ii) $\Theta(\alpha) := \Theta(\alpha) \cup \{\theta^o\}$
A-Cst	(i) $\Gamma^o \vdash a : \theta^o$ (ii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$ (iii) $\delta(q, a) = q_1 \cdots q_n$	(i) $\overline{\Gamma^o \vdash a : \theta^o}$ (ii) $\forall i \in [1..n] \cdot \Theta(\alpha_i) := \Theta(\alpha_i) \cup \{q_i\}$
A-α	(i) $\dfrac{\overline{\Gamma^o \vdash t : \alpha \to \theta_1^o} \quad \cdots}{J = \Gamma^o \vdash t\ u : \theta_1^o}$ (ii) $\theta^o \in \Theta(\alpha)$ (iii) $\Gamma^o \vdash u : \theta^o$ not a child of J	(i) $\dfrac{\overline{\Gamma^o \vdash t : \alpha \to \theta_1^o} \quad \Gamma^o \vdash u : \theta^o \quad \cdots}{\Gamma^o \vdash t\ u : \theta_1^o}$
A-Cas	(i) $\Gamma^o \vdash \mathsf{case}(s, t_1, \cdots, t_n) : \theta^o$ (ii) β fresh	(i) $\dfrac{\Gamma^o \vdash s : \beta}{\Gamma^o \vdash \mathsf{case}(s, t_1, \cdots, t_n) : \theta^o}$
A-Uni	(i) $\Gamma^o \vdash b_i : \beta$	(i) $\overline{\Gamma^o \vdash b_i : \beta}$ (ii) $\Theta(\beta) := \Theta(\beta) \cup \{b_i\}$
A-β	(i) $\dfrac{\overline{\Gamma^o \vdash t : \beta} \quad \cdots}{J = \Gamma^o \vdash \mathsf{case}(s, t_1, \cdots, t_n) : \theta^o}$ (ii) $b_i \in \Theta(\beta)$ (iii) $\Gamma^o \vdash t_i : \theta^o$ not a child of J	(i) $\dfrac{\overline{\Gamma^o \vdash t : \beta} \quad \Gamma^o \vdash t_i : \theta^o \quad \cdots}{\Gamma^o \vdash \mathsf{case}(s, t_1, \cdots, t_n) : \theta^o}$

Table 4.3: Rules of HORSC model checking algorithm

$$\mathcal{D} = (S, \mathsf{case}(H, G\ \mathrm{succ}, G\ \mathrm{pred}) : q_0, \Delta_1^o) \rightarrow (G, \lambda g.g\ \mathrm{zero} : \alpha_1 \to q_0, \Delta_2^o)$$
$$\rightarrow (H, b_1 : \beta, \Delta_3^o)$$
$$\rightarrow (H, H : \beta, \Delta_4^o) \swarrow (H, b_1 : \beta, \Delta_5^o)$$
$$\swarrow (H, H : \beta, \Delta_6^o)$$

$$\Delta_1^o = \dfrac{\dfrac{}{\Gamma^o \vdash H : \beta}\ \textsc{Var} \quad \dfrac{\dfrac{}{\Gamma^o \vdash G : \alpha_1 \to q_0}\ \textsc{Var} \quad \dfrac{}{\Gamma^o \vdash \mathrm{succ} : \alpha_2 \to q_0}\ \textsc{Term}}{\Gamma^o \vdash G\ \mathrm{succ} : q_0}\ \textsc{App}}{\Gamma^o \vdash \mathsf{case}(H, G\ \mathrm{succ}, G\ \mathrm{pred}) : q_0}\ \textsc{Case}$$

$$\Delta_2^o = \dfrac{\dfrac{\dfrac{}{\{g : \alpha_1\} \vdash g : \alpha_2 \to q_0}\ \textsc{Var} \quad \dfrac{}{\{g : \alpha_1\} \vdash \mathrm{zero} : q_1}\ \textsc{Term}}{\{g : \alpha_1\} \vdash g\ \mathrm{zero} : q_0}\ \textsc{App}}{\vdash \lambda g.g\ \mathrm{zero} : \alpha_1 \to q_0}\ \textsc{Abs}$$

$$\Delta_3^o = \dfrac{}{\emptyset \vdash b_1 : \beta}\ \textsc{Union} \qquad \Delta_5^o = \dfrac{}{\emptyset \vdash b_1 : \beta}\ \textsc{Union}$$

$$\Delta_4^o = \dfrac{}{\{H : \beta\} \vdash H : \beta}\ \textsc{Var} \qquad \Delta_6^o = \emptyset \vdash H : \beta$$

$$\Theta = \{\alpha_1 \mapsto \{\alpha_2 \to q_0\}, \alpha_2 \mapsto \{q_1\}, \beta \mapsto \{b_1\}\}$$
$$\Gamma^o = \{G : \alpha_1 \to q_0, H : \beta\}$$

Table 4.4: Final state of the algorithm with input $\mathcal{H}_2$ and $\mathcal{A}_4$ (Example 4.4).

(ii) The set of successor nodes of n *is the smallest that for each* $F_1 : \theta_1^o \in \Gamma^o$ *and for each* $\lambda\tilde{x}.s_1 \in \mathcal{R}(F_1)$ *contains a node labelled* $(F_i, s : \theta_1^o, \Delta_1^o)$

Proof. As defined in Algorithm 2, a step of the algorithm comprises an application of one of the A-Ξ rules ($\Xi \in \{\textsc{App}, \textsc{Cst}, \textsc{Var}, \textsc{Fun}, \textsc{Cas}, \textsc{Uni}\}$), followed by A-$\alpha$ or A-β. The base case is vacuously true while the App, Cst and Var are unchanged. Let Δ^o be an open derivation. We consider the remaining cases for an open judgement $J = \Gamma^o \vdash u : \theta^o$ in Δ^o in turn.

- A-Fun; $u = F$. After augmenting the typing environment of every judgement in Δ^o by the binding $F : \theta^o$, the original judgement J is closed by the Var rule. A new node is created for each member $\lambda\tilde{x}.s$ of $\mathcal{R}(F)$ that is labelled by $(F, s : \theta^o, \Delta_1^o)$ where Δ_1^o is an open derivation which is obtained by n applications of the Abs rule to $\emptyset \vdash \lambda\tilde{x}.s : \theta^o$, as required.

- A-Cas; $u = \mathsf{case}(s, t_1, \cdots, t_n)$. The introduction of the fresh type variable β where $\Theta(\beta) = \emptyset$ allows us to apply the Case rule with an empty disjunction for s and therefore without needing to prove anything about the t_i terms.

- A-Uni; $u = b_i$. Adding b_i to $\Theta(\beta)$ immediately allows us to apply Union, justifying this judgement. If b_i was not already a member of $\Theta(\beta)$ this triggers a use of the A-β rule so to add a subgoal for the corresponding choice term t_i at the original case judgement.

□

Lemma 4.11 (Complete cuts yield witnesses)**.** *Let $(\mathcal{D}, \Theta)$ be a state of the algorithm and there exists a complete cut C of $\mathcal{D}^j$. Define Γ to be the set:*

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

Then Γ is $(\mathcal{G}, \mathcal{A})$-consistent.

Proof. The reasoning from Lemma 3.3 holds here, except that in line with Theorem 4.1 we need to show that *every* member of $\mathcal{R}(F)$ can be typed according to each binding in Γ. As the relationship between open type environments of nodes and children is lifted in this way in Lemma 4.10, the result is immediate. □

We now reestablish the relationship between the computation of the algorithm, reduction of the HORSC and prefixes of run-trees over members of the tree language of the HORSC.

Lemma 4.12 (Reduction)**.** *In state $(\mathcal{D}, \Theta)$, for every judgement $J' \in \mathcal{D}$ with the most recent member of the sequence $\mathsf{history}(J')$ not having a type variable rightmost being $J = \Gamma^o \vdash t : \theta^o$ with $\mathsf{state}(\theta^o) = q$:*

(i) $\mathsf{history}(J)$ *restricted to only those judgements $\Gamma^o \vdash a : \theta^o$ where $a \in \Sigma$ is a sequence $J_0 \cdots J_n$ $(\forall i \in [0..n] \cdot J_i = \Gamma_i^o \vdash a_i : \theta_i^o)$ such that there exist words ζ, γ with $|\zeta| = |\gamma| = n$. For each $i \in [0..n]$ if we let ζ_i (resp. γ_i) be the prefix of ζ (resp. γ_i) of length i, then for some element $T \in [\![\mathcal{G}]\!]$, $T(\gamma_i) = a_i$ and if there exists a run-tree R over T, then $R(\zeta_i) = (\gamma_i, q_i)$ and $R(\zeta) = (\gamma, q)$.*

(ii) $\mathsf{history}(J')$ *restricted to only those judgements $\Gamma^o \vdash F : \theta^o$ where $F \in \mathcal{N}$ is a sequence $K_1 \cdots K_{m-1}$ such that if $J' = K_m$ then each $K_j (j \in [1..m])$ is related to a subterm t_j of some u_j where $S \to^+ u_1 \to^+ \cdots \to^+ u_m$ and each $\to^+$ contains exactly one reduction due to $\mathcal{R}$ (possibly many due to case). In particular, $u_m|_\zeta = t_m$ and where there are i judgements of the form $\Gamma^o \vdash a : \theta^o \in \mathsf{history}(K_j)$, $u_j|_{\zeta_i} = t_j$.*

Proof. By extension of the induction in Lemma 3.4. The definition of reify is updated to account for judgements whose open types end in a type variable as follows:

$$\mathsf{reify}(\Gamma^o \vdash t : \alpha_1 \to \cdots \to \alpha_n \to \beta) := \mathsf{reify}(\Gamma_1^o \vdash \mathsf{case}(s', t_1', \cdots, t_n') : \theta^o)$$

where

$$s' = \mathsf{resolve}(\Gamma^o, t)\ \mathsf{intro}(\alpha_1) \cdots \mathsf{intro}(\alpha_n)$$

$$\frac{\Gamma_1^o \vdash s : \beta \qquad \cdots}{\Gamma_1^o \vdash \mathsf{case}(s, t_1, \cdots, t_n) : \theta^o}$$

- A-Fun. From the induction hypothesis there is a witness of the form $t_m = F\,\tilde{v}$. The new t_{m+1} is

$$\mathsf{reify}(\{x_v : \alpha_v \mid v \in \tilde{v}\} \vdash s : q) = s[\tilde{v}/\tilde{x}]$$

  where $F\ \tilde{v} \to s[\tilde{v}/\tilde{x}]$ $(\lambda\tilde{x}..s \in \mathcal{R}(F))$. The inductive t_m is a subterm of u_m and t_{m+1} is a subterm of u_{m+1} (where $u_m \to u_{m+1}$ by reducing the redex t_m). Then $u_{m+1}|_\gamma = t_{m+1}$. As we are still talking about the same location in the run tree (ζ), it must still be labelled (γ, q), which is the type in the new judgement.

- A-Cas. In this case t_m, ζ and γ are unchanged. From the induction hypothesis v_m is $\mathsf{reify}(\Gamma^o \vdash \mathsf{case}(s, v_1, \cdots, v_n) : \theta^o)$ and now:

$$\mathsf{reify}(\Gamma^o \vdash s : \beta) = \mathsf{reify}(\Gamma^o \vdash \mathsf{case}(\mathsf{resolve}(\Gamma^o, s), v_1, \cdots, v_n) : \theta^o)$$

  which are the same as resolve is defined in terms of substitution which is idempotent.

- A-Uni. If b_i was already in $\Theta(\beta)$ the result is immediate. Otherwise the required application of A-β causes us to jump to the introduction of β. The new t_m is the result of reducing the case at the introduction of β so that the length of the reduction sequence $u_{m-1} \to^+ u_m$ increases by one:

$$\begin{aligned}\mathsf{reify}(\Gamma^o \vdash b_i : \beta) &= \mathsf{reify}(\Gamma^o \vdash \mathsf{case}(\mathsf{resolve}(\Gamma^o, b_i), v_1, \cdots, v_n) : \theta^o) \\ &\to \mathsf{reify}(\Gamma^o \vdash v_i : \theta^o)\end{aligned}$$

□

Theorem 4.13 (Correctness)**.** *Let $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$ be a HORSC and $\mathcal{A} = \langle \Sigma, Q, \delta, q_0\rangle$ a DTT.*

(i) If Algorithm 2 returns Yes *then $\mathcal{A}^\perp$ accepts every tree in $[\![\mathcal{G}]\!]$.*

(ii) If Algorithm 2 returns No *then $\mathcal{A}^\perp$ rejects some tree in $[\![\mathcal{G}]\!]$.*

(iii) Algorithm 2 terminates on every input.

Proof. (i) Immediate from Lemma 4.11.

(ii) Assume for contradiction that the algorithm returns No, but $\mathcal{A}^{\perp}$ accepts every member of $[\![\mathcal{G}]\!]$ so that there must exist a run-tree R. However, from Lemma 4.12, we know that there are words ζ, γ such that for some $T \in [\![\mathcal{G}]\!]$, $T(\gamma) = a$ and $R(\zeta) = (\gamma, q)$. The definition of a run tree requires that there must be a satisfying assignment for $\delta(q, a)$; as there is not we have our contradiction.

(iii) We again set N be the product of the number of non-terminals of $\mathcal{G}$ and the total number of types (of the corresponding kinds) of $\vdash_{\mathcal{A}}$. The algorithm then runs until every open judgement is at the end of a path in $\mathcal{D}$ of length N, at which point we are guaranteed that every path either ends in a closed leaf or contains a recurrence of a reified type binding. Reaching this point guarantees the existence of a complete cut and must eventually occur by extension of Lemma 3.6 (the extension is rather technical and so we defer it to Appendix A).

□

4.4 Implementation and Evaluation

In this section we will describe TravMC, an implementation of Algorithm 2 in F#. The huge worst-case complexity of the problem manifests in the very large number of possible types that can appear during the search, increasing the difficulty of finding a complete cut. We will explain a number of optimisations applied to the algorithm that help us to address this problem. We have carried out an empirical evaluation of TravMC, in which for comparison with other tools we have considered not just HORSC, but also standard HORS, which can be handled by our algorithm as a degenerate case. The implementation, and all the examples presented here, can be accessed through a web interface at `http://mjolnir.cs.ox.ac.uk/horsc/`.

Optimisations

The most powerful optimisation we implemented is *Actual Parameter Revisit Avoidance*. Consider a state $(\mathcal{D}, \Theta)$ of the algorithm and choose some node n labelled with open derivation Δ^o with $\Gamma^o \vdash F : \theta^o$ a leaf of Δ^o. If $\mathcal{R}(F) = \{\lambda\tilde{x}.s_1, \cdots, \lambda\tilde{x}.s_n\}$ then the children of n will include nodes labelled $(F, s : \theta^o, \Delta^o_i)$ for each $i \in [1..n]$.

Now we choose some formal parameter $x \in \tilde{x}$ (without loss of generality we consider the first in what follows). If x appears in two judgements $\Gamma_1^o \vdash x : \theta_1^o$ and $\Gamma_2^o \vdash x : \theta_2^o$ with $\mathsf{state}(\theta_1^o) = \mathsf{state}(\theta_2^o)$ in the children of n, then we can save some work. Let us say the respective open types are:

$$\theta_1^o = \alpha_1 \to \cdots \to \alpha_n \to X$$
$$\theta_2^o = \alpha_1' \to \cdots \to \alpha_n' \to X$$

where X is some type variable β or state q. When we close the first judgement using A-Var, A-α is triggered and we jump back to Δ^o to add a judgement for t below:

$$\frac{\overline{\Gamma^o \vdash F : \alpha \to \theta^o} \qquad \Gamma^o \vdash t : \theta_1^o \qquad \cdots}{\Gamma^o \vdash F\ t : \theta^o}$$

Now, when we close the second (and any subsequent) judgement(s) we optimise as follows:

(i) We do not trigger A-α, although we still update the instantiation map with $\Theta(\alpha) := \Theta(\alpha) \cup \{\theta_2^o\}$.

(ii) For any existing and future additions to $\Theta(\alpha_i)$ ($i \in [1..n]$) we add the same types to $\Theta(\alpha_i')$ and trigger A-α accordingly.

This optimisation is sound if x is at most order-1. To see why, observe that in the unoptimised case, the two judgements that would be added as siblings are $\Gamma^o \vdash t : \theta_1^o$ and $\Gamma^o \vdash t : \theta_2^o$. As $\mathsf{state}(\theta_1^o) = \mathsf{state}(\theta_2^o)$, the continued computation of the algorithm up to any point where A-Var is used on a judgement of the form $y : \alpha_i \vdash y : \theta_3^o$ will be identical for the two judgements. Therefore α_i and α_i' will be updated with the same type and note that with x being order-1, θ_3^o must be order-0 and so cannot interact with its context further. At higher orders, θ_3^o would contain further type variables and eliding the computation starting at $\Gamma^o \vdash t : \theta_2^o$ would leave no judgement $y : \alpha_i' \vdash y : \theta_4^o$ in a *different* location for a functional argument of x to in turn jump back to later.

Recalling the correspondence with traversals shown in Section 3.3, we can phrase it in those terms instead. Assume a traversal reaches a variable x in state q, and has previously visited another occurrence of x in state q. Then if on the previous occurrence the traversal subsequently visited the i-child of x in state q', then we can immediately traverse downwards to the i-child of the current x node, again in state q' and elide any intermediate computation.

The *canonical types* optimisation aids with the critical part of a *complete cut* (and thus termination) in finding two nodes with the same concrete type bindings. We can increase the chance of finding two such nodes by 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 $\theta_j \leqslant \theta_k$ (where $\leqslant$ is intersection type subtyping as defined by Barendregt et al. [1983][1]). 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 or a non-terminal symbol with n non-deterministic choices 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.

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 "Order", "Size" and "Result" in the table indicate the order, number of symbols in the scheme and result of the example respectively. The "H" and "G" columns contain timing data for Kobayashi's hybrid (TRecS version 1.32) and game-based algorithms (GTRecS version 0.10[2]). Those labelled "T" or "T_B" (resp. "T′") are for the algorithm introduced in this paper with (resp. without) the *Revisit Avoidance* optimisation at

[1]Note however, that we have a distinguished ω at each kind, which we write as $\top$, and so do not relate ω to $\omega \to \omega$.

[2]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.

				Kobayashi		TravMC		
Instance	Order	Size	Result	H	G	T	T_B	T′
example2-1	1	24	Y	2	1	34	0	33
fileocamlc	4	214	Y	8	1680	60	23	718
fileocamlc2	4	206	Y	7	1980	58	18	918
fileorder5-2	5	348	Y	109	–	201	167	–
filewrong	4	129	N	0	–	86	47	85
flow	4	33	Y	1	3	32	0	32
g35	3	86	Y	–	136	–	–	–
g41	4	65	Y	–	608	55	15	–
lock2	4	123	Y	10	–	64	23	132
m91	5	249	Y	39	–	429	381	–
order5	5	157	Y	5	–	62	8	46
order5-variant	5	164	Y	12	–	47	7	317
stress	1	64	Y	29	3	187	133	180

Table 4.5: HORS Model Checking comparison

order 1, the subscript B indicating a 'batch' processing mode. To ease comparison of the smaller numbers, all times are given in milliseconds. Where an algorithm did not terminate within 60 seconds this is indicated by "–". During our testing we found that if a tool did not terminate within a few seconds, it was very unlikely to return a result within any reasonable timeframe.

Table 4.5 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

HORSC	Order	Size	Result	TravMC	TravMC (HORS)
checknz	1	63	Y	46	36
checkpairs	1	182	N	53	93
filepath	1	4063	Y	1950	–
filter-nonzero	4	162	N	74	156
filter-nonzero-1	4	268	Y	1756	–
last	1	121	Y	71	45
map-head-filter	2	266	N	62	116
map-head-filter-1	2	492	Y	1080	1538
map-plusone	4	116	Y	83	161
map-plusone-1	4	143	Y	296	860
map-plusone-2	4	206	Y	4144	–
mkgroundterm	1	230	Y	179	96
risers	1	358	Y	113	127
safe-foldr1	2	413	Y	450	625
safe-head	2	304	Y	71	56
safe-init	2	520	Y	209	288
safe-tail	2	401	Y	88	74

Table 4.6: HORSC Model Checking results

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 [2011] 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 the earlier work of Ong and Ramsay [2011]. 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 B. 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 4.6, 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 speed of checking these HORSC model checking instances with an equivalent HORS (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 Section 4.1). The time to check the original instance is given in column "TravMC" and to check the encoding can be seen in the column "TravMC (HORS)". 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 [Mitchell and Runciman, 2008] 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.

RSFD	Order	Size	States	Result	TRecS	TravMC
gap_id	3	241	15	Y	248	15
homrep	4	123	2	Y	1767	7
merge_addr	1	35	3	Y	52	1
mult	1	34	2	Y	52	1
remove_b	2	46	4	Y	54	2
xhtmlm-drop-a	1	962	32	Y	1252	146
xhtmlm_id	1	726	32	Y	996	64
xhtmls-remove-meta	1	300	12	Y	277	9
xhtmlf_id	1	2842	50	Y	–	456

Table 4.7: RSFD Model Checking results

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 et al. [2010] model check *recursion schemes with finite data domains* (RSFD) as part of their work. 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 degenerate HORSC, our tool is also able to solve the RSFD model checking problem. We have compared the performance of our tool (column "TravMC") versus the TRecS (version 1.32) tool of Kobayashi *et al*. (column "TRecS") in Table 4.7. We have also included the number of states in the property automaton as for some of the examples checking properties of XML transformations it is very large (proportional to the number of supported tags). 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. As discussed in Section 4.1 it is not possible to represent general instances of control flow using finite data or non-determinism in RSFD without an additional CPS transform. However, even at higher orders or with a large number of automaton states, our tool can solve almost all the example instances.

Chapter 5

APT Model Checking

In the preceding chapters we have focused only on safety properties, expressible using deterministic trivial tree automata. In this chapter our central contribution is an attack on the original version of the HORS model checking problem from Definition 2.21:

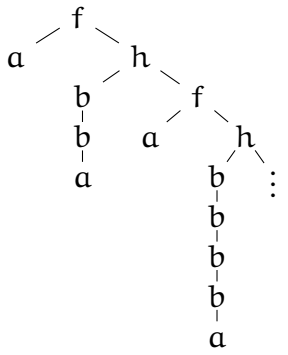
> Given a higher-order recursion scheme $\mathcal{G}$ and an alternating parity tree automaton $\mathcal{A}$, is the tree generated by $\mathcal{G}$, $[\![\mathcal{G}]\!]$, accepted by $\mathcal{A}$?

We will build on our algorithm from Chapter 3 to develop an algorithm for checking properties specified using unrestricted alternating parity tree automata. This involves two orthogonal extensions: (i) the transition function is no longer deterministic, and (ii) the acceptance condition is non-trivial. We will describe the type theory on which this extended algorithm is based, prove its correctness and give an empirical evaluation of the implementation with a comparison against the only other tool for solving the problem. This work was carried out by myself and published in the proceedings of SPIN 2014 [Neatherway and Ong, 2014].

Example 5.1. Our running example for this chapter will be the order-2 HORS $\mathcal{G}_4 = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where:

$$\Sigma = \{a \mapsto 0, b \mapsto 1, f \mapsto 2, h \mapsto 2\}$$
$$\mathcal{N} = \{S : o, F : (o \to o) \to o, H : (o \to o) \to o, B : (o \to o) \to o \to o\}$$
$$\mathcal{R} \begin{cases} S = F\ b \\ F = \lambda y.f\ a\ (H\ (B\ y)) \\ H = \lambda z.h\ (z\ a)\ (F\ z) \\ B = \lambda g x.g\ (g\ x) \end{cases}$$

and (a prefix of) the tree generated by $\mathcal{G}_4$ is shown below.

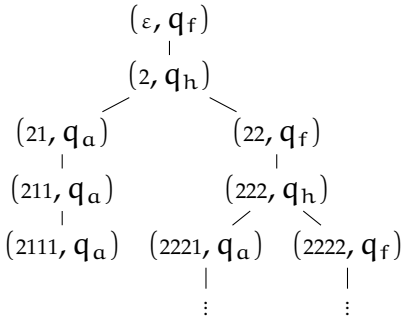


Each subsequent occurrence of h will have left subtree $b^{2^n}a$ with n increasing and unbounded; as such this tree is not regular.

$\mathcal{A}_5 = \langle \Sigma, \{q_f, q_h, q_a\}, \delta, q_f, \{q_f \mapsto 2, q_h \mapsto 2, q_a \mapsto 1\}\rangle$ where δ is as follows (omitting all that δ maps to f):

$$\begin{aligned}(q_f, f) &\mapsto (1, q_h) \vee (2, q_h)\\ (q_h, h) &\mapsto (1, q_a) \wedge (2, q_f)\\ (q_a, b) &\mapsto (1, q_a)\\ (q_a, a) &\mapsto t\end{aligned}$$

Thus $\mathcal{A}_5$ accepts a Σ-labelled tree T if in every path, f always has a child h, the right child of h is f, and the left child of h is a *necessarily finite* word in b^*a. A prefix of a run-tree of $\mathcal{A}_5$ over $[\![\mathcal{G}_4]\!]$ can be seen below. Notice that its structure reflects that the transition for (q_f, f) is disjunctive and the run-tree explores only the second child.



Without loss of generality we consider only *productive* recursion schemes (Definition 2.14) i.e. those schemes $\mathcal{G}$ where $[\![\mathcal{G}]\!]$ does not contain $\bot$. This technical convenience allows us to avoid the question of whether a divergent path of computation is accepted and handling this special case in the type system.

5.1 A Non-Weakening Type System

In this section we will describe the type theory due to Kobayashi and Ong [2009] that the algorithm is based on. First recall the definition of consistency for the type system for trivial automata: Fix a HORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ and DTT $\mathcal{A}$. We say that a type environment Γ is $(\mathcal{G}, \mathcal{A})$-consistent when:

(i) $\Gamma :: \mathcal{N}$

(ii) $\forall (F : \theta) \in \Gamma : \Gamma \vdash_{\mathcal{A}} \mathcal{R}(F) : \theta$

In contrast with these conditions, the second of which is essentially a rule for type-checking recursive calls, when the property has a non-trivial acceptance condition the definition of consistency becomes correspondingly more complex.

Example 5.2 (Unsoundness of trivial consistency)**.** Take the HORS induced by the single rewrite rule $S \to b\ S$ that generates the infinite tree b^ω. Imagine that we wish to check that the tree generated is finite, and use a single-state automaton with the transition $\delta(q, b) \mapsto q$ and priority function $\Omega(q) = 1$. The run-tree over b^ω contains a single path whose maximum priority is 1, and so it is rejected. However, using the definition of consistency above, the environment $\{S : q\}$ is consistent as witnessed by the following derivation:

$$\textsc{App}\ \frac{\textsc{Term}\ \dfrac{}{\{S : q\} \vdash b : q \to q} \qquad \textsc{Var}\ \dfrac{}{\{S : q\} \vdash S : q}}{\{S : q\} \vdash b\ S : q}$$

To prevent this we need some way of encoding the priorities of the automaton into the type system. We start with the following extended definition of types:

$$\frac{q \in Q}{q :: o} \qquad \frac{m_i \geqslant \max(\Omega(\theta_i), \Omega(\theta)) \quad \theta_i :: \kappa_1 \text{ (for all } i \in I) \quad \theta :: \kappa_2}{(\bigwedge_{i \in I}(\theta_i, m_i)) \to \theta :: \kappa_1 \to \kappa_2}$$

Note the additional component m of each type in an intersection on the left of an arrow, which will take on values in the image of the priority function Ω. Take a functional type without priorities, say $q_1 \wedge q_2 \to q$, which means that a term $\lambda x.t$ assigned this type will take an argument accepted from *both* states q_1 and q_2 and return a term accepted from state q. We can think of $\lambda x.t$ as a kind of tree context with two (or more) holes where the argument will be placed. With priorities, the type $(q_1, m_1) \wedge (q_2, m_2) \to q$ now records the maximum priority that would occur in a potential run-tree on the path from the root (labelled by state q as it is the return type) to the holes in the context. This intuition helps to motivate the requirement that m_1 in this type should be at least the greater of $\Omega(q_1)$ and $\Omega(q)$

as they will label the two ends of this path: q labels the root and q_1 labels the root of this hole in the tree context.

The rules for the type system are in Table 5.1 where we can see exactly how the priorities are introduced by the terminal symbols; in Example 5.2, the terminal symbol b would have had type $(q, 1) \to q$. There are a few other notable features of this system:

(i) The priority function is lifted from states to types so that $\Omega(\theta) = \Omega(\mathsf{state}(\theta))$.

(ii) Type environments are now bindings of non-terminals and variables to pairs of intersection types and priorities. That priority must be encountered prior to using the type binding in a leaf node with VAR, which is enforced by the APP rule using a function that lifts priorities in type environments:

$$\Gamma \Uparrow m = \{F : (\theta, \mathsf{max}(m, m')) \mid F : (\theta, m') \in \Gamma\}$$

(iii) The type system is generally non-weakening, that is to say that every element of a type environment must be used in a derivation. This is enforced by the environments being exactly size 0 and 1 in the rules TERM and VAR respectively. There is however a form of weakening in the ABS rule.

(iv) The TERM rule generalises that for the previous type systems to allow any satisfying assignment for the Boolean formula that is the right-hand side of the corresponding transition.

Example 5.3 (Example of typing $\mathcal{R}(H)$)**.** Take the non-terminal F from Example 5.1. We will show that $\mathcal{R}(H) = \lambda z.h\,(z\,a)\,(F\,z)$ can be assigned the type $((q_a, 1) \to q_a, 2) \to q_h$.

We abbreviate several types in the derivation:

$$\theta_h = ((q_a, 1) \to q_a, 2) \to q_h$$
$$\theta_f = ((q_a, 1) \to q_a, 2) \to q_f$$
$$\theta_a = (q_a, 1) \to q_a$$

In the following derivation, note that the leaves justified using VAR have a type binding with priority equal to that of the type itself as required. The binding for z has its priority increased by use of the APP rule when it is used as an argument for a term of higher priority: $z : (\theta_a, 1) \Uparrow 2 = z : (\theta_a, 2)$. This ensures that the fact that z is used *after* seeing priority 2 (in this case due to h) is included in the type signature of H (θ_h).

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

$$\frac{\begin{array}{c}\{(i, q_{ij}) \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k_i\} \text{ satisfies } \delta(q, a) \\ m_{ij} = \max(\Omega(q_{ij}), \Omega(q)) \text{ for each } i, j\end{array}}{\emptyset \vdash_{\mathcal{A}} a : \bigwedge_{j=1}^{k_1}(q_{1j}, m_{1j}) \to \cdots \to \bigwedge_{j=1}^{k_n}(q_{nj}, m_{nj}) \to q}\ \textsc{Term}$$

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

$$\frac{\Gamma, x : \bigwedge_{i\in I}(\theta_i, m_i) \vdash_{\mathcal{A}} t : \theta \qquad I \subseteq J}{\Gamma \vdash_{\mathcal{A}} \lambda x.t : (\bigwedge_{i\in J} \theta_i) \to \theta}\ \textsc{Abs}$$

Table 5.1: APT Intersection type assignment system

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash h : (q_a, 2) \to (q_f, 2) \to q_h} \quad \dfrac{}{z : (\theta_a, 1) \vdash z\, a : q_a}}{z : (\theta_a, 2) \vdash h\,(z\, a) : (q_f, 2) \to q_h} \quad \dfrac{\dfrac{}{F : (\theta_f, 2) \vdash F : \theta_f} \quad \dfrac{}{z : (\theta_a, 1) \vdash z : \theta_a}}{F : (\theta_f, 2), z : (\theta_a, 1) \vdash F\, z : q_f}}{F : (\theta_f, 2), z : (\theta_a, 2) \vdash h\,(z\, a)\,(F\, z) : q_h}}{F : (\theta_f, 2) \vdash \lambda z.h\,(z\, a)\,(F\, z) : \theta_h}$$

Definition 5.4 (Characterisation via parity games)**.** As shown by Example 5.2 we need a more sophisticated notion of consistency. Given an APT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \Omega\rangle$ and a recursion scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$, we define the associated parity game $\mathbb{G}(\mathcal{G}, \mathcal{A}) = \langle V_A, V_E, (S, q_0, \Omega(q_0)), E, \Omega'\rangle$ where

$$\begin{aligned} V_E &:= \{(F, \theta, m) \mid \theta :: \mathcal{N}(F), m \in \mathsf{dom}(\Omega)\} \\ V_A &:= \{\Gamma \mid \Gamma :: \mathcal{N}\} \\ E &:= \{((F, \theta, m), \Gamma) \mid \Gamma \vdash_{\mathcal{A}} \mathcal{R}(F) : \theta\} \cup \{(\Gamma, (F, \theta, m)) \mid F : (\theta, m) \in \Gamma\} \end{aligned}$$

$(S, q_0, \Omega(q_0)) \in V_E$ is the initial node, and the priority function Ω' maps (F, θ, m) to m and Γ to 0. The underlying directed graph has node-set $V_A \cup V_E$ and edge-set E.

Then $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}$ if, and only if, Éloïse has a winning strategy in the associated parity game $\mathbb{G}(\mathcal{G}, \mathcal{A})$, which we write $\vdash_{\mathcal{A}} \mathcal{G}$.

The parity game $\mathbb{G}(\mathcal{G}, \mathcal{A})$ can be understood intuitively as follows. The initial vertex, $S : q_0$, should be thought of as a challenge from Abelard to prove that S has type q_0. Éloïse responds by presenting a type environment Γ such that (guaranteed

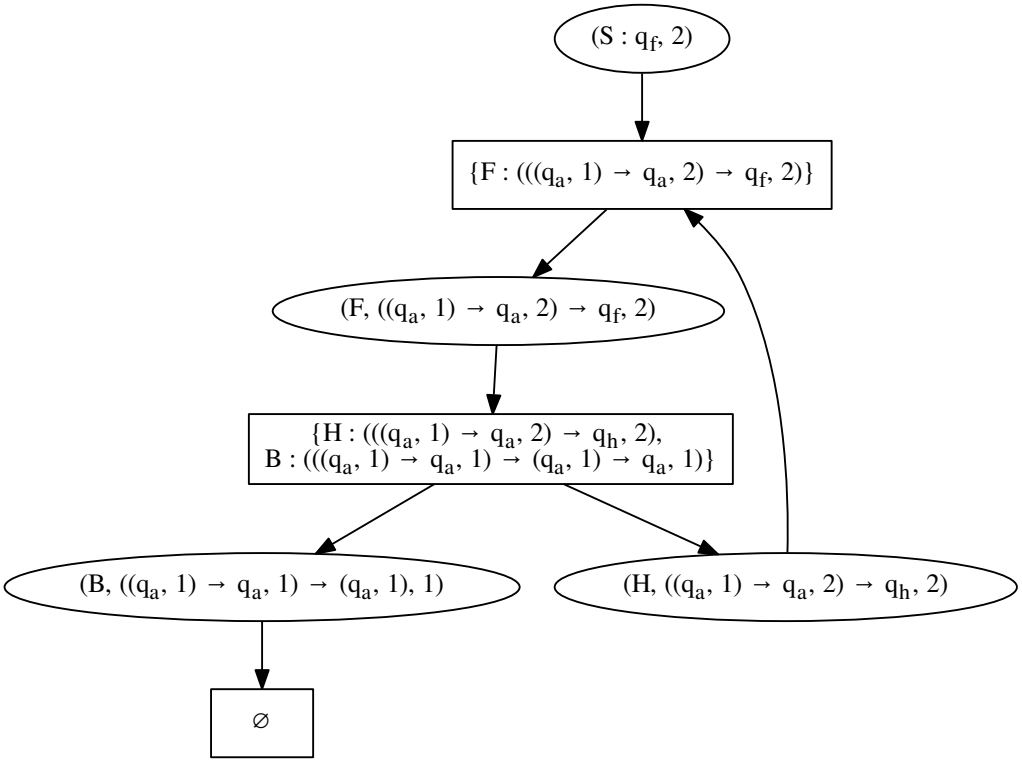


Figure 5.1: Witness for $[\![\mathcal{G}_4]\!] \in \mathcal{L}(\mathcal{A}_5)$

by the edge relation) $\Gamma \vdash \mathcal{R}(S) : q_0$. In turn, Abelard will choose some type binding $F : \theta$ from Γ and again challenge Éloïse to justify this binding. A finite play will be winning for Abelard if Éloïse cannot justify a type binding by proving that the right-hand side can also be assigned the same type, while Éloïse wins if she ever presents an empty type environment. Infinite plays reflect infinite paths in a run tree and so are won according to the parity condition.

In the degenerate case where every state is accepting (has even priority) as in the earlier chapters, every node in the game will have an even priority and so any infinite play is winning for Éloïse. This corresponds closely to the definition of consistent type environment used in Chapters 3 and 4.

A witness to the acceptance of $[\![\mathcal{G}_4]\!]$ by $\mathcal{A}_4$ can be seen in Figure 5.1, which contains a fragment of $\mathbb{G}(\mathcal{G}_4, \mathcal{A}_5)$ sufficient to exhibit a winning strategy for Éloïse. The elliptical (respectively rectangular) nodes are owned by Éloïse (respectively Abelard). The only infinite path in this fragment of the game has maximum priority 2, and so Éloïse wins. Note that Éloïse may have other choices in the full game not shown here, which would result in losing plays.

5.2 Algorithm

In contrast to the case where the property is specified using a trivial automaton, where all counter-examples to acceptance are finite, in the APT case we provide a semi-algorithm. The class APT is closed under complement (see Definition 2.20), and so for a problem instance $(\mathcal{G}, \mathcal{A})$, we can run the semi-algorithm for $(\mathcal{G}, \mathcal{A})$ and $(\mathcal{G}, \mathcal{A}^c)$ in parallel to recover completeness.

The definition of the instantiation map necessarily has to change in order to account for the priorities now present in the intersection types and the non-deterministic choices allowed in the automaton transition function (which means that terminal symbols may have multiple typings). We assume without loss of generality that the automaton transition function is given in disjunctive normal form. Then on the right-hand side of a transition, each $\mathbb{N} \times Q$ pair can be uniquely identified by a pair of naturals indicating which disjunct and which conjunct within that disjunct we have selected. In this way, a $\mathbb{N}$-labelled ranked tree can represent the choices made by the automaton, with each node being labelled by the index of the disjunct chosen and having a number of children equal to the number of conjuncts within that disjunct. We call such trees *choice trees* and refer to paths of choice trees using the function $\mathit{Path} = \mathbb{N}^* \to \mathbb{N}$. For notational convenience we use some further conventions. Paths are represented $\pi, \pi_1, \cdots$ and sets of paths (each defining a choice tree) as $\Pi, \Pi_1, \cdots$.

Definition 5.5 (Instantiation and Reification)**.** An instantiation map is extended to be a function $\Theta : A \to \mathcal{P}(\mathit{Path} \times \mathbb{P} \times \mathsf{cod}(\Omega))$, so that a type variable is now mapped to a set of triples consisting of a choice path, an open type as before, and a priority. A restriction operator $\lfloor_\Pi$ is used to reify open types with respect to a particular choice tree Π. We define $\Theta\lfloor_\Pi(\alpha) = \{\theta^o \mid (\pi, \theta^o) \in \Theta(\alpha) \wedge \pi \subseteq \Pi\}$. The reification map $\widehat{\Theta}\lfloor_\Pi$ is defined as follows:

$$\widehat{\Theta}_o(q) := q$$
$$\widehat{\Theta}_{\kappa_1 \to \kappa_2}(\alpha \to \theta^o) := \left(\bigwedge_{\theta_1^o \in \Theta\lfloor_\Pi(\alpha)} \widehat{\Theta}_{\kappa_1}(\theta_1^o)\right) \to \widehat{\Theta}_{\kappa_2}(\theta^o)$$

Example 5.6. 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:

$$\alpha_1 \mapsto \{(\pi_1, \alpha_3 \to q_2, 1), (\pi_1, \alpha_4 \to q_1, 1)\}$$
$$\alpha_2 \mapsto \{(\pi_1, q_1, 1)\}$$
$$\alpha_3 \mapsto \emptyset$$
$$\alpha_4 \mapsto \{(\pi_2, \alpha_5 \to q_0, 2)\}$$
$$\alpha_5 \mapsto \{(\pi_3, q_0, 3)\}$$

If $\Pi = \{\pi_1, \pi_2\}$ then

$$\widehat{\Theta}(\theta^o) = \bigwedge\{(\top \to q_2, 1), ((\top \to q_0, 2) \to q_1, 1)\} \to (q_1, 1) \to q_1$$

We also extend open judgements to take the form $\Gamma^o \vdash^\pi t : \theta^o$ where the new superscript π is a member of *Path*. The open type environments must now contain bindings to path, open type, priority triples (following the instantiation map). The restriction operator $\downarrow_\Pi$ is extended to open-type environments, judgements and pre-derivations, where it filters point-wise on $\pi \subseteq \Pi$ as before to yield environments, judgements and pre-derivations without paths. The reification map $\widehat{\Theta}$ in turn is lifted to apply the restriction operator before reifying as normal.

Algorithm 3: Model Checking

input : HORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, DTT $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \Omega \rangle$
output: Whether $[\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{A}^\perp)$
$\mathcal{D} :=$ singleton tree with label $((S, q_0, \Omega(q_0)), \emptyset \vdash_\varepsilon \mathcal{R}(S) : q_0)$
$\Theta := \{\alpha \mapsto \emptyset \mid \alpha \in A\}$
while *no* $\Pi \in \mathbb{T}$ *such that* $(\mathcal{D}\downarrow_\Pi)^j$ has a complete cut **do**
  $W :=$ unjustified$(\mathcal{D})$
  while $W \neq \emptyset$ **do**
    J := any element of W
    Apply matching rule A-Ξ possibly including A-Γ, A-α
    $W := W \setminus \{J\}$
    if $s \notin \mathcal{N}$ **then** $W := W \cup \{$all new unjustified judgements$\}$
  end
end
return YES

A state of the algorithm is a pair $(\mathcal{D}, \Theta)$ as in Chapter 3. $(\mathcal{D}, \Theta)\downarrow_\Pi$ behaves as expected, filtering on paths pointwise as before. In a particular state, there is a notion of a set of maximal choice trees $\mathbb{T}$ in terms of the paths found in the state tree by:

$$\Pi := \{\pi \mid \exists(\Gamma^o \vdash^\pi t : \theta^o) \in \mathcal{D}\}$$
$$\mathbb{T} := \left\{\bigcup T \mid T \in \mathcal{P}(\Pi), \bigcup T \text{ a tree}, \nexists T' \in \mathcal{P}(\Pi) \cdot \bigcup T' \text{ a tree} \wedge \bigcup T \subseteq \bigcup T'\right\}$$

Each member of $\mathbb{T}$ represents a consistent set of satisfying assignments to the automaton transition function and as such our new termination condition is triggered if any member in $\mathbb{T}$ induces a derivation tree with a complete cut. Algorithm 3 is modified in this way; note that there is also no possibility to exit for a No instance.

As given in Definition 3.6, $(\cdot)^\mathsf{j}$ selects the initial justified subtree of the (deterministic) state tree; here we apply the function only after a projecting with respect to some $\Pi \in \mathbb{T}$.

Definition 5.7 (Parity-consistent complete cuts)**.** When model checking against full APT, we generalise our previous definition of a *complete cut*. Recall that a cut C is a set of nodes that contains exactly one node on every path from the root of a tree. We now say that C is complete if for every $c \in C$:

(i) c is a leaf-node; or

(ii) there is an ancestor c' of c that has the same reified type binding as c *and* for all reified type bindings θ between c' and c inclusive, the greatest priority $\Omega(m)$ of their labels (of the form $F : (\theta^o, m), \Delta^o$) is even. In this case we say that c' is a *witnessing ancestor* of c.

As we will see in the proof of soundness, adjusting the definition in this way to take account of the priority information ensures that a complete cut gives rise to a type environment sufficient for Éloïse to exhibit a winning strategy in the corresponding parity game.

The rules of the algorithm for model checking against APT can be seen in Table 5.2. As before they are applied according to the shape of the unjustified judgement selected, with an additional rule, A-Γ, which is required in the presence of the non-weakening type system to keep the open derivations valid. This rule is triggered whenever a type binding is added to the environment (by A-Var or A-Fun) to allow justifying the open judgement with the Var typing rule and flows this type binding back up the typing derivation towards the root. If the maximal priority seen on the path between an operator and the use of its operand is m, then the priority associated with any type bindings used in the operand branch of the open derivation is lifted to be at least m, exactly as required by the App typing rule in Table 5.1.

The remaining changes to the rules are to handle the increased expressiveness of alternating automata (as opposed to purely deterministic). Unjustified judgements have a word w associated with them, recording the most recent satisfying assignment encountered in the automaton transition function. This is initialised to ε (as seen in Algorithm 3) and is extended every time the A-Cst rule is used. Notice that the update to Θ is tagged with a choice tree extended by setting the node w to i, for each satisfying assignment i. New judgements created from the same satisfying assignment will therefore be compatible (in the sense of defining a tree) and together be part of a maximal choice tree in $\mathbb{T}$.

Rule	If	Then
A-App	(i) $\emptyset \vdash^{\pi}_{w} t\ u : \theta^o$ (ii) α fresh	(i) $\dfrac{\emptyset \vdash^{\pi}_{w} t : \alpha \to \theta^o}{\emptyset \vdash^{\pi} t\ u : \theta^o}$
A-Fun	(i) $J = \emptyset \vdash^{\pi}_{w} F : \theta^o \quad (\in \Delta^o)$ (ii) $\mathcal{R}(F) = \lambda x_1 \ldots x_n.s$ (iii) $\Delta^o(\varepsilon) = \Gamma^o \vdash \lambda x_1 \cdots x_n.s : \theta^o_1$	(i) $\overline{\{(\pi, F : (\theta^o, \Omega(\theta^o)))\} \vdash^{\pi} F : \theta^o}$ (ii) Trigger A-Γ (iii) Add new rightmost child $(F : (\theta^o, m), \Delta^o_1)$ to Δ^o, $\Delta^o_1 = \dfrac{\emptyset \vdash^{\pi}_{w} s : q}{\vdots \quad \emptyset \vdash^{\pi} \mathcal{R}(F) : \theta^o}$
A-Var	(i) $J = \emptyset \vdash^{\pi}_{w} x_i : \theta^o \quad (\in \Delta^o)$ (ii) $\Delta^o(\varepsilon) = \Gamma^o \vdash^{\pi} \lambda x_1 \cdots x_n.s : \theta^o_1$ (iii) $\theta^o_1 = \alpha_1 \to \cdots \to \alpha_n \to q$	(i) $\overline{\{(\pi, x_i : (\theta^o, \Omega(\theta^o)))\} \vdash^{\pi} x_i : \theta^o}$ (ii) Trigger A-Γ; $(\pi, x_i : (\theta^o, m)) \in \Gamma^o$ (iii) $\Theta(\alpha_i) := \Theta(\alpha_i) \cup \{(\pi, \theta^o, m)\}$ (iv) Trigger A-α with w
A-Cst	(i) $\emptyset \vdash^{\pi}_{w} a : \theta^o$ (ii) $\theta^o = \alpha_1 \to \cdots \to \alpha_n \to q$ (iii) $\delta(q, a) \neq f$ (iv) For each $i \in I$, $\{(i_k, q_{ik}) \mid k \in K_i, 1 \leqslant i_k \leqslant n\}$ is a minimal set satisfying $\delta(q, a)$	(i) $\overline{\emptyset \vdash^{\pi} a : \alpha_1 \to \cdots \to \alpha_n \to q}$ (ii) $\forall i \in [1..n], \forall k \in K_i$: Let $m = \max(\Omega(q), \Omega(q_{ik}))$ Let $\theta^o_1 = (\pi \cup \{(w, i)\}, q_{ik}, m)$ $\Theta(\alpha_i) := \Theta(\alpha_i) \cup \{\theta^o_1\}$ and trigger A-α with k
A-α	(i) $\dfrac{\overline{\Gamma^o \vdash^{\pi_1} t : \alpha \to \theta^o_1} \quad \cdots}{J = \Gamma^o_1 \vdash^{\pi_1} t\ u : \theta^o_1}$ (ii) $(\pi, \theta^o, m) \in \Theta(\alpha)$ (iii) $\Gamma^o_2 \vdash^{\pi} u : \theta^o$ not a child of J (iv) Triggered with w	(i) $\dfrac{\overline{\Gamma^o \vdash^{\pi_1} t : \alpha \to \theta^o_1} \quad \emptyset \vdash^{\pi}_{w} u : \theta^o \quad \cdots}{\Gamma^o_1 \vdash^{\pi_1} t\ u : \theta^o_1}$
A-Γ	(i) $J = \Gamma^o \vdash^{\pi_1} t\ u : \theta^o_1$ (ii) $\Gamma^o_1 \vdash^{\pi_1} t : \alpha \to \theta^o_1$ child of J (iii) $\forall i \in I \cdot \Gamma^o_2 \vdash^{\pi} u : \theta^o_i$ child of J and $(\pi_i, \theta^o_i, m_i) \in \Theta(\alpha)$	(i) Set Γ^o to smallest set such that: a) $\Gamma^o_1 \subseteq \Gamma^o$ b) $(\Gamma^o_i \Uparrow m_i) \subseteq \Gamma^o$ (for each $i \in I$)

Table 5.2: Rules of algorithm

Correctness

It suffices to show that the algorithm is semi-complete: as we noted earlier, we will then run two copies of the algorithm in parallel against the property automaton and its complement.

Soundness

Lemma 5.1 (Alternating invariant)**.** *Given a state of the algorithm* $(\mathcal{D}, \Theta)$ *then for every* $\Pi \in \mathbb{T}$ *the restricted state* $(\mathcal{D}, \Theta)\downarrow_\Pi$ *satisfies the deterministic algorithm invariant Lemma 3.2. Let* $\mathcal{D} = \mathcal{D}\downarrow_\Pi$ *in:*

(i) $\mathcal{D}$ *contains valid open derivations, and*

(ii) for each derivation having root $\{F_1 : (\theta_1^o, m_1), \ldots, F_n : (\theta_n^o, m_n)\} \vdash t : \theta^o$ *and located at position* m *in* $\mathcal{D}$*, then for each* $1 \leqslant i \leqslant n$, $\mathcal{D}(mi) = (F_i : (\theta_i^o, m), \Delta_i^o)$ *and* $\Delta_i^o(\varepsilon) = \Gamma_i^o \vdash \mathcal{R}(F_i) : \theta^o$.

Proof. By induction over the rules of Algorithm 3. The base case is trivial and A-App is unchanged.

- A-Fun. The paths are unchanged. The new binding $(F : (\theta^o, \Omega(\theta^o)))$ allows justification of the leaf node with the Var rule. We also add $(F : (\theta^o, m))$ to each ancestor via A-Γ, where if the previous judgement was the operand in the App rule for a type with priority m' we apply $(- \Uparrow m')$ as required. The new child is labelled as required, with vacuous (correct) uses of the Abs rule.

- A-Var. The paths are unchanged. The new binding is added to the environment consistently as for A-Fun. The triggering of A-α carries the next choice, but the judgements are unchanged from the DTT case. The well-formedness of types requires that the addition of (θ^o, m) to α_i should have $m \geqslant \max(\Omega(q), \Omega(\theta^o))$. m is at least $\Omega(\theta^o)$ as A-Γ only increases the priority in type environments. m is also guaranteed to be at least $\Omega(q)$ as the path to the root taken by A-Γ is either leftmost in Δ^o so that $\mathsf{state}(\theta^o) = q'$, or at some point it is in operand position to operator type with $\mathsf{state}(\theta^o) = q'$ so that by the induction hypothesis, $(- \Uparrow m_i)$ lifts the priority to at least $\Omega(\theta^o)$.

- A-Cst. This is where new paths are introduced. Following the use of A-α, a number of new judgements will have been introduced, each with a new path $\pi \cup \{(w, i)\}$ for each member i of the minimal satisfying assignments I. As a result Π will no longer be in $\mathbb{T}$, being replaced by $\Pi_i = \Pi \cup \{(w, i)\}$, again for each $i \in I$ (the set of disjunctive choices). Considering an arbitrary Π_i we find that as in the A-Var case, the new state tree restricted by Π_i has Θ updated exactly in line with the addition of new judgements.

□

Definition 5.8 (Parity game induced by algorithm state)**.** In what follows, we assume a choice tree $\Pi \in \mathbb{T}$. For concision and clarity, we elide the restriction notation, and assume that each component of the state used below is that obtained by application of $\lfloor_\Pi$.

A state of the algorithm, $(\mathcal{D}, \Theta)$, and can be considered to induce a parity game, generated by the function $\mathbb{G}^-(\mathcal{D}, \Theta) = \langle V_E, V_A, (S, q_0, \Omega(q_0)), E, \Omega\rangle$. Intuitively each node of $\mathcal{D}$ becomes an Éloïse node, represented by its type binding, and the type environment required to prove it becomes an Abelard node. Formally we require that V_E, V_A and E are the smallest sets such that for each node $\mathcal{D}(n) = (F : \theta^o, \Delta^o)$ with root judgement $\Gamma^o \vdash t : \theta^o$ where:

(i) $\theta = \widehat{\Theta}(\theta^o)$,

(ii) $\Gamma = \widehat{\Theta}(\Gamma^o)$,

then

1. $(F, \theta, m) \in V_E$, and if n is justified then also
2. $\Gamma \in V_A$,
3. $((F, \theta, m), \Gamma) \in E$, and
4. $\{(\Gamma, (F', \theta', m')) \mid (F' : (\theta', m')) \in \Gamma\} \subseteq E$ (note that as Δ^o is closed, the nodes that will generate each (F', θ', m') must appear in $\mathcal{D}$).

Finally we inherit the priorities of $\mathbb{G}(\mathcal{G}, \mathcal{A})$: $\Omega'(F, \theta, m) = \Omega(m)$ and $\Omega'(\Gamma) = 0$.

Lemma 5.2 (Subgame soundness)**.** *Say we have two parity games $\mathcal{G}_1, \mathcal{G}_2$, where $\mathcal{G}_1$ is defined using a subset of the vertexes and edges of $\mathcal{G}_2$, Abelard's choice is not restricted in $\mathcal{G}_1$ and the priority functions agree. Then a winning strategy for Éloïse in $\mathcal{G}_1$ is necessarily also winning in $\mathcal{G}_2$. In this case we say that $\mathcal{G}_1$ is a* sound subgame *of $\mathcal{G}_2$.*

Proof. Consider (for $i \in \{1, 2\}$) $\mathcal{G}_i = \langle V_{i_E}, V_{i_A}, v_0, E_i, \Omega_i\rangle$. If $V_{1_E} \subseteq V_{2_E}, V_{1_A} \subseteq V_{2_A}, E_1 \subseteq E_2, \Omega_1 \subseteq \Omega_2$, and $\forall v_1 \in V_{1_A}, v_2 \in V_{2_E} \cdot (v_1, v_2) \in E_2 \Rightarrow (v_1, v_2) \in E_1$. Given a winning strategy for Éloïse f_E, by playing according to this strategy in the game $\mathcal{G}_2$ the current vertex will always be in $V_{1_E} \uplus V_{1_A}$. This is a trivial induction given the conditions above. Now assume for contradiction that there is a losing play for Éloïse in $\mathcal{G}_2$ even if she plays according to f_E. Then as all the vertexes visited are from $V_{1_E} \uplus V_{1_A}$ and the priority functions agree, this play must also be losing for Éloïse in $\mathcal{G}_1$. This contradicts our assumption that f_E is a winning strategy. □

Lemma 5.3 (Game soundness)**.** *If Éloïse has a winning strategy* f_E *for* $\mathbb{G}^-(\mathcal{D}, \Theta)$*, then* f_E *is also a winning strategy for* $\mathbb{G}(\mathcal{G}, \mathcal{A})$*.*

Proof. It suffices to show the relationship required by Lemma 5.2. Set $\mathbb{G}^-(\mathcal{D}, \Theta) = \langle V_\mathsf{E}, V_\mathsf{A}, (S, q_0, \Omega(q_0)), \mathsf{E}, \Omega\rangle$ and $\mathbb{G}(\mathcal{G}, \mathcal{A}) = \langle V'_\mathsf{E}, V'_\mathsf{A}, (S, q_0, \Omega(q_0)), \mathsf{E}', \Omega'\rangle$. Then clearly $V_\mathsf{E} \subseteq V'_\mathsf{E}$ and $\Omega \subseteq \Omega'$. Likewise $V_\mathsf{A} \subseteq V'_\mathsf{A}$. To see why, take an arbitrary node $\Gamma \in V_\mathsf{A}$. Then there must exist a node with a closed derivation $((\mathsf{F} : (\theta^o, \mathfrak{m})), \Delta^o)$ in $\mathcal{D}$ with $\widehat{\Theta}(\Delta^o(\varepsilon)) = \Gamma \vdash \mathcal{R}(\mathsf{F}) : \theta$. From consistency of the derivation tree (Lemma 5.1) we have $\Gamma \vdash \mathcal{R}(\mathsf{F}) : \theta$, and therefore $\Gamma \in V'_\mathsf{A}$. $\mathsf{E} \subseteq \mathsf{E}'$ follows. The final condition requiring that Abelard's choice is not reduced holds by construction of $\mathbb{G}^-(\mathcal{D}, \Theta)$, which has an edge from Γ to $(\mathsf{F}, \theta, \mathfrak{m})$ for every $(\mathsf{F} : (\theta, \mathfrak{m})) \in \Gamma$, exactly as in the definition of $\mathbb{G}(\mathcal{G}, \mathcal{A})$. □

Lemma 5.4 (Soundness)**.** *The existence of a complete cut implies Éloïse has a winning strategy in the parity game* $\mathbb{G}^-(\mathcal{D}, \Theta) = \langle V_\mathsf{E}, V_\mathsf{A}, (S, q_0, \Omega(q_0)), \mathsf{E}, \Omega\rangle$*.*

Proof. When a complete cut exists, for every non-leaf member of the cut there is an interior node with the same reified-type binding and the maximum priority between them is even. In this proof we let $\mathsf{env}(\mathsf{J}) = \Gamma^o$ where $\mathsf{J} = \Gamma^o \vdash \mathsf{t} : \theta^o$. We define a witnessing strategy, which relies on the strong relationship between vertexes in the game graph $\nu, \nu_1, \nu_2, \cdots$ and nodes in the state tree $\mathfrak{n}, \mathfrak{n}_1, \mathfrak{n}_2, \cdots$:

$$\begin{aligned}
\mathsf{game}_\mathsf{A}(\mathfrak{n}) &:= (\widehat{\Theta}(\mathsf{env}(\Delta^o(\varepsilon), \mathsf{state}(\theta^o)))) && \text{where } \mathcal{D}(\mathfrak{n}) = (\mathsf{F} : (\theta^o, \mathfrak{m}), \Delta^o)\\
\mathsf{game}_\mathsf{E}(\mathfrak{n}) &:= (\mathsf{F}, \widehat{\Theta}(\theta^o), \mathfrak{m}) && \text{where } \mathcal{D}(\mathfrak{n}) = (\mathsf{F} : (\theta^o, \mathfrak{m}), \Delta^o)\\
\mathsf{tree}(\Gamma) &:= \{\mathfrak{n} \mid \mathcal{D}(\mathfrak{n}) = (\mathsf{F} : (\theta^o, \mathfrak{m}), \Delta^o), \Gamma = \widehat{\Theta}(\mathsf{env}(\Delta^o(\varepsilon))), \mathfrak{n} \in \mathcal{D}^\mathsf{j}\}\\
\mathsf{tree}(\mathsf{F}, \theta, \mathfrak{m}) &:= \{\mathfrak{n} \mid \mathcal{D}(\mathfrak{n}) = (\mathsf{F} : (\theta^o, \mathfrak{m}), \Delta^o), \theta = \widehat{\Theta}(\theta^o), \mathfrak{n} \in \mathcal{D}^\mathsf{j}\}
\end{aligned}$$

The strategy for Éloïse is defined as:

$$\mathsf{f}_\mathsf{E}(\nu) = \begin{cases} \mathsf{game}_\mathsf{A}(\mathfrak{n}') & \exists \mathfrak{n} \in \mathsf{tree}(\nu) \cdot \mathfrak{n}' \text{ witnessing ancestor of } \mathfrak{n} \\ \mathsf{game}_\mathsf{A}(\text{largest } \mathfrak{n} \in \mathsf{tree}(\nu)) & \text{otherwise} \end{cases}$$

A maximal play where Éloïse plays according to f_E is guaranteed to either be finite and winning for Éloïse, or infinite. This is guaranteed as Éloïse can always play: consider an arbitrary node $\nu \in V_\mathsf{E}$. If there is an $\mathfrak{n} \in \mathsf{tree}(\nu)$ such that $\mathfrak{n}$ has a witnessing ancestor, then Éloïse plays to $\mathsf{game}_\mathsf{A}(\mathfrak{n})$; otherwise play $\mathsf{game}_\mathsf{A}(\mathfrak{n})$ for the largest $\mathfrak{n}$ (furthest from the root). In both cases, the new vertex ν' will have $\mathsf{tree}(\nu) \subseteq \mathcal{D}^\mathsf{j}$.

Observe further that if Éloïse does not encounter a witnessing ancestor then the largest $\mathfrak{n}$ is always further from the root than the previous. Take an arbitrary sequence of three vertices starting with Éloïse: $\nu_1 = (\mathsf{F}_1 : (\theta_1, \mathfrak{m}_1))$, $\nu_2 = \Gamma$ and $\nu_3 =$

$(F_2 : (\theta_2, m_2))$. Then the largest n in $\mathsf{tree}(\nu_1)$ is labelled with an open type binding that reifies to $F_1 : (\theta_1, m_1)$ and an environment that reifies to Γ. As $(\nu_2, \nu_3) \in \Gamma$, also $(F_2 : (\theta_2, m_2)) \in \Gamma$. As n is justified it must have a child labelled by each type binding in its reified environment Γ, which includes $F_2 : (\theta_2, m_2)$. One such child will be n', which must be in $\mathsf{tree}(\nu_3)$ and so the next largest node will be at least as large as n', which in turn is larger than n.

Finally, as we have established that finite plays are winning for Éloïse, we must show the same for infinite plays. We choose some infinite play $\langle \nu_i \rangle_{i \in \omega}$. From the previous paragraph some set of vertices that correspond (via tree) to members of the complete cut will be encountered infinitely often. Let us denote the set of members of the cut encountered by C. Every time a vertex corresponding to some $c \in C$ is encountered the vertex corresponding to the witnessing ancestor of c will be moved to. As a result, the vertices corresponding to the paths between the members of C and their witnessing ancestors comprise exactly those that appear infinitely often. From the definition of a complete cut the maximal priorities along each of these paths is even, and so the overall maximal priority is even. □

Lemma 5.5 (Alternating reduction)**.** *Given a* Yes *instance (and a corresponding run tree with automaton choices), there exists a* $\Pi \in \mathbb{T}$ *such that* $(\mathcal{D}, \Theta)\lfloor_\Pi$ *is in correspondence with the run tree in the sense of Lemma 3.4. If a step of the algorithm yields a new state* $(\mathcal{D}', \Theta')$ *then the new* $\Pi' \in \mathbb{T}$ *will be such that* $\Pi \subseteq \Pi'$ *and there are no paths* π *in* $(\mathcal{D}, \Theta)$ *such that* $\pi \in \Pi' \setminus \Pi$*.*

Proof. The proof follows the deterministic case of Lemma 3.4. Note that for the inductive cases, if the judgement being closed has $\pi \not\sqsubseteq \Pi$ then the tree restricted by Π will be unchanged and the invariant will hold trivially.

(i) Base case; $\emptyset \vdash^{\emptyset}_{\varepsilon} \mathcal{R}(S) : q_0$. By definition $\mathbb{T} = \{\emptyset\}$, and $(\mathcal{D}, \Theta)\lfloor_\emptyset = ((S : q_0, \emptyset \vdash \mathcal{R}(S) : q_0), \emptyset)$ as in the deterministic case.

(ii) A-App, A-Fun, A-Var. For these cases, the path on the new judgements is unchanged and so they are all present in the tree restricted by Π. The reasoning of Lemma 3.4 applies directly and Π remains a witness.

(iii) A-Cst. In this case the open judgement is $J = \Gamma^o \vdash^{\pi}_{w} a : \theta^o$ with $\mathsf{state}(\theta^o) = q$, which by assumption is part of $(\mathcal{D}, \Theta)\lfloor_\Pi$. Furthermore, J is related to a point β in the run tree and $r(\beta) = (\gamma, q)$. As we have assumed that $\delta(q, a)$ is in DNF, the children of the β in the run tree will necessarily be a superset of one of the disjuncts. If this disjunct is index i, then the new witness $\Pi' = \Pi \cup \{(w, i)\}$. Then each new judgement added from this disjunct will be present in the new tree restricted by T' and from the definition of a run-tree as in Lemma 3.4 each of these judgements is related to a child of β.

□

Lemma 5.6 (Priority context)**.** *For* $i \in \{1, 2\}$*, let* $\mathcal{D}(n_i) = (F_i : (\theta_i^o, m_i), \Delta_i^o)$ *be two labelled nodes of the state tree such that* n_2 *is a child of* n_1*. Then* m_2 *will be equal to the greatest priority observed in the computation path between the roots of the two derivations. Formally, if* $w_1 = \mathsf{history}(\Delta_1^o(\varepsilon))$ *and* $w_1 w_2 = \mathsf{history}(\Delta_2^o(\varepsilon))$*, then* $m_2 = \mathsf{max}(\{\Omega(\mathsf{state}(\theta^o)) \mid \Gamma^o \vdash t : \theta^o \in w_2\})$*.*

Proof. We prove a stronger result. Fix an open derivation Δ^o with root judgement $J_1 = \Gamma_1^o \vdash t_1 : \theta_1^o$ and an arbitrary judgement $J_2 = \Gamma_2^o \vdash t_2 : \theta_2^o$. Let $\Omega_\Gamma(J_1, J_2) = m$ where any binding $\xi : (\theta^o, m') \in \Gamma_2^o$ results in (via A-Γ) $\xi : (\theta^o, m) \in \Gamma_1^o$. Further let $\Omega_{\mathsf{his}}(J_1, J_2) = \mathsf{max}(\{\Omega_\Gamma(\mathsf{state}(\theta^o)) \mid \Gamma^o \vdash t : \theta^o \in w_2\})$ where $w_1 = \mathsf{history}(J_1)$ and $w_1 w_2 = \mathsf{history}(J_2)$ as before. Then we require:

(1) $\Omega_\Gamma(J_1, J_2) = \Omega_{\mathsf{his}}(J_1, J_2)$

(2) for each type variable α in θ_2^o, the maximum priority has not changed since it was introduced i.e. $\Omega_{\mathsf{his}}(\mathsf{intro}(\alpha), J_2) = \Omega_\Gamma(\mathsf{state}(\theta_2^o))$

We proceed by induction over the rules of the algorithm. The base case is trivial.

- A-App. Follows immediately from the induction hypothesis.

- A-Fun. This step was necessarily preceded by n uses of A-App:

$$\dfrac{\dfrac{\emptyset \vdash F : \alpha_1 \to \cdots \to \alpha_n \to \alpha'_1 \to \cdots \to \alpha'_m \to q}{\vdots}}{\emptyset \vdash F\, t_1 \cdots t_n : \alpha'_1 \to \cdots \to \alpha'_m \to q}$$

  and includes a further $n + m$ uses of the Abs typing rule in the new typing derivation. For every one of these judgements the type ends in q and so the priority does not change from $\Omega_\Gamma(q)$. Therefore (1) holds immediately and for each α_i (2) is trivial. For each α'_i, (IH-2) gives (2).

- A-Var. The single new derivation added by A-α is labelled J_2 below:

$$\dfrac{\dfrac{\dfrac{}{\vdash t : \alpha \to \theta_3^o} \qquad \vdash u : \theta_3^o\ (= J_2)}{\vdash t\, u : \theta_3^o\ (= J_3)}}{\dfrac{\vdots}{\vdash \mathcal{R}(F) : \theta_1^o\ (= J_1)}}$$

  By definition of A-Var, a judgement (say J'_2) in another derivation (with root J'_1) was closed prior to the addition of J_2.

For (1):

$$\begin{aligned}
\Omega_{\mathsf{his}}(J_1, J_2) &= \mathsf{max}\{\Omega_{\mathsf{his}}(J_1, J_3), \Omega_{\mathsf{his}}(J_3, J_1'), \Omega_{\mathsf{his}}(J_1', J_2)\} \\
&= \mathsf{max}\{\Omega_{\mathsf{his}}(J_1, J_3), \Omega_{\mathsf{his}}(J_3, J_1'), \Omega_{\mathsf{his}}(J_1', J_2')\} && (\mathsf{ty}(J_2') = \mathsf{ty}(J_2)) \\
&= \mathsf{max}\{\Omega_\Gamma(J_1, J_3), \Omega_{\mathsf{his}}(J_3, J_1'), \Omega_\Gamma(J_1', J_2')\} && (\text{IH-1}) \\
&= \mathsf{max}\{\Omega_\Gamma(J_1, J_3), \Omega_\Gamma(J_1', J_2')\} && (\text{IH-2}) \\
&= \Omega_\Gamma(J_1, J_2) && (\text{def}^{\text{n}}\ \text{A-}\Gamma)
\end{aligned}$$

where the final step relies on A-Var adding $(\theta^o, \Omega_\Gamma(J_1', J_2'))$ to $\Theta(\alpha)$.

The priority has not changed between J_2' and J_2, so (2) follows immediately from (IH-2).

- A-Cst. We consider an arbitrary member of some satisfying assignment, (i, q_i) where the justified judgement J_2' has the form $\emptyset \vdash a : \alpha_1 \to \cdots \to \alpha_n \to q$. The new judgement added by A-α is J_2:

$$\dfrac{\dfrac{\dfrac{}{\vdash t : \alpha_i \to \theta_3^o} \qquad \vdash u : \theta_3^o\ (= J_2)}{\vdash t\,u : q_i\ (= J_3)}}{\vdots}}{\vdash \mathcal{R}(F) : \theta_1^o\ (= J_1)}$$

The newly created judgement has the type q_i, so (2) holds vacuously. For (1) we split on whether α_i was introduced in the most recent series of A-App as for the A-Fun case. If yes, then it follows from (IH-1) and the addition of $(q_i, \mathsf{max}(\Omega(q), \Omega(q_i)))$ to $\Theta(\alpha_i)$. If no, then:

$$\begin{aligned}
\Omega_{\mathsf{his}}(J_1, J_2) &= \mathsf{max}\{\Omega_{\mathsf{his}}(J_1, J_3), \Omega_{\mathsf{his}}(J_3, J_2'), \Omega_{\mathsf{his}}(J_2', J_2)\} \\
&= \mathsf{max}\{\Omega_\Gamma(J_1, J_3), \Omega_{\mathsf{his}}(J_3, J_2'), \Omega_{\mathsf{his}}(J_2', J_2)\} && (\text{IH-1}) \\
&= \mathsf{max}\{\Omega_\Gamma(J_1, J_3), \Omega(q), \Omega_{\mathsf{his}}(J_2', J_2)\} && (\text{IH-2}) \\
&= \mathsf{max}\{\Omega_\Gamma(J_1, J_3), \Omega(q), \Omega(q), \Omega(q')\} && (\text{def}^{\text{n}}\Omega_{\mathsf{his}}) \\
&= \Omega_\Gamma(J_1, J_2) && (\text{def}^{\text{n}}\ \text{A-}\Gamma, \Theta(\alpha_i))
\end{aligned}$$

As the maximum priority does not change when using the A-Fun rule to create a new judgement, the statement of the lemma is a corollary. □

Lemma 5.7 ($\mathcal{G}$ productive implies eventually A-Cst)**.** *Running Algorithm 3 with input $(\mathcal{G}, \mathcal{A})$ where $\mathcal{G}$ is productive will always eventually fire the* A-Cst *rule along each path of computation.*

Proof. Assume for contradiction that this is not the case. The computation must be both infinite in length and continue in a single path, as only A-Cst can cause the number of open judgements to change (varying with the arity of the encountered terminal symbol). The result will be an infinite sequence of judgements $\langle J_i \rangle_{i \in \omega}$ where for all $i \in \omega$, $\mathsf{history}(J) = \langle J_j \rangle_{j \in [0..i-1]}$. We assume that our counter example judgement is J_k so that for all $i > k$, $\mathsf{tm}(J_i)$ is not headed by a terminal symbol. From Lemma 3.5 and the assumption that we never see A-Cst again, for any $i \in \omega$ there exists a $j > i$ such that $\mathsf{tm}(J_j)$ is headed by a nonterminal symbol. From Lemma 5.5 we can see that after J_k, the related term is always at some position β so that $S \to^* u$ and $u|_\beta$ is never headed by a terminal symbol. As we always see more A-Fun, the length of the reduction sequence from S always increases with every redex occurring at position β. Then after every finite number of reductions $u^\perp|_\beta = \perp$ and so $[\![\mathcal{G}]\!](\beta) = \perp$. This contradicts our assumption of productivity, and we are done. □

Theorem 5.8 (Semi-completeness)**.** *Let $\mathcal{G}$ be a HORS and $\mathcal{A}$ an APT. Algorithm 3 returns* Yes *if, and only if, $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}$.*

Proof. The forward direction is a direct result of Lemma 5.4. For the reverse direction, first fix an accepting run tree R. From Lemma 5.5 there will always be some choice tree $\Pi \in \mathbb{T}$ in correspondence with this run tree. Assume for contradiction that the algorithm does not terminate. In this case, there must always be a path τ in the state tree that does not include a witnessing ancestor (see Definition 5.7, else there would be a complete cut). In the limit, this path is infinite and from Lemma 3.6 it must be induced by some infinite sequence of judgements $\tau' = J_0 J_1 \cdots$ that constitute a path of computation of the algorithm.

From Lemma 5.7 τ' contains infinitely many uses of the A-Cst rule. Consider the priorities of the states seen in the judgements of τ' in the context of the correspondence with R. As R is accepting, the maximal priority of the states seen infinitely often (m) must be even, and this holds also of τ'.

Each node in the infinite path induced in the state tree, τ, is labelled with a priority. From Lemma 5.6 the maximal such priority seen infinitely often must also be m and so after some finite point in τ the nodes are labelled by no priority greater than m. However, the number of possible labellings of the nodes labelled with a binding having priority m is bounded by the finite number of intersection types. It follows that there must be at least two nodes in τ having the same label (including priority m) and there is no node labelled by a priority greater than m between them. Therefore one of these nodes is a witnessing ancestor of the other and we have a contradiction. □

Given an input instance $(\mathcal{G}, \mathcal{A})$, and running the semi-algorithm on $(\mathcal{G}, \mathcal{A})$ and $(\mathcal{G}, \mathcal{A}^c)$ concurrently, we obtain an algorithm.

5.3 Implementation and Evaluation

The implementation, TravMC2, builds on our earlier work described in Chapter 4. There is one other model checker in the literature, an extension of Kobayashi's TRecS tool: TRecS-APT [Fujima et al., 2013]. We have benchmarked TravMC2 against TRecS-APT using a number of examples, including all those from the TRecS-APT paper. We have modified other examples, abstractions of real programs, from the literature to check liveness properties. The tests were carried out on a 2.6GHz Intel Core 2 Quad processor running Windows 7 and the results can be seen in Table 5.3. For each problem instance we give its size in terms of the number of symbols in the HORS, the maximum order of any function symbol, and the number of priorities in the property automaton. The timings are given in seconds; when a tool did not terminate within 60 seconds this is indicated by "–". The tool and benchmarks are available to download from `http://mjolnir.cs.ox.ac.uk/web/horsapt/`.

We can see that TravMC2 is almost always faster than TRecS-APT across the benchmark suite. In particular, on the larger examples (over size 200) in our collection, TravMC2 does seem to outperform TRecS-APT consistently. Both tools time-out on examples past a certain size, indicating that more tuning and optimisation is required.

There are a number of differences between the algorithms used in the two tools, which may help to explain the difference in performance. TRecS-APT, like the original TRecS, builds a configuration graph by reducing the input HORS in tandem with the automaton transition function. Periodically, candidate types are extracted from the graph, which may contain type variables representing unknown behaviour from areas of the graph that have not yet been expanded. Eliminating the type variables causes a large number of additional candidates to be generated. The extracted types are then used to construct a sound subgame of the full parity game described in Section 5.1. Constructing such a parity game on each iteration may be expensive, especially with large numbers of candidate types; by comparison, TravMC2 never explicitly constructs a parity game.

We applied the optimisations described in Section 4.4, although we found that the additional book-keeping required for the actual parameter revisit avoidance optimisation above order 0 prevented it from improving the runtime significantly. This is due to the requirement to distinguish different paths that would have been created along the elided traversal fragment and bears further investigation.

New to this version of the tool was *trivial path conflation*. In the case where the property is specified using a trivial automaton, it is possible to safely confuse paths of exploration corresponding to different non-deterministic automaton choices. Observe that for a trivial automaton, every counter-example must be finite. We record those paths Π_{bad} that are conjunctive with any path that has encountered a

Benchmark	Size	Order	Priorities	Result	TravMC2	TRecS-APT
file	74	3	2	Y	0.09	0.21
imperative	79	3	2	Y	0.11	0.30
imperative-awt	82	3	2	Y	0.11	0.30
gcalloc	84	2	3	Y	0.10	0.07
reverse	95	2	2	Y	0.10	0.07
lock1	100	4	1	Y	0.09	0.09
pgm	122	4	2	N	0.27	0.69
merge	135	2	2	Y	0.13	0.52
homrep	142	3	2	Y	0.44	0.36
bsort	146	2	2	Y	0.11	0.22
intercept	151	4	2	Y	0.13	0.23
intercept-awt	155	4	2	Y	1.74	0.28
var-dwt	160	5	2	Y	0.16	1.57
order5-variant-awt	163	5	2	Y	0.17	2.52
loop-dj-2	181	5	5	N	2.12	0.49
fileocamlc-awt	226	4	2	N	0.36	2.02
twofiles	289	3	3	N	0.17	1.10
twofilesexn	296	3	3	Y	0.20	0.95
fileocamlc	311	4	3	Y	0.33	6.59
merge3	538	2	2	Y	1.02	–
map-plus-one	583	5	2	N	1.14	–
dna	699	2	4	Y	4.28	–
map-plus-one-1	854	5	2	N	31.02	–
search-e-church	1303	6	2	N	1.59	–
fold-right	1855	5	1	Y	–	–

Table 5.3: TravMC2 Benchmarks

potential counterexample i.e. reached a judgement $\Gamma^o \vdash^\pi a : \tilde{\alpha} \to q$ where there is no satisfying assignment for (q, a) in the transition function of the automaton. Then, when carrying out the termination check, we consider all paths not in Π_{bad} conjunctively, as long as the resulting set includes as least one choice tree (if it does not, then we can return No).

The soundness of this optimisation relies on the fact that a trivial automaton where every disjunction in the transition function is replaced with a conjunction defines a smaller tree language. Removing certain paths from the choice tree corresponds to an unfolding of the automaton, adding a new initial portion with some transitions having a smaller set of satisfying assignments.

As every counter-example is finite, the (incorrect) type information they provide is gradually filtered out until eventually, even if there are infinitely many counter-

examples, all those remaining are of such length that the tool is guaranteed to find a consistent prefix of the derivation tree that does not include any bad type information. Recall the proof of completeness in Chapter 3 (Theorem 3.7). In this setting we are guaranteed to terminate for the same reason – eventually we will see a repetition of types labelling the nodes along every path in the state tree. The generalised termination condition degenerates cleanly to that from Chapter 3 when the property automaton has a trivial acceptance condition.

This optimisation is a powerful tool to combat the many paths of exploration introduced by non-deterministic choices.

Benchmark	Size	Order	Result	T_{opt}	T	TRecS-APT
fileocamlc	218	4	Y	0.62	–	–
fileocamlc2	210	4	Y	0.51	–	–
merge2	454	2	Y	1.78	–	4.43
order5-2	141	5	Y	2.70	–	–
rev	109	2	Y	0.16	37.76	0.41
twofiles	143	4	Y	0.12	0.19	3.19

We investigated the effect of trivial path conflation and the results can be seen in the table above, which contains a number of examples with disjunctive choices added to the property. Timings for this optimisation are in the column "T_{opt}" compared with the standard algorithm in "T". The results are quite striking and underline the difficulty of efficiently maintaining the non-deterministic choices independently.

Chapter 6

Conclusions

6.1 Summary

In this dissertation we have explored the connection between Ong's *traversals* induced by the fully-abstract game semantics of higher-order recursion schemes and Kobayashi's system of intersection types. We have given a new model checking algorithm inspired by traversals, but working directly with intersection types. We have further shown that the computation of this algorithm as it builds typing derivations is isomorphic to the tree of traversals for the same recursion scheme. This fresh insight shows that the two points of view are really two sides of the same coin, and that the interested researcher should feel free to approach the area from the direction he feels most comfortable without worrying about missing intuition. A pleasant property of our algorithm is that a state of the algorithm (consisting of a tree of typing derivations and a reification map) constitutes a proposed certificate that the scheme satisfies the given property. As a result, it is simple to extract a witnessing type environment from a terminating state of the algorithm.

Motivated by the problem of verification of functional programs, we have proposed a new target for abstraction procedures — *higher-order recursion schemes with cases* — which include finite data domains and non-determinism. These are both key elements of Boolean programs, which are a common output of abstraction of first-order imperative programs. By characterising the model checking problem for HORSC (and properties specified using deterministic trivial tree automata) using a novel intersection and union type system we have proved that the problem is decidable. We have extended our decision procedure for HORS to HORSC and implemented it as a tool, TravMC. Building on earlier work of Ong and Ramsay, a CEGAR loop for verification of higher-order programs, we have used our model checker to verify properties of functional programs up to a few hundred lines of code. Previous work on higher-order model checking has either used only a

very restrictive form of data (RSFD) or elided it entirely, choosing to represent data using higher-order functions. In our empirical analysis we have started to explore the trade-offs between these representations. We have shown that for the test cases we have considered so far, the HORSC representation is model-checked significantly more quickly, which offers strong supporting evidence for the use of HORSC.

The checking of safety properties is useful, but more complex *liveness* properties ("something good must eventually happen") represent an important additional class that is also very relevant to practice. We further developed our tool to allow the checking of properties specified using *alternating parity tree automata*, a very expressive formalism. The results of benchmarking TravMC2 against the existing competitor TRecS-APT are encouraging, and give reason to proceed with the development of a verification framework involving reduction to HORS/APT model checking.

6.2 Further Directions

Extending HORSC

Our claim that HORSC are a suitable target for abstraction of higher-order functional programs is supported by our empirical results, which show the cost of model checking equivalent HORS to be significantly higher. Although there is often a benefit to working with the most compact formalism possible, as it can simplify both proof and implementation work, translating to simpler representations involves a loss of information about the structure of the input. Our introduction of union types to describe the behaviour of terms that may non-deterministically reduce to members of a finite data domain is a first step towards handling more complex control flow structures and call-by-value programs.

However, we have so far only considered relatively small input programs. This is partly due to the application of our model checker as a backend for an abstraction-refinement procedure that produces *weak pattern-matching recursion schemes* rather than HORSC, and then applies a rather complex encoding. The weak pattern-matching control construct is similar to pattern matching in Haskell, except that the pattern-match variables may not be used on the right-hand side of the match. To encode this using HORSC, we have to use a finite data domain equal in size to the number of discrete patterns that may appear in the program. If, for example, a program contained the following simple function:

```
F [] = t1
F (0:_) = t2
F _ = t3
```

then the HORSC translate will have to be based on the finite domain containing empty lists, non-empty lists headed by zero and all other non-empty lists, for a size of three. The CEGAR loop of Ong and Ramsay [2011] that we employed refines the abstraction by unfolding the patterns used on the left-hand side. As weak pattern-matching does not allow pattern-match variables to appear on the right-hand side, pattern unfolding offers an alternative way to propagate information to the right-hand side. Unfolding the patterns increases the size of the finite domain needed to represent terms to be matched against and the number of possible union types is exponential in the size of the finite domain. The effect of this is clear to see in our experiments, where the increase in runtime associated with a refinement step was often an order of magnitude.

When checking reachability properties it is safe to represent all terms matching a particular pattern using a single atom, but when checking properties of the output term computed by a program this no longer suffices. In this case our encoding to HORSC represents every term of the program as an atom of kind d, which is available for control flow via the case construct, and a term of kind o, which can be used to construct the output term (our encoding always generates a HORSC with a start symbol of kind o). For those examples where the property to be checked coincides with a pattern used in the program this is not necessary, but this does not occur in general. As HORSC do not include pairs natively, we were forced to use a CPS encoding for these examples. In this way pairs can instead be represented as two separate arguments, and the need to return pairs is removed. For example, a function from naturals to naturals would be represented as $(\mathsf{d} \times \mathsf{o}) \to (\mathsf{d} \times \mathsf{o})$ if we had product types in HORSC and instead becomes

$$\mathsf{d} \to \mathsf{o} \to (\mathsf{d} \to \mathsf{o} \to \bullet) \to \bullet$$

which is has an additional continuation argument to which the two representations of the term can be passed, rather than returning them to the calling context.

Model Checking for weak PMRS

A more ambitious extension would be to look at extending the algorithm to check weak PMRS directly. Even without the CPS transform, the HORSC encoding of a given weak PMRS is significantly larger. However, it is not clear how to extend the model checking algorithm in this case. Consider the setting of natural numbers modelled as an algebraic datatype (with constructors s and z) and the simple set of patterns $\{z, s\ z, s\ (s\ x)\}$, which match 0, 1 and 2 or greater. In this case we would introduce a new base type for each pattern, say p_z, p_{sz} and p_{ss}, and assign types to

the constructors as follows:

$$z : p_z$$
$$s : p_z \to p_{sz}$$
$$s : p_{sz} \to p_{ss}$$
$$s : p_{ss} \to p_{ss}$$

An important property of the algorithm when running on HORS is that the return type of every judgement is known. In the extension to HORSC, when the current open judgement contains a term with return kind d this is no longer the case, as we have a type variable in the rightmost position. The limitation of the constructors of return kind d to be order-0 means that the return type will not have to change as it does when encountering a terminal of kind o. Consider however an open judgement for a term when we are trying to determine which pattern(s) it matches:

$$\frac{\dfrac{\vdash t : ??}{\vdash s\ t : \beta}}{\vdash \mathsf{case}(s\ t, u_1, u_2, u_3) : q}$$

It is natural to use a fresh type variable that will be updated as we discover that the scrutinee matches particular patterns just as in the HORSC case. However, without β having any contents to guide the search, when we encounter a term headed by s, the only way to proceed is by introducing a fresh type variable again. Without associating any of the intermediate data learned with the type variables, any non-terminals encountered will quickly lead to a complete cut incorrectly being found, as each call has a reified return type of $\bot$ ($= \bigvee \emptyset$). One possible approach would be to create an open judgement speculatively for each pattern of the correct kind:

$$\frac{\vdash s\ t : p_z \qquad \vdash s\ t : p_{sz} \qquad \vdash s\ t : p_{ss}}{\vdash \mathsf{case}(s\ t, u_1, u_2, u_3) : q}$$

which would preserve the property of the return type being concrete, but would suffer from analysing scrutinee terms multiple times.

Call-by-value programs

Both HORSC and, to a greater extent, weak PMRS embody some aspects of call-by-value computation. This occurs when the order of evaluation is forced by a control flow construct, and is also visible in the addition of union types to the type system. It would be interesting to consider a type system and model checking algorithm for true call-by-value schemes (or innermost-outermost reduction). Tsukada and

Kobayashi [2014] have considered the analysis of such schemes, but their type system characterises *failure*, rather than safety, which complicates the construction of a model checking algorithm. A compact type system characterising safety remains an open problem, as does an efficient model checking algorithm for call-by-value HORS.

Flow-accelerated HORS/APT model checking

Our implementation of a HORS/APT model checker, TravMC2, has yielded encouraging results, showing that despite the worst-case complexity of the problem we can check properties of schemes of a non-trivial size and order. We have also seen that for the larger examples in our benchmark set, our checker outperforms the existing competitor TRecS-APT. However, our analysis further shows that our approach suffers when significant non-determinism is combined with a non-trivial acceptance condition in the property, although for properties specified using a trivial automaton (which allow us to 'conflate' the different choices) we can recover much of the lost performance. Our recent joint work with Ong and Ramsay [Ramsay et al., 2014] on Preface offers a potential answer to this problem. The tool Preface solves the HORS/ATT model checking problem using techniques of flow analysis in the mould of 0-CFA [Shivers, 1991] and Jones and Andersen [2007], with the innovation of distinguishing terms according to the intersection types we can assign to them.

Computation of this flow-accelerated algorithm is organised into rounds, constructing a graph that over-approximates the reduction of the HORS in parallel with running the automaton and then extracting types from regions of the graph that represent trees definitely accepted or rejected by the automaton from a given state. It seems likely that lifting the technique to the HORS/APT model checking problem will be possible, although the accepting and rejecting regions will need to be defined using a parity game. In the trivial case, unless the current type environment is strong enough to witness acceptance, there will always be a non-empty rejecting region corresponding to some finite counterexample trace (spurious or otherwise). However, when working with APT properties, the non-determinism in the property can obscure such regions, and so a more precise approach to the flow analysis will be needed.

Importantly, the algorithm behind Preface conflates different automaton choices as we did here in the "trivial path conflation" optimisation, pruning them away only when it is clear that they will not be satisfied. Lifting this behaviour to non-trivial properties is difficult as we can no longer rely on finite counterexamples to acceptance along a particular path. Indeed, while it does not seem possible for our traversal-based algorithm, we believe that it can be done for an extension

to Preface, preserving its efficiency even when combining non-determinism and non-trivial acceptance conditions.

APT Abstraction-refinement

There have been several approaches proposed for reducing the verification of safety properties of higher-order functional programs to higher-order model checking [Ong and Ramsay, 2011; Kobayashi et al., 2011], and for checking termination of such programs using binary reachability due to Kuwahara et al. [2014]. We would like to investigate the reduction of verification of liveness properties of higher-order functional programs to the HORS/APT model checking problem perhaps by proving fair termination of a program augmented with a non-constraining progress monitor as described by Balaban et al. [2007].

Verification of Haskell Programs

Verification is in increasing demand in industry, but it is key to keep programmer burden low. Our techniques apply to higher-order pure functional computation and so Haskell programs are the natural target. As the Core language used by the Haskell compiler GHC is an extension of System F [Weirich et al., 2013], which is much closer to PMRS, this may provide a viable route for translation. As a backend model checker, we would like to either use a checker built on an extension of HORSC, as described above, or the Preface tool, which has so far shown itself to be very robust with regards to scalability.

We would also aim to demonstrate how our techniques can be included as a light-weight static analysis method into the build-compile loop. Tools such as hLint[1] and GHC's `-Wall` flag offer cheap checks of relatively shallow properties, frequently helping to catch bugs. Conversely, many approaches to verification require extensive programmer interaction to prove deeper properties. We intend to offer a middle ground, requiring little to no annotation of the source code, and quick runtimes for checking of properties such as pattern-match safety that are simple to describe. This would make running the tool an ideal step in automated test suites and continuous integration scripts. We will aim to find an industrial partner to ensure our tool is relevant to practice.

[1] `http://hackage.haskell.org/package/hlint`

Appendix A

Well-foundedness of HORSC Reduction

Definition A.1. Fix a HORSC $\mathcal{H} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ with finite domain $B \subseteq \Sigma$. Without loss of generality we assume that the non-data terminals Σ are kinded only using o. Then we define:

$$h(\mathcal{H}) = \mathcal{G} = \langle \Sigma \setminus B, \mathcal{N}', \mathcal{R}', S \rangle$$

where:

$$\mathcal{N}' = \{F : H(\kappa) \mid F : \kappa \in \mathcal{N}\} \cup \{B_i : o^{|B|} \to o \mid b_i \in B\}$$
$$\mathcal{R}' = \{F \to \lambda \tilde{x}.h(t) \mid \lambda \tilde{x}.t \in \mathcal{R}(F)\} \cup \{B_i \to \lambda x_1 \cdots x_{|B|}.x_i\}$$

and for kinds:

$$h(o) = o$$
$$h(d) = o^{|B|} \to o$$
$$h(\kappa_1 \to \kappa_2) = h(\kappa_1) \to h(\kappa_2)$$

and terms:

$$h(a) = a \qquad (a \in \Sigma \setminus B)$$
$$h(b_i) = B_i \qquad (b_i \in B)$$
$$h(F) = F \qquad (F \in \mathcal{N})$$
$$h(x) = x \qquad (x \in \mathcal{V})$$
$$h(t_1\ t_2) = h(t_1)\ h(t_2)$$
$$h(\mathsf{case}(s, t_1, \cdots, t_n)) = h(s)\ h(t_1) \cdots h(t_n)$$

We take $\twoheadrightarrow$ to be the union of the standard HORS reduction $\to_{\mathcal{G}}$ with the relation

$$\{(a\ s_1 \ldots s_{\mathsf{arity}(a)}, s_i) \mid a \in \Sigma, 1 \leqslant i \leqslant \mathsf{arity}(a)\}.$$

Lemma A.1. *The function* h *is a bisimulation relation between* $\mathcal{H}$ *and* $h(\mathcal{H}) = \mathcal{G}$ *under outermost reduction i.e.* $t \twoheadrightarrow_{\mathcal{H}} t' \iff h(t) \twoheadrightarrow_{\mathcal{G}} h(t')$.

Proof. By induction on the structure of t, which is ground (contains no free variables).

- $t = a\ s_1 \cdots s_n$, $h(t) = a\ h(s_1) \cdots h(s_n)$. Then for each $i \in [1..n]$, $t \twoheadrightarrow_{\mathcal{H}} s_i$ and $h(t) \twoheadrightarrow_{\mathcal{G}} h(s_i)$ as required.
- $t = F\ s_1 \cdots s_n$, $h(t) = F\ h(s_1) \cdots h(s_n)$. For each $\lambda\tilde{x}.s \in \mathcal{R}(F)$, $\lambda\tilde{x}.h(s) \in \mathcal{R}'(F)$. Then choose one such arbitrarily so that $t \twoheadrightarrow_{\mathcal{H}} s[\tilde{s}/\tilde{x}]$ and $t \twoheadrightarrow_{\mathcal{G}} h(s)[\widetilde{h(s)}/\tilde{x}]$. By simple induction on h over terms, $h(s)[\widetilde{h(s)}/\tilde{x}] = h(s[\tilde{s}/\tilde{x}])$ as required.
- $t = \mathsf{case}(s, t_1, \cdots, t_n)$, $h(t) = h(s)\ h(t_1) \cdots h(t_n)$. Consider first the situation where $s = b_i$ $(\in B)$ so that $h(t) = B_i\ h(t_1) \cdots h(t_n)$. Then $t \twoheadrightarrow_{\mathcal{H}} t_i$ and from the definition of $\mathcal{R}'$ $h(t) \twoheadrightarrow_{\mathcal{G}} h(t_i)$. If $s \notin B$ then necessarily we reduce s and we appeal to the induction hypothesis.

□

Lemma A.2. *If there exists a finite N such that all paths in the state tree D induced by a path of computation are bounded by N, then this path of computation is finite.*

Proof. As before we appeal to Kobayashi and Ong [2009, Lemma 4.8, Appendix B], who showed that if a reduction sequence of a non-deterministic HORS is well-founded (as defined in Lemma 3.6), then it is finite. We use the bisimulation just proved to extend the result to HORSC.

The definitions of well-foundedness and length of paths in the state tree are unchanged by the use of HORSC. Take a possibly-infinite path of computation $\tau' = J_1 J_2 \cdots$ that induces no infinite path in the state tree. We wish to show that τ' must be finite. By Lemma 4.12 we have a witnessing well-founded reduction sequence $\tau_H : S = u_1 \twoheadrightarrow u_2 \twoheadrightarrow \cdots$ and from the bisimulation above a corresponding reduction sequence $\tau_G : S = h(u_1) \twoheadrightarrow h(u_2) \twoheadrightarrow \cdots$.

It suffices to show:

(i) If τ_H is well-founded then τ_G is well-founded.

(ii) If τ_G is finite then τ_H is finite.

For (i), we assume that the reduction sequence τ_H is well-founded so that there is no infinite sequence of the form $S = F_1 > F_{i_1}^{\ell_1} > F_{i_2}^{\ell_2} > \cdots$. From the bisimulation above, τ_G may not contain such a sequence if the F_{i_j} are members of $\mathcal{N}$, but it may contain one using the new non-terminals (the B_i). However, the B_i have no non-terminals on their right-hand sides and so a term headed by any B_i cannot

be related to any later term in the reduction sequence by $>$. For (ii), the result is immediate from the bisimulation, as τ_G and τ_H must be the same length.

Then, as we assumed that τ' did not induce any infinite path in the state tree, τ_H must be well-founded. From (i), Kobayashi and Ong's result, and (ii), we obtain that τ_H is finite. A final application of Lemma 4.12 gives τ' finite as required. $\square$

Appendix B

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:

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

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:

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

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:

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

Applying F to the third term with the hole in the first argument $F(a \Box y)$ yields the set $\{ z,\ s\ z,\ s\ (s\ z) \}$ (because terms that are instances of $a \Box y$ are also instances of $a \Box z$ and $a \Box (s\ z)$). 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 B.1. *Let* I *be a base and* $\xi\ p_1 \cdots p_{j-1} \Box p_{j+1} \cdots p_m$ *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* $q \in F(\xi\ p_1 \cdots p_{j-1} \Box p_{j+1} \cdots p_m)$ *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 $t' \coloneqq \xi\ \sigma p_1 \cdots \sigma p_{j-1}\ t\ \sigma p_{j+1} \cdots \sigma p_m \in T(\Sigma)$. 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, $q_j \in F(\xi\ p_1 \cdots p_{j-1} \Box p_{j+1}\ p_m)$. □

Lemma B.2. *Let* I *be a base and* $\xi\ p_1 \cdots p_{j-1} \Box p_{j+1} \cdots p_m$ *be some term in* I *with one type-*b *argument replaced by a hole. For all* $p,\ q \in F(\xi\ p_1 \cdots p_{j-1} \Box p_{j+1} \cdots p_m)$*, if* p *and* q *unify, then* $p = q$*.*

Proof. Let $q,\ q' \in F(\xi\ p_1 \cdots p_{j-1} \Box p_{j+1} \cdots p_m)$ 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 $\hat{\theta} \coloneqq \theta \cup \theta'$. Similarly, since the elements of a base are linear, we can consider the substitution:

$$\varepsilon(x) \coloneqq \begin{cases} \beta(x) & \text{if } x \in FV(p_l) \\ \alpha(x) & \text{otherwise} \end{cases}$$

Consider next, the term:

$$t \coloneqq \xi\, \hat{\theta} q_1 \cdots \hat{\theta} q_{l-1}\, \hat{\theta} q'_l\, \hat{\theta} q_{l+1} \cdots \hat{\theta} q_{j-1}\, q\, \hat{\theta} q_{j+1} \cdots \hat{\theta} q_m$$

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 $i \in ([1..m] \setminus \{l, j\})$, $\gamma(\varepsilon p_i) = \gamma(\alpha p_i) = \gamma(\theta q_i) = \gamma(\hat{\theta} q_i) = \delta r_i$ and $\gamma(\varepsilon p_l) = \gamma(\beta p_l) = \gamma(\theta' q_l) = \gamma(\hat{\theta} q_l) = \delta r_l$. Hence $\xi\, r_1 \cdots r_m$ □

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 B.1

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

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

Table B.1: Definition of $\mathcal{N}^{\#}$ and $\mathcal{R}^{\#}$.

Bibliography

J. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 459–472. ACM, 2011. ISBN 978-1-4503-0490-0. (Cited on pages 5 and 25.)

G. Arroyo, J. G. Ramos, S. Tamarit, and G. Vidal. A transformational approach to polyvariant bta of higher-order functional programs. In M. Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers*, volume 5438 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2008. ISBN 978-3-642-00514-5. (Cited on page 11.)

A. Bakewell and D. R. Ghica. Game-based safety checking with mage. In A. Poetzsch-Heffter, editor, *SAVCBS*, pages 85–87. ACM, 2007. ISBN 978-1-59593-721-6. (Cited on page 5.)

I. Balaban, A. Pnueli, and L. D. Zuck. Modular ranking abstraction. *International Journal of Foundations of Computer Science*, 18(01):5–44, Feb 2007. ISSN 1793-6373. doi: 10.1142/s0129054107004553. (Cited on page 118.)

T. Ball and S. Rajamani. Bebop: a symbolic model checker for Boolean programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *Lecture Notes in Computer Science*, pages 113 – 130. Springer, 2000a. ISBN 3-540-41030-9. (Cited on page 3.)

T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000b. (Cited on page 8.)

T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3. ACM, 2002. ISBN 1-58113-450-9. (Cited on pages 3 and 33.)

T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside Microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004. ISBN 3-540-21377-5. (Cited on page 3.)

H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983. (Cited on pages 25 and 88.)

J. G. P. Barnes. *High Integrity Software - The SPARK Approach to Safety and Security*. Addison-Wesley, 2003. ISBN 978-0-321-13616-9. (Cited on page 2.)

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24287-1. doi: 10.1007/978-3-540-30569-9_3. (Cited on page 2.)

I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In D. L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-58179-6. doi: 10.1007/3-540-58179-0_53. (Cited on page 3.)

V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In C. Runciman and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 51–63. ACM, 2003. ISBN 1-58113-756-7. (Cited on page 25.)

B. Boehm, H. D. Rombach, and M. V. Zelkowitz. *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili*, volume 1. Springer, 2005. ISBN 9783540245476. (Cited on page 1.)

A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. ISBN 3-540-63141-0. (Cited on page 3.)

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. (Cited on page 6.)

J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 46–51, 1990. (Cited on page 3.)

M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978. (Cited on page 25.)

B. Courcelle and M. Nivat. The algebraic semantics of recursive program schemes. In J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4-8, 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1978. (Cited on page 6.)

CVE-2014-0092. lib/x509/verify.c in GnuTLS before 3.1.22 and 3.2.x before 3.2.12 does not properly handle unspecified errors when verifying X.509 certificates from SSL servers, which allows man-in-the-middle attackers to spoof servers via a crafted certificate. National Vulnerability Database, March 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0092`. (Cited on page 1.)

CVE-2014-1266. SSLVerifySignedServerKeyExchange vulnerability. National Vulnerability Database, March 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266`. (Cited on page 1.)

L. Damas and R. Milner. Principal type-schemes for functional programs. In R. A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982. ISBN 0-89791-065-6. (Cited on page 4.)

W. Damm. The IO- and OI-hierarchy. *Theoretical Computer Science*, 20:95–207, 1982. (Cited on pages 7 and 17.)

O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. (Cited on page 32.)

W.-P. de Roever and J. W. de Bakker. A calculus for recursive program schemes. In M. Nivat, editor, *Proc. IRIA symposium on Automata, Languages and Programming*. North Holland, 1972. (Cited on page 6.)

D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005. (Cited on page 2.)

E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10): 859–866, October 1972. ISSN 0001-0782. doi: 10.1145/355604.361591. (Cited on page 2.)

E. W. Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In F. L. Bauer and K. Samelson, editors, *Language Hierarchies and Interfaces, International Summer School, Marktoberdorf, Germany, July 23 - August 2, 1975*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1975. ISBN 3-540-07994-7. (Cited on page 2.)

M. Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84–, March 1997. ISSN 0163-5948. doi: 10.1145/251880.251992. (Cited on page 1.)

ECMA International. *Standard ECMA-334 - C# Language Specification*. Ecma International, 4 edition, June 2006. URL `http://www.ecma-international.org/publications/standards/Ecma-334.htm`. (Cited on page 4.)

E. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin / Heidelberg, 1980. (Cited on page 3.)

E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 368–377, 1991. (Cited on page 19.)

J. Engelfriet and E. M. Schmidt. IO and OI. I. *Journal of Computer and System Sciences*, 15(3):328–353, 1977. (Cited on page 16.)

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In J. Knoop and L. J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM, 2002. ISBN 1-58113-463-0. (Cited on page 2.)

R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967. (Cited on page 2.)

K. Fujima, S. Ito, and N. Kobayashi. Practical alternating parity tree automata model checking of higher-order recursion schemes. In C. chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2013. ISBN 978-3-319-03541-3. (Cited on pages 9 and 110.)

D. R. Ghica and A. Bakewell. Clipping: A semantics-directed syntactic approximation. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 189–198. IEEE Computer Society, 2009. ISBN 978-0-7695-3746-7. (Cited on page 5.)

G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. ISBN 978-3-540-87826-1. (Cited on page 6.)

E. Grädel, W. Thomas, and T. Wilke. *Auotmata, Logics, and Infinite Games*. LNCS 2500. Springer-Verlag, 2002. (Cited on page 23.)

A. Haddad. IO vs OI in higher-order recursion schemes. In D. Miller and Z. Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*, volume 77 of *EPTCS*, pages 23–30, 2012. (Cited on page 18.)

M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2008. (Cited on page 7.)

A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002. (Cited on page 1.)

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on page 2.)

G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. (Cited on page 3.)

D. Hopkins and C.-H. L. Ong. Homer: A higher-order observational equivalence model checker. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture*

Notes in Computer Science, pages 654–660. Springer, 2009. ISBN 978-3-642-02657-7. (Cited on page 5.)

D. Hopkins, A. S. Murawski, and C.-H. L. Ong. Hector: An equivalence checker for a higher-order fragment of ML. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 774–780. Springer, 2012. ISBN 978-3-642-31423-0. (Cited on page 5.)

J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000. (Cited on page 5.)

A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005. (Cited on pages 7 and 31.)

N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375:120–136, 2007. (Cited on pages 5, 33 and 117.)

G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2, 2014. (Cited on page 6.)

T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002. ISBN 3-540-43366-X. (Cited on page 7.)

N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428. ACM, 2009a. ISBN 978-1-60558-379-2. (Cited on pages 7, 8, 25 and 35.)

N. Kobayashi. Model-checking higher-order functions. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 25–36. ACM, 2009b. ISBN 978-1-60558-568-0. (Cited on pages 8, 25 and 30.)

N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In M. Hofmann, editor, *Foundations of Software*

Science and Computational Structures - 14th International Conference, FoSSaCS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, volume 6604 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2011. ISBN 978-3-642-19804-5. (Cited on pages 8, 31 and 90.)

N. Kobayashi. Model checking higher-order programs. *Journal of the ACM*, 60(3): 20, 2013. (Cited on page 28.)

N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 179–188. IEEE Computer Society, 2009. ISBN 978-0-7695-3746-7. (Cited on pages 8, 25, 30, 35, 51, 52, 95 and 120.)

N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7 (4), 2011. (Cited on page 29.)

N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 495–508. ACM, 2010. ISBN 978-1-60558-479-9. (Cited on pages 32 and 92.)

N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233. ACM, 2011. ISBN 978-1-4503-0663-8. (Cited on pages 33 and 118.)

G. M. Kobele and S. Salvati. The IO and OI hierarchies revisited. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 336–348. Springer, 2013. ISBN 978-3-642-39211-5. (Cited on page 18.)

D. Kozen. Results on propositional mu-calculus. *Theoretical Computer Science*, 2: 333–354, 1983. (Cited on pages 3 and 19.)

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, pages 179–192. ACM, 2014. ISBN 978-1-4503-2544-8. (Cited on page 6.)

O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000. (Cited on page 20.)

T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In Z. Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 392–411. Springer, 2014. ISBN 978-3-642-54832-1. (Cited on page 118.)

C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM Press, 2001. (Cited on page 5.)

X. Leroy. Some uses of Caml in industry. Commercial Users of Functional Programming Workshop, 2007. URL `http://cufp.org/archive/2007/slides/XavierLeroy.pdf`. (Cited on page 4.)

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52 (7):107–115, 2009. (Cited on page 6.)

A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In R. Parikh, editor, *Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer, 1985. ISBN 3-540-15648-8. (Cited on page 68.)

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978. (Cited on pages 3, 4 and 28.)

Y. Minksy and S. Weeks. Caml trading – experiences with functional programming on Wall Street. *Journal of Functional Programming*, 18:553–564, 7 2008. ISSN 1469-7653. doi: 10.1017/S095679680800676X. (Cited on page 4.)

N. Mitchell and C. 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. ISBN 978-1-60558-064-7. doi: http://doi.acm.org/10.1145/1411286.1411293. (Cited on pages 64 and 91.)

D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987. (Cited on page 23.)

D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of trees and its complexity. *Theoretical Computer Science*, 97(2):233–244, 1992. (Cited on page 21.)

R. P. Neatherway and C.-H. L. Ong. TravMC2: higher-order model checking for alternating parity tree automata. In N. Rungta and O. Tkachuk, editors, *SPIN*, pages 129–132. ACM, 2014. ISBN 978-1-4503-2452-6. (Cited on page 93.)

R. P. Neatherway, C.-H. L. Ong, and S. J. Ramsay. A traversal-based algorithm for higher-order model checking. In P. Thiemann and R. B. Findler, editors, *International Conference on Functional Programming*, pages 353–364. ACM, 2012. ISBN 978-1-4503-1054-3. (Cited on pages 35 and 63.)

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. (Cited on page 6.)

U. Norell. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481862. (Cited on page 6.)

C.-H. L. Ong. Observational equivalence of third-order Idealized Algol is decidable. In *Proceedings of IEEE Symposium on Logic in Computer Science, 22-25 July 2002, Copenhagen Denmark*, pages 245–256. Computer Society Press, 2002. (Cited on page 5.)

C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 81–90, Los Alamitos, CA, USA, 2006. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.2006.38. (Cited on pages 7, 8, 9, 24, 35, 36, 52 and 57.)

C.-H. L. Ong and S. J. Ramsay. Verifying functional programs with pattern matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 587–598, 2011. (Cited on pages 9, 33, 63, 91, 115 and 118.)

S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. H. Reppy and J. L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61. ACM, 2006. ISBN 1-59593-309-3. (Cited on page 4.)

S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal Functional Programming*, 17(1):1–82, 2007. (Cited on page 5.)

S. L. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`. (Cited on page 4.)

G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975. (Cited on page 32.)

J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin / Heidelberg, 1982. (Cited on page 3.)

S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 61–72. ACM, 2014. ISBN 978-1-4503-2544-8. (Cited on page 117.)

M. Reinhold, K. Bourrillion, S. Poole, and A. Haley. JSR 337: JavaTM SE 8 Release Contents. `https://jcp.org/en/jsr/detail?id=337`, 2014. Accessed: 2014-03-06. (Cited on page 4.)

J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. (Cited on page 11.)

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008. (Cited on page 5.)

W. C. Rounds. Context-free grammars on trees. In P. C. Fischer, S. Ginsburg, and M. A. Harrison, editors, *Proceedings of the 1st Annual ACM Symposium on Theory of Computing, May 5-7, 1969, Marina del Rey, CA, USA*, pages 143–148. ACM, 1969. (Cited on page 16.)

RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011. (Cited on page 1.)

S. Salvati and I. Walukiewicz. Krivine machines and higher-order schemes. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 2011. ISBN 978-3-642-22011-1. (Cited on page 12.)

R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In E. Albert and S.-C. Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, pages 53–62. ACM, 2013. ISBN 978-1-4503-1842-6. (Cited on page 33.)

D. Sereni. Termination analysis and call graph construction for higher-order functional programs. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 71–84. ACM, 2007. ISBN 978-1-59593-815-2. (Cited on page 5.)

O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnige-Mellon Univeristy, May 1991. (Cited on pages 5, 33 and 117.)

K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. ISBN 978-3-540-71065-3. (Cited on page 6.)

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *Journal of Functional Programming*, 23(4):402–451, 2013. (Cited on page 6.)

G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002. (Cited on pages 1 and 2.)

T. Terauchi. Dependent types from counterexamples. In *Principles of Programming Languages*, pages 119–130, 2010. (Cited on page 6.)

T. Tsukada and N. Kobayashi. Untyped recursion schemes and infinite intersection types. In C.-H. L. Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus,*

March 20-28, 2010. Proceedings, volume 6014 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2010. ISBN 978-3-642-12031-2. (Cited on page 12.)

T. Tsukada and N. Kobayashi. Complexity of model-checking call-by-value programs. In A. Muscholl, editor, *FoSSaCS*, volume 8412 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2014. ISBN 978-3-642-54829-1. (Cited on pages 18 and 116.)

H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 75–86. ACM, 2013. ISBN 978-1-4503-1832-7. (Cited on page 33.)

M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of IEEE Annual Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1986. (Cited on page 3.)

Various. DO-333 formal methods supplement to do-178c and do-278a. Technical report, RTCA & EUROCAE, December 2011. (Cited on page 2.)

N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013. ISBN 978-3-642-37035-9. (Cited on page 5.)

D. Vytiniotis, S. L. Peyton Jones, K. Claessen, and D. Rosén. Halo: haskell to logic through denotational semantics. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 431–442. ACM, 2013. ISBN 978-1-4503-1832-7. (Cited on page 6.)

S. Weirich, J. Hsu, and R. A. Eisenberg. System fc with explicit kind equality. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 275–286. ACM, 2013. ISBN 978-1-4503-2326-0. (Cited on page 118.)

D. N. Xu, S. L. Peyton Jones, and K. Claessen. Static contract checking for haskell. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 41–52. ACM, 2009. ISBN 978-1-60558-379-2. (Cited on page 6.)

P. C. Yves Bertot. *Interactive Theorem Proving and Program Development: Coq'Art : the Calculus of Inductive Constructions*. Springer-Verlag, 2004. (Cited on page 2.)

Index of Definitions

Index of Notations