Intersection Types and Higher-Order Model Checking

Steven J. Ramsay
Merton College
University of Oxford

A dissertation submitted for the degree of
Doctor of Philosophy in Computer Science
Trinity Term 2013

Abstract

Higher-order recursion schemes are systems of equations that are used to define finite and infinite labelled trees. Since, as Ong has shown, the trees defined have a decidable monadic second order theory, recursion schemes have drawn the attention of research in program verification, where they sit naturally as a higher-order, functional analogue of Boolean programs. Driven by applications, fragments have been studied, algorithms developed and extensions proposed; the emerging theme is called *higher-order model checking*. Kobayashi has pioneered an approach to higher-order model checking using intersection types, from which many recent advances have followed. The key is a characterisation of model checking as a problem of intersection type assignment. This dissertation contributes to both the theory and practice of the intersection type approach.

A new, fixed-parameter polynomial-time decision procedure is described for the alternating trivial automaton fragment of higher-order model checking. The algorithm uses a novel, type-directed form of abstraction refinement, in which behaviours of the scheme are distinguished according to the intersection types that they inhabit. Furthermore, by using types to reason about acceptance and rejection simultaneously, the algorithm is able to converge on a solution from two sides. An implementation, Preface, and an extensive body of evidence demonstrate empirically that the algorithm scales well to schemes of several thousand rules. A comparison with other tools on benchmarks derived from current practice and the related literature puts it well beyond the state-of-the-art.

A generalisation of the intersection type approach is presented in which higher-order model checking is seen as an instance of exact abstract interpretation. Intersection type assignment is used to characterise a general class of safety checking problems, defined independently of any particular representation (such as automata) for a class of recursion schemes built over arbitrary constants. Decidability of any problem in the class is an immediate corollary. Moreover, the work looks beyond whole-program verification, the traditional territory of model checking, by giving a natural treatment of higher-type properties, which are sets of functions.

Acknowledgements

I am very grateful to my supervisor, Luke Ong, for the patient guidance and constant encouragement that he has given to me during the course of the research contained in this dissertation. My thanks are also due to Martin Hofmann and Hongseok Yang, who gave much of their time to ensure its proper assessment. Finally, I gratefully acknowledge the help of the *Engineering and Physical Sciences Research Council*, whose financial support has been essential.

Contents

Contents **vii**

1 Introduction **1**
1.1 The difficulties of constructing correct software . . . 1
1.2 Verification and software model checking . . . 2
1.3 Functional programming . . . 3
1.4 Higher-order model checking . . . 5
1.5 Contributions and structure . . . 6

2 Higher-Order Model Checking **9**
2.1 Higher-order recursion schemes . . . 9
2.2 Model checking problems for recursion schemes . . . 23
2.3 The intersection type characterisation . . . 27
2.4 Algorithms for higher-order model checking . . . 33
2.5 Applications in verification . . . 39

3 An Intersection Refinement Type System with Subtyping **45**
3.1 Intersection types . . . 45
3.2 Intersection type assignment . . . 49
3.3 Intersection type checking . . . 54
3.4 Consistency of type environments . . . 56
3.5 Higher-order model checking is type inference . . . 60

4 Model Checking via Type Directed Abstraction Refinement **63**
4.1 Introduction . . . 63
4.2 Type directed abstraction refinement . . . 65
4.3 A decision procedure for model checking . . . 70
4.4 A narrated example . . . 80
4.5 Correctness of the decision procedure . . . 84
4.6 PREFACE: a higher-order model checker . . . 97

4.7 Related work . . . 102

5 Intersection Types as Exact Abstract Interpretations **107**
5.1 Introduction . . . 107
5.2 Term languages, property languages and queries . . . 109
5.3 Concrete and abstract properties . . . 118
5.4 Exact abstraction at ground types . . . 122
5.5 Applications to higher-order model checking . . . 129
5.6 Exact abstraction at higher types . . . 134
5.7 Related work . . . 138

6 Conclusion **141**
6.1 Summary . . . 141
6.2 Discussion . . . 142
6.3 Future directions . . . 144

A Evaluation: Complete Results **147**

Bibliography **157**

Index of Definitions **163**

Index of Notations **165**

Chapter 1

Introduction

1.1 The difficulties of constructing correct software

> "Because, in a sense, the whole is 'bigger' than its parts, the depth of a hierarchical decomposition is some sort of logarithm of the ratio of the 'sizes' of the whole and the ultimate smallest parts. From a bit to a few hundred megabytes, from a microsecond to a half an hour of computing confronts us with completely baffling ratio of 10^9! The programmer is in the unique position that his is the only discipline and profession in which such a gigantic ratio, which totally baffles our imagination, has to be bridged by a single technology."
>
> E. W. Dijkstra

Software is complex. Of course, the size of modern computer programs, which may be documents tens of millions of lines long, is a major factor in complexity. That such software, whose complete comprehension is well beyond the faculties of any one person, works even some of the time is a great achievement of the methods of modern software engineering; that it fails the rest of the time is a great frustration of its users. However, the sheer scale of the objects that are involved is only a small part of the problem. There is also the great depth of the hierarchies involved, as pointed out in the excerpt above by Dijkstra, one of the great pioneers of the science of computer programming. Such hierarchies are inevitable when thinking about how to instruct a machine, whose basic operations can be applied to (perhaps) 64 bits at a time, in how to perform complex processing tasks that may ultimately involve many gigabytes of data. Still other factors are social and related to the education of computer programmers, the expectations of their clients and the lack of generally accepted standards.

The consequence of software complexity, and our apparent inability to manage it effectively, is an unacceptable rate of defects. At the time of writing, it would be very surprising to find any serious user of computing equipment who had not suffered frustration with the inadequacies of faulty computer programs. That this statement is hardly controversial is already a serious indictment of the state of affairs, but such is the pervasiveness of computing in industry and in business that faulty software also carries with

it a tangible financial cost. In a 2002 report [RTI, 2002], the National Institute of Standards and Technology (NIST) estimated the cost to the United States national economy of failing to identify defects in computer software as $59.5 billion, annually. To put the figure into context, the economic cost of Hurricane Sandy which, among other disastrous effects, caused the cancellation of almost 20,000 airline flights, left millions without power, destroyed thousands of homes, and completely closed the New York Stock Exchange for two days, was recently estimated at $65 billion [see US NOAA 2013].

Conventionally, the main weapon with which to combat software defects is testing. Testing is the work of running a given program on various inputs and observing the corresponding outputs. In many respects, this follows the same well-established pattern found in engineering more generally: once a component of some design is built, it is tested in order to check that it performs correctly within some tolerances and, if this is the case, then it is accepted. However, software is unlike the familiar materials of other engineering disciplines. Its behaviour does not conform to well known physical principles; instead the laws which it obeys are a function of the code that the programmer has written. In effect, each piece of software is a new material, whose attributes must be divined by experimentation. Even two pieces of software written to the same specification and in the same design are unlikely to share all the same attributes, since each will almost certainly have its own unique set of faults which serve to distort the original intention in unpredictable ways. Consequently, to be effective, software testing must be extremely thorough, and hence expensive. A study undertaken by Maximilien and Williams [2003] at IBM found that, by adopting a test-driven development methodology, they were able to reduce the density of defects discovered from 7 per thousand lines of code to (only) 3.7 per thousand. However, this involved writing 34,000 lines of tests with which to assess their 71,400 line program; a ratio which is not uncommon in projects where quality assurance is taken seriously.

1.2 Verification and software model checking

Since thorough testing is often prohibitively expensive and yields relatively weak assurances that the most serious defects have been identified, the problem of checking the correctness of software is a very large and intensively researched area of computer science. This area, *software verification*, is characterised by the application of mathematical techniques in order to *prove* that given programs have certain properties, such as freedom from particular classes of faults.

Traditionally, work on verification in computer science has been divided into two camps. On the one hand, many elegant *conceptual* methods have been devised for proving, *by hand*, that given programs satisfy almost arbitrarily complicated properties. These include high-level logics specialized to particular types of property, and theories of programming that formalise algorithms, data structures or even whole programming paradigms. On the other, very powerful *mechanical* procedures have been developed to prove, *automatically*, that given programs satisfy more restricted classes of properties. These include general methods of abstraction and approximation as well as highly efficient decision pro-

cedures. However, especially in the past decade, the boundaries between the two have blurred. Now, in the former camp, computer automation has become an increasingly indispensable assistant to the human theorem prover, and in the latter opportunities for human interaction have been exploited as automation has become more tightly integrated into the work-flow of software engineers. Nevertheless, in this dissertation we shall be exclusively concerned with a fully automatic technique: model checking, which has been shown to be a particularly fruitful perspective on the algorithmic analysis of computer programs.

Model checking encompasses a very old family of problems that originated in mathematical logic. For a given logic $\mathcal{L}$, the $\mathcal{L}$-model checking problem is usually phrased as follows: given an $\mathcal{L}$-sentence ϕ and an $\mathcal{L}$-structure M, is M a model of ϕ? Of course, there are many variations, but the essential aspect is that both the structure M and the property ϕ are given in the input and the problem is to determine whether the latter is satisfied by the former. The problem has become an important one in computer science for two reasons. First, the setting is convenient because the essential features of the execution of a program can be usefully described by a relational structure M and interesting correctness properties ϕ can be formulated in the language of a logic $\mathcal{L}$ over such structures. Second, it is often the case that M can be presented finitely and then, for sufficiently constrained logics, the model checking problem may be decidable.

The adoption of model checking by computer science began in the early eighties with the work of Clarke and Emerson [1981] and, independently, Queille and Sifakis [1982]. As a measure of the success of the approach, in 2007, Clarke, Emerson and Sifakis were jointly honoured by the Association of Computing Machinery (ACM) with their highly prestigious A. M. Turing Award:

> "...for their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries"

Indeed, since this early work, which was largely focused on checking properties of hardware systems, model checking has been successfully adapted to an impressive array of verification tasks. In one direction, by developing new logics, it is now possible to specify much richer classes of properties. This has included not only generalisations of the logic used by Clarke and Emerson, but also quantitative and probabilistic logics whose truth is not Boolean. In another direction, by studying classes of finitely presentable structures, new computational situations could be modelled. For example, fifteen years after the initial work, Bouajjani, Esparza, and Maler [1997] presented the first decision procedures for pushdown model checking, which concerns itself with the satisfiability of formulae over the configuration graphs of pushdown automata.

1.3 Functional programming

An alternative line of attack on the problem of constructing correct software is to improve the languages in which software is written. The idea is that programmers can be forced to write code with fewer bugs if they program in languages which either simply do not

have the power to express programs containing certain classes of error, or discourage the programmer from doing so. A very old paradigm in programming languages, *functional programming*, has seen an incredible resurgence in popularity in the last decade for exactly this reason. Where functional programming was once the preserve of academia, often considered too esoteric for industrial use, it is now gaining much more acceptance in the mainstream. Languages, such as Microsoft's F#, are being actively developed to meet a growing demand in the financial and scientific sectors; both are areas where a premium is put on defect-free software. Functional programming offers some hope to engineers who want to construct correct software, since the basic unit of the program is the function. Since functions always yield the same outputs for the same inputs, they are an inherently simpler entity to understand and reason about than the procedures and statements that form the atoms of mainstream imperative programming.

Although there is no generally accepted definition of what constitutes a functional programming language, it is commonly regarded that languages such as Standard ML, OCaml, Haskell and F# are certainly functional. One criterion that is rarely disputed is the requirement that functional languages should have *first class* support for functions. To say that functions have this status is another way of saying that anywhere the programmer might describe some computation on an integer or a string of text or on some other data type, she may just as easily describe some computation on a function. In other words, functions are first class citizens, in the sense of being welcome anywhere that any other data type might be welcome. Hence, such programming languages support *higher-order functions* (since functions may operate on other functions supplied to them as input, or return them in the form of output), and it is common to describe the programs written in these languages as being "higher-order". It is with this feature of functional programming that we shall be especially interested in this dissertation.

To illustrate the distinction more concretely, consider the function that squares a number given to it as input. When given any number n as an input, *square* gives back as output the result of multiplying n by itself. So if we feed to *square* the input 2, then the output will be the number 4. This is a first-order function, since both its inputs and outputs are numbers, and numbers are not functions. By contrast, the function *twice* is the function that takes as input a first-order function f and a number n and gives back as output the result of feeding n as input to f, then feeding the corresponding output $f(n)$ back to f as a new input and finally returning the corresponding output $f(f(n))$ as the overall output. In other words, *twice* applies f to n, twice consecutively. So if we feed *twice* the inputs *square* and 2 then the output will be the number 8 because that is the square of the square of 2. The function *twice* is a second-order function, because one of its inputs is itself a first-order function.

In first-order programs, there is a clean separation between functions (which are necessarily first order) on the one hand and data on the other. Functions perform computation and that computation is on data, but the data itself has no ability to compute. In higher-order programs functions still perform computations, but computation can be on functions (as well as other data). A popular slogan for functional programming asserts that "functions *are* data". This blurring of the distinction between control and data presents a unique challenge to automated program verification generally and model checking in particular,

and it is this fact that gives *higher-order model checking* its unique flavour.

1.4 Higher-order model checking

At the turn of the century, model checking techniques had been developed for a wide range of computational situations including, as we mentioned previously, the verification of properties of the configuration graphs of pushdown automata. Whilst pushdown automata can accurately model systems of first-order procedures over finite data domains, and hence finite data models of first-order, imperative programs, they are lacking as models of higher-order programs. It was not until 2002 that Knapik, Niwinski, and Urzyczyn [2002] showed the decidability of the monadic second-order (MSO) theory of the infinite trees generated by higher-order pushdown automata. However, although very natural analogues of their first-order counterparts, these structures do not completely capture systems of higher-order, recursive procedures in the same way that pushdown automata capture first-order systems[1]. Hence, the situation was only partly resolved. It was finally settled in 2006, with Ong's landmark paper demonstrating MSO decidability of the trees generated by *higher-order recursion schemes* [Ong, 2006], and thus opening the door to higher-order model checking.

Higher-order recursion schemes are systems of equations for defining finite and infinite trees. They are an old formalism which has been reinvigorated by the modern interest in finitely presentable infinite structures for model checking. Recursion schemes can be viewed as a kind of restricted class of functional programs. The programs that they define are built over constructors for data types using higher-order functions and general recursion but lacking in any ability to inspect or de-construct the data with which they compute. It is this restriction that makes them interesting as abstract models of higher-order computation.

The application of recursion scheme model checking to problems concerning the verification of functional programs was picked up in earnest by Kobayashi in a paper of 2009 [Kobayashi, 2009a], which represented two important contributions to the field. First, Kobayashi sketched a methodology by which a recursion scheme could be used in order to present (an abstract description of) the computation tree of a functional program — the history of all the actions that the program takes on any given input — thus making this object amenable to property checking. Second, he gave a characterisation of a model checking problem for recursion schemes by means of a type assignment problem in an intersection type system.

The characterisation is as follows: the tree defined by a recursion scheme satisfies a given property if, and only if, the same scheme can be assigned a type in an intersection type system induced by the property. By casting model checking as a problem of type assignment, it is made amenable to techniques and intuitions developed in the programming languages community, which brings an appealing new perspective to the problem. The

[1] In fact, higher-order pushdown automata capture the same class of trees as *safe* recursion schemes, which are higher-order recursive procedures satisfying a particular syntactic constraint. It is now known that this constraint is a genuine limitation to expressivity [Parys, 2012].

approach has been very successful, with many advances in higher-order model checking following from the intersection type characterisation.

Since this initial work of Ong and Kobayashi, the promise of sophisticated program analyses based on recursion scheme model checking have driven a rapidly developing field. Several such program analyses have been proposed and range from traditional predicate abstraction-based reachability checking to the verification of pattern-matching safety and highly accurate control flow analyses. In turn, these applications require efficient decision procedures for the underlying model checking problem and several have been developed, implemented and evaluated in quick succession and with very promising results. Furthermore, extensions that try to equip recursion schemes with the ability to more naturally model specific computational features, and alternative characterisations, such as through higher-order collapsible pushdown automata, have been studied and fragments of the model checking problem classified. In this dissertation, we present three further contributions to both the theory and practice of higher-order model checking.

1.5 Contributions and structure

In this dissertation we present a new intersection type system with a restricted form of subtyping and a simple and efficient type checking procedure. Following the pioneering work of Kobayashi [2009a] and Kobayashi and Ong [2009] on intersection type systems without subtype theories, type assignment in this system is shown to characterise the (co-)trivial automaton model checking problem for higher-order recursion schemes. We then present two applications of this system which help to advance the theory and practice of higher-order model checking:

- We describe a new, fixed-parameter polynomial-time decision procedure for the (co-)trivial automaton model checking problem for recursion schemes. The algorithm solves the corresponding intersection type assignment problem. It is based on a novel, type directed form of abstraction refinement in which behaviours of a scheme are distinguished by the abstraction according to the intersection types that they inhabit (the properties that they satisfy). Unlike other intersection type approaches, our algorithm reasons both about acceptance by the property automaton and acceptance by its dual, simultaneously, in order to minimize the amount of work done by converging on the solution to a problem instance from both sides. We present an implementation of the algorithm in a tool, Preface, and an extensive body of evidence to demonstrate empirically that the algorithm scales well to schemes of several thousand rules. When evaluated on the current set of benchmarks derived from the related literature, Preface is shown to outperform other state-of-the-art model checkers convincingly.

- We describe a new, semantic framework for property checking of recursion schemes which subsumes the alternating trivial automaton model checking problem. The framework presents intersection type assignment as an exact abstract interpretation of a denotational semantics of recursion schemes. The framework allows for proofs

of the soundness and completeness of intersection type characterisations of safety property checking for recursion schemes which are local, in the sense of depending only on a condition concerning the constants over which the scheme is built. Decidability of the safety property checking problem for the given class of schemes is an immediate corollary. Using the framework, we reconstruct the interesting features of variants of recursion scheme from the literature and give short and simple proofs of the decidability of the corresponding property checking problems. Finally, we show the versatility of the framework by looking beyond traditional model checking problems, which check properties of ground type objects like words and trees, to characterisations of problems involving properties of higher-order functions.

In more detail, the remaining parts of the dissertation are structured as follows:

- In Chapter 2 we survey the definitions, results and applications from higher-order model checking, with which this dissertation is primarily concerned. This chapter formally defines the key notions of higher-order recursion schemes and alternating (co-)trivial automata, a pair of which form an instance of the (co-)tivial automaton model checking problem for recursion schemes. We then survey the intersection type approach to this problem, thus explaining our interest in the two seemingly disparate entities in the title. We next take some time to introduce the various decision procedures for the model checking problem that have been presented in the literature, thus setting the scene for our own contribution in Chapter 4. Finally, we survey applications of the model checking problem in verification so as to give the reader a better idea of how a study of the problem is useful in practice.

- In Chapter 3 we introduce an extension of Kobayashi's intersection type system which we surveyed in Chapter 2. Our system admits a parametrised notion of subtyping between intersection types. Whilst subtyping is a well studied feature of intersection type systems, such systems are not usually concerned with the efficiency of type checking algorithms, which is essential here due to the intended applications. We describe a novel system that has a relatively limited form of subtyping but a correspondingly simple and efficient type checking procedure. Although many of the facts that we present in this chapter are straightforward adaptations of those for related systems in the literature, we take the opportunity to present a result which sheds some light on the relationship between the notion of co-consistent type environment and recursion scheme reduction. Using this result we are able to show quite directly that, despite the fact that the system admits a greater number of solutions to type assignment problems, it still characterises recursion scheme model checking.

- We put the type system to use as part of a decision procedure for higher-order model checking in Chapter 4. The justification of the algorithm, which computes successive approximations to the required type assignment, depends upon the extra freedom afforded by subtyping. We first describe the algorithm informally through a worked example. The intention is to give the reader some intuitions about the key ideas. We

then give a precise description in terms of a number of inductive and coinductive constructions. The algorithm is shown to solve the (co-)trivial automaton model checking problem for recursion schemes and to run, in the worst case, in time that is bounded by a polynomial function of the size of the scheme (assuming all other relevant parameters are fixed). The overall shape of the proof of the correctness of the algorithm is sketched as part of the description, but all the proofs are relegated to the following section. The reader is then guided through an example of the algorithm in action, which will hopefully make any remaining details clear. Finally we discuss our implementation of the algorithm in a tool, PREFACE, and present a digest and analysis of its empirical evaluation.

- In Chapter 5 we present a new, semantic framework for safety property checking of higher-order recursion schemes which subsumes the trivial automaton model checking problem. At the core of the presentation is the notion of exact abstract interpretation. We start by giving a quite standard interpretation of recursion schemes, parametrised over a set of basic constants, within the discipline of elementary domain theory. We follow this by showing how one may interpret intersection refinement types as safety properties, which are certain closed subsets of domains. One can put safety properties and intersection types into a Galois correspondence, and then by phrasing property checking as an inclusion of safety properties, the setting is brought into the sphere of abstract interpretation. We show that, whenever the intersection types assigned to the basic constants are an exact abstraction of those constants, then the strongest intersection type assignable to a given closed, ground type term is exactly the best abstraction of the term according to the Galois connection. Consequently, ground type property checking is decidable. The framework is applied to reconstruct, in a short and natural way, the decidability of various extensions of the higher-order model checking problem from the literature. Finally, the same framework is shown to extend to property checking of higher-types, thus moving properly beyond higher-order model checking.

- We conclude in Chapter 6 by summarising the main results of the thesis and discussing their strengths and weaknesses. We look forward to future work and consider some of the more interesting directions arising from the material presented here.

We survey the literature related to higher-order model checking in Chapter 2, but otherwise we discuss work closely related to our contributions within their respective chapters.

Chapter **2**

Higher-Order Model Checking

2.1 Higher-order recursion schemes

In the seventies and early eighties, a popular approach to the semantics of programming languages was to consider programs as systems of recursive equations and then to interpret the equations by means of a two stage process. In the first stage, the equations are solved in a free algebra of infinite trees and then in the second stage, the infinite trees are interpreted by some appropriate algebra homomorphism in order to obtain the overall meaning of the program. In separating the two stages, control structure of the program can be studied independently of the meaning of the associated operations on data, with applications in program transformation, program equivalence and definability. A good introduction to the subject is Guessarian's *Algebraic Semantics* [Guessarian, 1981]; Courcelle and Nivat [1978] compiled an excellent survey in an MFCS article from 1978, but a more up-to-date reference is Courcelle's handbook article [Courcelle, 1990].

In this early work the recursive equations, or *recursive program schemes*, were first order, in the sense that the unknowns stood in place of ground type data or first-order functions. It was not long until the setting was generalised to higher-orders, with Damm, Fehr and Indermark laying much of the groundwork in a series of papers from around 1980 [Damm, 1977, Damm et al., 1978, Damm, 1982]. Although recursion schemes were relatively neglected by mainstream research during much of the nineties, in the last decade there has been a resurgence in popularity. This is in large part due to the recent recognition of the important role played by so-called *finitely presentable infinite structures* in the modelling of software systems for the purpose of automated verification. In this setting, it is the fact that recursion schemes give rise to finite and infinite labelled trees that is most interesting.

Syntax

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

An important feature of higher-order recursion schemes is that they obey a typing discipline. This ensures that they are sufficiently constrained in their expressive power to be interesting from an algorithmic as well as language-theoretic point of view. For the purposes of higher-order model checking, the typing discipline is usually taken to be a simple type system over a single atom, o, but let us make two remarks. First, to have a broader set of base types does not create any problems but it complicates things in a way that is unnecessary for this dissertation. Second, it was shown by Tsukada and Kobayashi [2010] that to ensure decidable safety property checking it is necessary only for the scheme to be typable in a particular class of intersection type system. This result is particularly interesting since it implies that, for the purposes of model checking, it is possible to consider recursion schemes which obey an ML-style, polymorphic type system. Nevertheless, we will constrain ourselves to the usual definition and study recursion schemes that are simply typed. However, because *intersection* types will play such a large part in this dissertation we will refer to simple types as *kinds*, leaving us free to use the word *types* to refer exclusively to intersection types.

2.1.1 Definition (Simple kinds)**.** The *kinds* over o, denoted $(\kappa \in)\ \mathbb{S}$, are formed by the grammar:

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

As usual, we use parentheses to disambiguate the structure of such expressions, observing that the arrow associates to the right. The *arity* and *order* of a kind are natural numbers determined by the following functions:

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

If a kind has order 0 (and hence has arity 0) we say that it is *ground*. A *kind environment*, Δ, is a finite, partial function mapping variables and function symbols to kinds.

Of course, the terms play all the major roles and, of all them, the applicative terms (those without any occurrence of a lambda abstraction) do most of the work. Terms are built from variables, function symbols and constants using application and n-ary abstraction. The use of n-ary abstractions and the annotation of abstractions by the kinds of the variables is a convenience which, as we shall see shortly, makes the kind assignment system better behaved for our purposes.

2.1.2 Definition (Terms)**.** Let $(a,\ b,\ c \in)\ \Sigma$ be a set of atomic constants. The set of *terms* over Σ, is defined by the grammar:

$$s,\ t,\ u,\ v \quad ::= \quad x \mid F \mid c \mid s\,t \mid \lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.\,t$$

The *free variables* of a term t, are the subset of $\mathcal{V}$ denoted $\mathsf{FV}(t)$, is defined as follows:

$$\begin{array}{lcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(F) & = & \varnothing \\ \mathsf{FV}(a) & = & \varnothing \\ \mathsf{FV}(s\,t) & = & \mathsf{FV}(s) \cup \mathsf{FV}(t) \\ \mathsf{FV}(\lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.\, t) & = & \mathsf{FV}(t) \backslash \{x_1, \ldots, x_n\} \end{array}$$

Those variables in a term t that are not in $\mathsf{FV}(t)$ are called *bound*. A term t with $\mathsf{FV}(t)$ empty is called *closed*. A term t which contains no occurrence of an abstraction is called *applicative*. We will follow Barendregt's convention 2.1.12 of [Barendregt, 1984] and identify those terms that differ only in a choice of bound variable names. A *substitution* is a finite, partial function mapping variables and function symbols to terms. The action of a (capture-avoiding) substitution ρ on a term t is written by $t[\rho]$ and defined in the usual way.

Notation. We will feel free to omit the kind annotations on abstractions whenever they are unimportant or clear from the context. We will often simply write the maplets of the substitution directly when we do not need to give it a name, e.g. $t[s_1/x_1, \ldots, s_n/x_n]$ is the term resulting from substituting each occurrence of x_i by s_i ($i \in [1..n]$) in t.

All the situations that we shall consider in this dissertation will take place relative to a *signature* which declares a number of first-order term constants along with their arities.

2.1.3 Definition (Signature)**.** A first-order *signature*, is a set of atomic constants Σ equipped with a ranking function rank which maps each constant $c \in \Sigma$ to its *rank*, which is the arity of the constant. Since constants are first-order, every signature gives rise to a notion of kinding, that assigns to each constant c of rank n its *kind* which is the simple type defined as follows:

$$\mathsf{kind}(c) = \underbrace{o \to \cdots \to o}_{n\text{-times}} \to o$$

Starting from the kinding of the constants defined by the signature, kinding is extended to all terms through the usual system for simple type assignment. However, note that, as a technical convenience, we do not consider general terms of the simply typed lambda calculus since we constrain n-ary abstractions to have ground kind bodies.

2.1.4 Definition (Kind assignment in Σ)**.** A *kind term judgement* is an expression of the form $\Delta \vdash t : \kappa$ which is provable in the system for simple kind assignment in Figure 2.1. As usual, we use the notation $\Delta, x : \kappa$ to denote the environment $\Delta \cup \{x : \kappa\}$ whenever $x \notin \mathsf{dom}(\Delta)$. A *well-kinded term* over t is one for which there is a derivable judgement $\Delta \vdash t : \kappa$. Note that derivations of a given judgement are unique. We will frequently refer to the *order* and *arity* of a kinded term $\Delta \vdash t : \kappa$ as a shorthand for the order and arity of its kind κ. In this way, the arity of a kinded term $\Delta \vdash c : \kappa$ is identified with the rank of c.

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

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

$$\frac{\Delta,\, x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash t : o}{\Delta \vdash \lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.\, t : \kappa_1 \to \cdots \to \kappa_n \to o}\ \text{(S-Abs)}$$

Figure 2.1: Kind assignment.

A higher-order recursion scheme (HORS), is just a collection of equations between kinded terms where the equations have a particular form. The shape of each is:

$$F = \lambda x_1 \cdots x_n.\, t$$

in which t is an applicative term whose free variables are constrained to be among the parameters $\{x_1, \ldots, x_n\}$. The idea is that the function symbol F is *defined* by the abstraction $\lambda x_1 \cdots x_n.\, t$. An alternative notation which, as we shall see, perhaps better reflects the standard definition of reduction for schemes is the following:

$$F\, x_1 \cdots x_n = t$$

With this notation the resemblance with functional programs is clear. Both are common in the literature but we shall stick with the former.

Due to their roots in language theory, recursion schemes are sometimes classified as a kind of higher-order grammar and some of the associated nomenclature is drawn from this field. In particular, the function symbols that appear in equations are called *non-terminal symbols* and the constants are called *terminal symbols*.

2.1.5 Definition (Higher-order recursion schemes)**.** A *higher-order recursion scheme* $\mathcal{G}$ is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ consisting of the following components:

- Σ is a signature. In the context of recursion schemes, we shall commonly refer to the domain of this signature as the set of *terminal* symbols.
- $\mathcal{N}$ is a finite set of kinded *non-terminal* symbols; that is, a map from a finite subset of $\mathcal{F}$ to kinds in $\mathbb{S}$ of arbitrary order. We will often view this mapping as a kind environment and hence typically display its contents as a finite set of typings of the form $F : \kappa$.
- $\mathcal{R}$ is a function mapping each non-terminal symbol $F : \kappa_1 \to \cdots \to \kappa_n \to o$ to its definition, which is a term of the form $\lambda x_1 \cdots x_n.\, t$ with t an applicative term. We will typically view this mapping as a finite set of equations and write $F = \lambda x_1 \cdots x_n.\, t$

whenever $\mathcal{R}(F) = \lambda x_1 \cdots x_n.\, t$. For technical convenience we shall assume that the mapping is injective.

- S is a distinguished non-terminal symbol of ground kind called the *start* symbol.

A higher-order recursion scheme is well kinded just if, for each $F : \kappa \in \mathcal{N}$, the judgement $\mathcal{N} \vdash \mathcal{R}(F) : \kappa$ is provable. Observe that one consequence of this requirement is that the kinds annotating the abstractions in the image of $\mathcal{R}$ are therefore completely determined by $\mathcal{N}$ and hence we will typically omit them. The *order* of a recursion scheme is the highest order of (the kind of) any of its non-terminal symbols. We shall exclusively consider well-kinded higher-order recursion schemes in what follows.

We will sometimes introduce particular instances of recursion schemes by giving their rules, following the convention that non-terminals start with an upper-case letter and terminals start with a lower-case letter. The variables in a scheme are always clear since they are bound by the defining abstractions. Let us see some examples of recursion schemes.

2.1.6 Example. *The "standard" example of a recursion scheme is the following. It is a scheme built over the constants a of arity 2, b of arity 1 and c of arity 0. The scheme defines two non-terminals, the start symbol $S : o$ and a function $F : o \to o$. The non-terminals are defined by the rules:*

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

To see that the scheme is well kinded observe that, by the constant rule, $\vdash a : o \to o \to o$, $\vdash b : o \to o$ and $\vdash c : o$. Also notice that it follows from the definition that the body t of each function definition $\lambda x_1 \cdots x_n.\, t$ is necessarily of kind o. In this case $\mathcal{N} \vdash F\,c : o$ and $\mathcal{N},\, x : o \vdash a\,x\,(F\,(b\,x)) : o$.

2.1.7 Example. *Although the standard example has many virtues, it is unfortunately only a first order scheme — the highest order of any of its non-terminals is the order of F, which is 1. Hence, let us consider a variation of the standard example which defines three non-terminal symbols: $S : o$, $F : (o \to o) \to o$, $B : (o \to o) \to o \to o$ and $\Omega : o$. The non-terminals are defined by the rules:*

$$\begin{array}{rcl} S & = & F\,b \\ F & = & \lambda f.\, a\,(f\,\Omega)\,(F\,(B\,f)) \\ B & = & \lambda g\,x.\, g\,(g\,x) \\ \Omega & = & \Omega \end{array}$$

This variation is a second order scheme, since both F and B have second order kinds. Note the degenerate equation $\Omega = \Omega$ is perfectly valid according the definition above since $\mathcal{N} \vdash \Omega : o$.

Scheme reduction

Higher-order recursion schemes are endowed with a notion of behaviour by orienting the equations from left to right and treating them as rewrite rules. Much like in the operational semantics of functional programs (and the usual reduction of combinator systems), the notion of rewriting is *weak*, in the sense that only fully applied non-terminals are reduced.

2.1.8 Definition (Scheme reduction)**.** Given a recursion scheme, the associated notion of *scheme reduction* is the binary relation $\rhd_w$ between applicative terms, defined as the compatible closure of the following rule:

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

We say that a ground kind term of the form $F\, s_1 \cdots s_n$ is a *redex*. A *scheme reduction sequence* is a, possibly empty, finite sequence of terms $\langle s_i \rangle_{i=0}^{n}$ in which, for all $i \in [0..n-1]$, $s_i \rhd s_{i+1}$. We say that a reduction sequence $\langle s_i \rangle_{i=0}^{m}$ is *complete* just if s_m contains no function symbols.

Notation. Since scheme reduction is the primary notion of reduction that we shall consider in this dissertation, we shall sometimes just refer to it, without further qualification, as *reduction* and write $\rhd$ for the infix relation.

In his extensive monograph Damm [1982] considered two more specialised sorts of reduction for higher-order grammars, namely *innermost-outermost* and *outermost-innermost*. In the former, redexes are only contracted if they do not properly contain any other redex and in the latter redexes are only contracted if they are not properly contained by any other redex. Damm was able to show that, for the purposes of reaching normal forms, it was sufficient to consider only outermost-innermost reduction, in the sense that if it is possible to reduce to some ground term t not containing any function symbols and without any restriction on the position of redexes that are reduced, then it must be possible to reduce to t in the outermost-innermost reduction mode. See also the work of de Miranda [2006] and Haddad [2012] for two more recent studies. We shall, however, not concern ourselves with any restrictions on reduction in this dissertation, we say that reduction is *unrestricted*.

2.1.9 Example. *Recall the scheme from Example 2.1.6. Starting from S, one possible reduction sequence is as follows:*

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

In fact, since there is exactly one redex in each term of the sequence, this is the only reduction sequence of length five.

2.1.10 Example. *Recall the scheme from Example 2.1.7. Starting from S, one possible reduction is as follows:*

$$\begin{array}{lll} S & \rhd & F\,b \\ & \rhd & a\,(b\,\Omega)\,(F\,(B\,b)) \\ & \rhd & a\,(b\,\Omega)\,(a\,(B\,b\,\Omega)\,(F\,B\,(B\,b))) \\ & \rhd & a\,(b\,\Omega)\,(a\,(b\,(b\,\Omega))\,(F\,B\,(B\,b))) \end{array}$$

another reduction sequence of the same length is, by choosing a different final redex:

$$\begin{array}{lll} S & \rhd & F\,b \\ & \rhd & a\,(b\,\Omega)\,(F\,(B\,b)) \\ & \rhd & a\,(b\,\Omega)\,(a\,(B\,b\,\Omega)\,(F\,(B\,(B\,b)))) \\ & \rhd & a\,(b\,\Omega)\,(a\,(B\,b\,\Omega)\,(a\,(B\,(B\,b)\,\Omega))\,(F\,(B\,(B\,(B\,b))))) \end{array}$$

and yet another (quite boring) reduction sequence just contracts the Ω redex repeatedly:

$$\begin{array}{lll} S & \rhd & F\,b \\ & \rhd & a\,(b\,\Omega)\,(F\,(B\,b)) \\ & \rhd & a\,(b\,\Omega)\,(F\,(B\,b)) \\ & \rhd & a\,(b\,\Omega)\,(F\,(B\,b)) \end{array}$$

Decomposition of reduction

In this subsection we will decompose scheme reduction into two more elementary notions, namely δ-reduction and weak β-reduction. This is not required by any standard presentation of recursion schemes, but it will be useful for illuminating the relationship between reduction and (co-)consistent type environments in Chapter 3 and for showing computational adequacy of an interpretation in Chapter 5. Hence, the reader only interested in the standard presentation can safely skip ahead to the next subsection which concerns the definition of the value tree of a scheme.

We will decompose weak reduction into δ-reduction, which unfolds the definitions of non-terminal symbols according to the rules in the scheme, and weak β-reduction which contracts the newly created weak β-redexes. Let us start with δ-reduction:

2.1.11 Definition (δ-reduction)**.** Given a scheme $\mathcal{G}$ we define the notion of δ-*reduction*, $\rhd_\delta$, on terms as the compatible closure of the rule:

$$\frac{}{F \rhd_\delta \mathcal{R}(F)}\ (\delta)$$

A δ-*reduction sequence* is defined analogously with scheme reduction sequences.

Weak β-reduction will only contract ground redexes and all the associated parameters are substituted simultaneously. Furthermore, weak β-contractions do not occur underneath a lambda.

2.1.12 Definition. Given a scheme $\mathcal{G}$ we define the notion of *weak β-reduction*, $\rhd_\beta$, on terms as the closure of the rule:

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

under the term forming operation of application, but *not* abstraction. A *weak β-reduction sequence* is defined analogously with scheme reduction sequences. We write the union of the two relations $\rhd_\delta$ and $\rhd_\beta$ as $\rhd_{\delta\beta}$.

Given an applicative term s, we will show that any $\delta\beta$-reduction sequence from s to some term t involving k weak β-reductions can be replayed as a length k weak reduction sequence from s to some term t'. If there were more δ-reductions than weak β-reductions in the sequence leading to t, then t may contain extraneous lambda abstractions. Hence, we relate t to t' by folding them up again, defined as follows.

2.1.13 Definition. Define the *fold*, s^+, of a term s inductively by the following clauses:

$$\begin{array}{lcl}
x^+ & = & x \\
F^+ & = & F \\
c^+ & = & c \\
(u\, v)^+ & = & u^+\, v^+ \\
(\lambda x_1 \, \cdots \, x_n.\, t)^+ & = & \begin{cases} F & \text{if } \lambda x_1 \, \cdots \, x_n.\, t^+ = \mathcal{R}(F) \\ \lambda x_1 \, \cdots \, x_n.\, t^+ & \text{otherwise} \end{cases}
\end{array}$$

Of course, the folding up of definitions undoes all the work of δ-reduction.

2.1.14 Lemma. *In any recursion scheme, if $s \rhd_\delta^* t$ then $s^+ = t^+$.*

Proof. We prove by induction on the proof of reduction that $s \rhd_\delta t$ implies $s^+ = t^+$ and then the result follows by induction on the length of the sequence.

- If the reduction is by contraction of a δ-redex then $s = F$ and $t = \mathcal{R}(F)$. Since $\mathcal{R}(F)$ is of the form $\lambda x_1 \, \cdots \, x_n.\, r$ with r applicative, it follows that $s^+ = F = t^+$.
- If the reduction is on the left of an application, $s = u\, v$ and $t = u'\, v$ and $u \rhd_\delta u'$. Therefore, $s^+ = u^+\, v^+$, $t^+ = u'^+\, v^+$ and the result follows from the induction hypothesis. The case of reduction on the right of an application is symmetric.
- If the reduction is underneath an abstraction, $s = \lambda x_1 \cdots x_n.\, u$ and $t = \lambda x_1 \cdots x_n.\, u'$ and $u \rhd_\delta u'$. It follows from the induction hypothesis that $u^+ = u'^+$ and hence $s^+ = t^+$.

□

The rest of the subsection is devoted to proving that any $\delta\beta$-reduction sequence can be lifted to a weak reduction sequence over the folded terms. More precisely:

$$s \rhd^*_{\delta\beta} t \qquad \text{implies} \qquad s^+ \rhd^*_w t^+$$

The proof of the result is the composition of two further facts. The first is that, on its own, β-reductions can be lifted to a weak reduction sequence between folded terms. This is because, every time a β-redex $\mathcal{R}(F)\ s_1\ \cdots\ s_n$ is contracted, there is a corresponding scheme redex $F\ s_1\ \cdots\ s_n$ in the folded version.

2.1.15 Lemma. *In any recursion scheme, if $s \rhd^k_\beta t$ then $s^+ \rhd^k_w t^+$.*

Proof. We show that, for all terms s, if $s \rhd_\beta t$ then $s^+ \rhd_w t^+$ by induction on the witness to the reduction.

- If the reduction is by contracting a weak β-redex, then necessarily s is of the form $\mathcal{R}(F)\ s_1\ \cdots\ s_n$ and $t = r[s_1/x_1, \ldots, s_n/x_n]$ whenever $\mathcal{R}(F) = \lambda x_1\ \cdots\ x_n.\, t$. Necessarily $s^+ = F\ s_1^+\ \cdots\ s_n^+$ and $t^+ = r[s_1^+/x_1, \ldots, s_n^+/x_n]$ (since r is applicative).
- If the reduction is by reducing on the left of an application, then s is of the form $u\, v$ and $u \rhd_\beta u'$ and $t = u'\ v$. Then $s^+ = u^+\ v^+$ and $t^+ = u'^+\ v^+$ and the result follows from the induction hypothesis. The case for reducing on the right is similar.

The result then follows by induction on the length of the reduction sequence. □

The second fact is that, because it never destroys β-redexes but only creates them, δ-reduction can always be moved to the front in any $\delta\beta$-reduction sequence.

2.1.16 Lemma. *In any recursion scheme, if $s \rhd^*_{\delta\beta} t$ then there is some u such that $s \rhd^*_\delta u \rhd^*_\beta t$.*

Proof. By induction on the length ($= n$) of the reduction sequence. If $n = 0$ then take $u = s = t$. Otherwise n if of the form $k+1$. Let us assume then that the sequence is of shape $s \rhd^*_{\delta\beta} t' \rhd_\delta t$ (in the other case the result follows immediately from the induction hypothesis). To see the result, observe that it is always possible to exchange such reductions, i.e. for all s, t and u: if $s \rhd_\beta t \rhd_\delta u$ then there is some t' such that $s \rhd_\delta t' \rhd_\beta u$. We prove this by induction on the proof of the weak β reduction. If the reduction was by a weak β-contraction, then s is of the form $(\lambda x_1\ \cdots\ x_n.\, r)\ s_1\ \cdots\ s_n$ and t is of the form $r[s_1/x_1, \ldots, s_n/x_n]$ (and there is some F such that $\mathcal{R}(F) = \lambda x_1 \cdots x_n.\, r$). Let us say that the second reduction is by delta contracting some non-terminal G. Then either G occurs already in r, in which case $r = C[G]$ for some context C and the result follows from taking $t' = (\lambda x_1 \cdots x_n.\, C[\mathcal{R}(G)])\, s_1 \cdots s_n$, or G occurs already in one of the arguments $s_i = C[G]$ and so the result follows from taking $t' = (\lambda x_1\ \cdots\ x_n.\, r)\ s_1\ \cdots\ s_{i-1}\ C[\mathcal{R}(G)]\ s_{i+1}\ s_n$. If the reduction was by reducing on the left of an application, then s is of the form $v\, w$ and $v \rhd_\beta v'$ and $t = v'\ w$. If G occurs in v' then, by the induction hypothesis, the reduction $v \rhd_\beta v' \rhd_\delta v''$ can be rearranged to $v \rhd_\delta v''' \rhd v''$ and hence take t' to be $v'''\ w$. Reduction on the right is proved symmetrically. □

Then the desired result is a corollary of the previous two facts and also the observation that, on its own, δ-reduction is irrelevant when lifted to δ-expansion normal forms.

2.1.17 Corollary. *In any recursion scheme:*

$$s \rhd^*_{\delta\beta} t \quad \textit{implies} \quad s^+ \rhd^*_w t^+$$

Proof. Follows immediately from Lemmas 2.1.15, 2.1.14 and 2.1.16. □

The value tree of a scheme

We said in the introductory comments of this section that much of the recent interest in recursion schemes has been due to their ability to define finite and infinite labelled trees for the purposes of modelling systems for verification. Let us now see how the trees come about. First, we employ the usual formalisation of labelled tree.

2.1.18 Definition (Labelled trees)**.** Let A be a set without restriction. An *A-labelled tree* is a partial function $T : \mathbb{N}^* \to A$ whose domain is prefix closed. In case the set A is equipped with a ranking $\mathsf{rank} : A \to \mathbb{N}$, then there is a corresponding notion of ranked tree. An *A-ranked and labelled tree* is an A-labelled tree T which, moreover, satisfies the condition: if $T(w) = a$ and $\mathsf{rank}(a) = n$ then: $w \cdot i \in \mathsf{dom}(T)$ iff $i \in [1..n]$.

To be clear, let us see some examples:

2.1.19 Example. *Fix the signature Σ of Example 2.1.6. An example of a Σ-ranked and labelled tree is the structure T given by:*

$T(\epsilon)$	$=$	a	$T(211)$	$=$	c
$T(1)$	$=$	c	$T(221)$	$=$	b
$T(2)$	$=$	a	$T(222)$	$=$	a
$T(21)$	$=$	b	$T(2211)$	$=$	b
$T(22)$	$=$	a	$T(22111)$	$=$	c

with the graphical depiction of the same structure to the right. As another example, the full, infinite binary tree B over a can be specified by, for all $w \in \{1, 2\}^*$, $B(w) = a$.

We will use term notation for trees wherever possible, which we prefer for its brevity. We will more often write the first tree of the foregoing example as $a\ c\ (a\ (b\ c)\ (a\ (b\ (b\ c))\ a))$ and the second as, e.g. $a\ (a\ B\ B)\ B$.

Recursion schemes define trees by reduction and the idea is that the trees defined this way should be labelled by the terminal symbols in the signature of the scheme. Roughly speaking, the tree of a scheme comes about by constructing an *infinite* reduction sequence starting from S. In the limit of the sequence is a (possibly infinite) term which has had all of its non-terminals rewritten away and we view that term as a (possible infinite) tree.

However, trees defined in this way can have branches missing, which is unsatisfactory. For example, recalling the definition of Ω from Example 2.1.7, the non-terminal Ω is never "re-written away" in the tree defined by the scheme $S = a\ c\ \Omega$ since each contraction of an Ω-redex creates a new one. If we leave the tree undefined at 2 then it will not be well ranked, which will make life technically inconvenient. Hence, the convention is to adopt a new symbol $\bot$, which stands for (unproductive) non-termination or undefinedness. Trees can be ordered according to definedness in a natural way.

2.1.20 Definition (The domain of finite and infinite trees)**.** Let Σ be a signature not including $\bot$. We write $\Sigma^\bot$ for the set $\Sigma \cup \{\bot\}$ with $\mathsf{rank}(\bot) = 0$. Then $\Sigma^\bot$ can be ordered by setting $c_1 \sqsubseteq c_2$ just if $c_1 = c_2$ or $c_1 = \bot$. The order can be naturally extended to the set of all non-empty, $\Sigma^\bot$-ranked and labelled trees, $\Sigma^\bot\mathsf{Trees}^\infty$, by $T_1 \sqsubseteq T_2$ iff for all $w \in \mathsf{dom}(T_1)$, $T_1(w) \sqsubseteq T_2(w)$. Under this order, $\Sigma^\bot\mathsf{Trees}^\infty$ is a directed complete, partially ordered set with a least element (the least element is the tree consisting of only a root labelled by $\bot$). With each symbol $c \in \Sigma^\bot$ of rank n we assign an n-ary continuous function $\underline{c}$ on $\Sigma^\bot\mathsf{Trees}^\infty$ (a tree in case $n = 0$) recursively by the following clauses:

$$\begin{array}{lcll} \underline{c}(T_1)\cdots(T_n)(\epsilon) & = & c & \\ \underline{c}(T_1)\cdots(T_n)(i \cdot w) & = & T_i(w) & \text{when } i \in [1..n] \text{ and } w \in \mathsf{dom}(T_i) \end{array}$$

When c is a just a tree ($n = 0$), we will typically omit the underline since no confusion can arise.

Under the ordering of trees given in the definition, we have that one tree is smaller (less well defined) than another just if it is possible to obtain the former by replacing some subtrees of the latter by bottom.

2.1.21 Example. *Consider the following inequalities, written using term notation:*

$$a \bot\bot \quad \sqsubseteq \quad a \bot (a\,(b\,c)\,\bot) \quad \sqsubseteq \quad a\,c\,(a\,(b\,c)\,\bot) \quad \not\sqsubseteq \quad a\,(b\,c)\,(a\,(b\,c)\,\bot)$$

We can use the ordering defined above to make precise the development of a tree through the reducts of some ground kind term. For example, consider again the reduction sequence from Example 2.1.9:

$$S \rhd F\,c \rhd a\,c\,(F\,(b\,c)) \rhd a\,c\,(a\,(b\,c)\,(F\,(b\,(b\,c)))) \rhd a\,c\,(a\,(b\,c)\,(a\,(b\,(b\,c))\,(F\,(b\,(b\,(b\,c))))))$$

It is possible to see the tree developing by projecting out the completed prefix of terminals from each term in the sequence. For the first two terms in the sequence there is no prefix, then there is a prefix of a tree rooted at a with left child labelled by c and then the right child is fleshed out with a sub-prefix rooted at a with left child labelled by b and so on. We define how to project out the prefix of the tree from a term.

2.1.22 Definition. Given a ground kind term t we embed it as $t^{\perp}$ into the domain of finite and infinite $\Sigma^{\perp}$-labelled trees in a way which is defined recursively by the following clauses:

$$\begin{array}{lcl} (F\, s_1 \,\cdots\, s_n)^{\perp} & = & \perp \\ (x\, s_1 \,\cdots\, s_n)^{\perp} & = & \perp \\ (a\, s_1 \,\cdots\, s_n)^{\perp} & = & \underline{a}(s_1^{\perp}) \cdots (s_n^{\perp}) \\ ((\lambda x_1 \,\cdots\, x_n.\, t)\, s_1 \,\cdots\, s_n)^{\perp} & = & \perp \end{array}$$

Finally we define the tree generated by a term and hence the tree generated by a scheme. The idea is that the tree generated by a term t is the result of gluing together all of the finite prefixes of trees that can be reached as a result of reducing in all possible ways from t. The tree generated by a scheme is just the tree generated by the start symbol of that scheme. Since scheme reduction is confluent, it follows that the set of such prefixes is directed and hence there is a well-defined limit.

2.1.23 Definition (Value tree)**.** For any ground kind term s, the *tree generated by* s, $\mathsf{Tree}(s)$, is defined as:

$$\mathsf{Tree}(s) = \bigsqcup\{t^{\perp} \mid s \rhd_w^* t\}$$

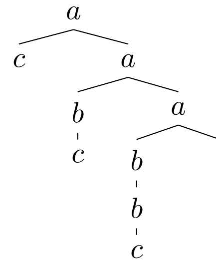
The *value tree* $\mathsf{Tree}(\mathcal{G})$ of a given recursion scheme $\mathcal{G}$ is just $\mathsf{Tree}(S)$. Later, we shall need to talk about trees generated by weak β-reduction and so we extend the definition in the obvious way, writing $\mathsf{Tree}_{\beta}(s)$ for $\bigsqcup\{t^{\perp} \mid s \rhd_{\beta}^* t\}$.

Note that, in the domain of finite and infinite trees, the least upper bound acts a little like a union operation on trees that ignores the presence of bottom. Let us consider the two main examples we introduced at the start of the section.

2.1.24 Example. *Consider again the scheme from Example 2.1.6. We recount the definition of the rules below for convenience:*

$$\begin{array}{lcl} S & = & F\, c \\ F & = & \lambda x.\, a\, x\, (F\, (b\, x)) \end{array}$$

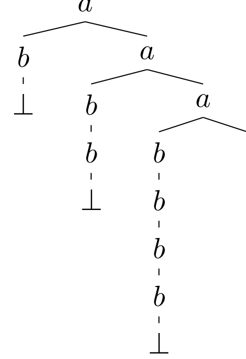
The start symbol S generates an infinite a, b and c-labelled tree, which is depicted on the right. This tree has no $\perp$-labelled nodes since, except for the first, every redex contraction produces a new terminal symbol in head position.



2.1.25 Example. *Consider again the scheme from Example 2.1.7. We recount the definition of the rules below for convenience:*

$$\begin{array}{rcl} S &=& F\,b \\ F &=& \lambda f.\, a\,(f\,\Omega)\,(F\,(B\,f)) \\ B &=& \lambda g\,x.\, g\,(g\,x) \\ \Omega &=& \Omega \end{array}$$

Here the start symbol S generates an infinite a, b and $\perp$-labelled tree, which we depict on the right. Due to the non-productiveness of Ω, this tree contains $\perp$ on every branch off the spine.



Extensions of recursion schemes and alternative presentations

The most striking feature of the reduction associated with recursion schemes is that it is not influenced by the constants over which such schemes are built. In a sense, recursion schemes are "write-only" programs. Hence, the complex control flow of a recursion scheme stems entirely from the interactions between (higher-order) functions. Though this is quite an elegant model of computation, it is not always convenient, and several authors have sought to extend recursion schemes with new features that influence the control-flow in more obvious ways. Of course, since recursion schemes are essentially ground type terms of the simply typed lambda calculus with fixed point combinators, it is already possible to encode many desirable features in the existing formalism. However, such encodings are often considered undesirable, let us remark on two reasons. First, they typically raise the type-theoretic order of the scheme which, as we shall see shortly, is disastrous to the theoretical worst-case time complexity for model checking. Second, many algorithms are, in practice, tuned for schemes that resemble functional programs and this resemblance can disappear as a result of syntactically heavy encodings. Hence, several extensions have attempted to build such features directly into the schemes and deal with the consequences to model checking directly in the associated algorithms. As well as these variations on recursion schemes, there have been two other significant alternative presentations of the class of trees that they define, both of which are kinds of machine models. On the one hand, collapsible pushdown automata play the role of a classical, automata theoretic characterisation and Krivine machines play the role of a lambda calculus evaluator. In the remains of this section we will survey some of the literature surrounding these different structures.

Recursion schemes with finite data domains. Kobayashi, Tabuchi, and Unno [2010] extended recursion schemes with a kind of case statement over a finite enumeration in a restrictive type discipline, the resulting structures are called recursion schemes with finite data domains (RSFD). More specifically, they introduced an additional base kind d, the kind of *data*, and new nullary constants $\mathtt{d_1}, \ldots, \mathtt{d_n}$ and $(n+1)$-ary constant `case`. The reduction relation of RSFD has an extra rule to reflect the operational semantics of `case`,

which is the obvious one. The kind assignment to terms of the scheme is restricted so that, as expected, each `di` is given kind d and `case` is assigned the kind $d \to o \to \cdots \to o \to o$; but also non-terminals are restricted to have kinds of the form $\kappa_1 \to \cdots \to \kappa_n \to o$. In this way, though inputs to functions may be data values, no function ever outputs a data value itself. This restriction had the consequence that such schemes can be easily checked by a minor extension to existing recursion scheme model checkers. In particular, they are shown to have a decidable trivial automaton model checking problem by constructing a special purpose intersection type system that characterises it. Although the kind restriction seems quite a severe limitation on the expressive power RSFD, note that when the scheme is presented in continuation passing style (which is the case in Kobayashi's original proposal for using recursion schemes in the verification of functional programs [Kobayashi, 2009a]) the restriction no longer bites, since no functions return.

Weak pattern matching recursion schemes. We introduced pattern matching recursion schemes (PMRS) and their weak variant (wPMRS) in joint work with Ong [Ong and Ramsay, 2011]. These schemes are intended to capture the key features of computation over algebraic data types, which are specified by arbitrary first order terminal symbols. The general form of rules in such schemes is:

$$\begin{array}{rcl} F & = & \lambda x_1 \cdots x_n \, p_1 . \, t_1 \\ & \vdots & \\ F & = & \lambda x_1 \cdots x_n \, p_k . \, t_k \end{array}$$

in which each p_i is a pattern: a ground kind term containing no occurrence of a function symbol and whose only variables occur in the leaves. These families of rules were further divided into two classes. The *pattern matching* rules have no restriction on the patterns $p_1, \ldots, p_k$ except that they are required to be orthogonal and exhaustive. The *pure* rules require that the patterns are trivial, in the sense of being a variable. Hence, PMRS and wPMRS also allowed for the specification of some non-determinism. The schemes must be kinded as usual but, unlike RSFD, there are no special restrictions regarding the data types. In PMRS the bodies $t_1, \ldots, t_k$ may contain variables which are bound by the corresponding patterns. This leads to a very general form of pattern matching so that PMRS are Turing powerful. Hence, they have no interesting algorithmic properties. In wPMRS the bodies $t_1, \ldots, t_k$ may not contain variables bound by the corresponding patterns, which restricts the expressive power significantly. Indeed, we sketched a proof of the decidability of trivial automaton model checking for wPMRS though an intersection type system characterisation tailored to pattern matching. However, there is as yet no known efficient algorithm for deciding the problem.

Higher-order recursion schemes with cases. Higher-order recursion schemes with cases (HORSC) were introduced in our joint work with Neatherway and Ong [Neatherway et al., 2012]. The work was part of an on-going attempt to develop an efficient algorithm for the model checking of wPMRS. This work reported a intermediate step which concerned a class

of recursion schemes between RSFD and wPMRS in their expressive power. Like RSFD, they introduced a special kind for data d inhabited by a finite enumeration $\mathtt{d_1}, \ldots, \mathtt{d_n}$ and a case expression `case` of arity $n + 1$. However, like wPMRS, there are no special kinding restrictions on the schemes so that, in particular, non-terminals can return data. Furthermore, non-terminals can have several defining abstractions, so these schemes are non-deterministic. HORSC were shown to have a decidable trivial automaton model checking problem by characterisation in an intersection type system with ground kind unions, which captured the interaction of non-determinism with the case statement and led to an efficient decision procedure.

Collapsible pushdown automata and Krivine machines. Finally we consider two different presentations of the class of trees generated by recursion schemes. Tree-generating collapsible pushdown automata (CPDA) were introduced by Hague, Murawski, Ong and Serre as an automata-theoretic characterisation of this class. CPDA extend the higher-order pushdown automata of Maslov [1976] which, when formulated in a way which allows them to define a tree, have been shown to be equi-expressive with *safe* recursion schemes (those that fall within a certain syntactically defined subclass). The idea of higher-order pushdown automata is to have a stratified system of stacks analogous to the stratification of recursion schemes given by their order. A 0-level stack is just a stack of symbols as known from ordinary pushdown automata, but a $(k + 1)$-level stack is a stack of k-level stacks. The stack-manipulating operations of the automaton such as push and pop are similarly stratified. CPDA furthermore add the stack operation *collapse* which is used as a kind of iterated, higher-order pop in order to reach some previously saved stack configuration. This is the key ingredient in characterising the class of trees generated by recursion schemes. An alternative machine based characterisation is given by Salvati and Walukiewicz [2011] who present a proof of mu-calculus decidability for recursion schemes in terms of Krivine machines executing terms of the lambda-Y calculus. Since recursion schemes can be viewed quite naturally as ground type terms (with first-order free variables) of this calculus, and Krivine machines are a well understood and well-adapted machine model for the evaluation of terms of lambda calculi, the authors are able to demonstrate compact proofs of both local and global model checking.

2.2 Model checking problems for recursion schemes

A remarkable feature of recursion schemes is that the trees they define have a decidable monadic second order-theory. Equivalently, the mu-calculus model checking problem is decidable over these trees or, in automata theoretic terms, the acceptance problem for alternating parity tree automata is decidable. However, most work in model checking practice has focused on a particular fragment that expresses safety properties and this fragment is also the focus of this dissertation. The automaton model corresponding to this fragment is the class of alternating (co-)trivial tree automata and the first half of this section concerns their definition. The second half introduces the corresponding model checking problem (acceptance checking problem) and surveys the surrounding literature.

Alternating (co-)trivial automata

The alternation structure of an alternating (co-)trivial automaton is best given by means of an assignment of a propositional formula to each combination of state and constant.

2.2.1 Definition (Positive Boolean formulae)**.** Given a finite set X, the *positive Boolean formulas over* X, denoted $(\phi \in)\ \mathsf{B}^+(X)$, are defined by the following grammar:

$$\phi ::= \mathsf{t} \ \mid\ \mathsf{f} \ \mid\ x \ \mid\ \phi_1 \wedge \phi_2 \ \mid\ \phi_1 \vee \phi_2$$

Given a positive Boolean formula ϕ, an *assignment* is a finite subset S of X. An assignment S is said to be a *satisfying assignment* for ϕ, written $S \models \phi$, when assigning t to elements of S and f to elements of $X \backslash S$ makes ϕ true (under the usual interpretation of the Boolean connectives).

For alternating (co-)trivial automata, the atomic propositions are pairs (i, q) of a natural $i \in \mathbb{N}$ and a state q. The idea is that such a pair is true of a sequence of trees just if the ith member of the sequence is accepted by state q of the automaton. A transition $(q, c) \mapsto \phi$ should be read as asserting that for a tree with root labelled by c to be accepted from state q, it had better be that the children of the root, taken as a sequence, satisfy ϕ.

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

- The *signature*, $(a, b, c \in)\ \Sigma$, is a finite set of ranked constants.
- The *state space*, $(q \in)\ Q$, is a finite set.
- The *transition function*, δ, is a function in

$$\Pi_{(q,a)\in Q\times\Sigma}.\ \mathsf{B}^+([1..\mathsf{rank}(a)] \times Q)$$

- The *initial state*, q_0, is a distinguished element of Q.
- The *accepting states*, F are either all of Q or empty.

In case $F = Q$, we say that the ATT has a *trivial* acceptance condition, otherwise $F = \varnothing$ and we say that it has a *co-trivial* acceptance condition.

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

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

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

We say that a run tree R is *accepting* just if, on every infinite branch (path) of R, some state $q \in F$ occurs infinitely often. The *language* of an ATT $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of Σ-ranked and labelled trees T for which there exists an accepting run-tree on T.

We will most often introduce such a structure as an alternating trivial or alternating co-trivial automaton and omit the final component of the specification. Furthermore, when specifying the transition function as an input-output relation, we will feel free to omit those maplets whose image is f. In other words, unless otherwise stated, pairs $(q, c) \in Q \times \Sigma$ are assumed to map to false.

Deterministic (co-)trivial automata are a sub-class for which the transition function is restricted to specifying, for each state q and symbol c, either f or else exactly one condition for each child of c. Due to this restriction, it is quite common in the literature to specify such transitions by a partial mapping, associating pairs (q, c) with a sequence of states $q_1 \cdots q_n$; though we will avoid this in favour of uniformity.

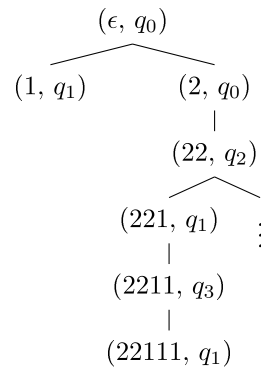
2.2.3 Definition (Deterministic (co-)trivial automata)**.** We shall say that an alternating (co-)trivial automaton is *deterministic* (DTT) just if, for each (q, c) with $\mathsf{rank}(c) = n$, the corresponding formula $\delta(q, c)$ is either false or has the form $(1, q_1) \wedge \cdots \wedge (n, q_n)$ for some $q_i \in Q$ $(i \in [1..n])$.

Alternating (co-)trivial automata lie at two extremes in terms of their acceptance condition. For trivial automata, since *every* state is accepting, no run trees are rejected on account of their infinite branches. Hence, for a tree to be accepted by a trivial automaton, it is sufficient for the automaton to have a run tree upon it. At the other extreme, co-trivial automata have *no* accepting states. Hence, a run-tree is accepting only if it is finite. Consequently, we will often think of acceptance by (co-)trivial automata just in terms of the existence of (finite) run trees.

2.2.4 Example. *Consider the alternating trivial automaton over the states* q_0, q_1, q_2 *and* q_3*, in which the transition function is defined by the following clauses:*

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

This automaton accepts those trees that have an infinite a-labelled spine and of every two consecutive branches off the spine, at least one is required to be labelled by an even number of b nodes terminated by a c. The run tree over the tree generated by the scheme in Example 2.1.24 is depicted to the right.



In this dissertation we will be interested in co-trivial automata only in so far as they help us to understand trivial automata. In particular, when we come to discuss our algorithm for model checking, it will be very useful to be able to reason not just about those terms that generate trees accepted by a trivial automaton, but also those terms that

do not. The terms that generate trees rejected by a given trivial automaton, generate trees that are accepted by its dual, which is a co-trivial automaton. The relationship is formalised as follows:

2.2.5 Definition (Complement)**.** Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ be a (co-)trivial automaton. Define the *de Morgan dual* $\mathcal{A}^c$ by $\langle \Sigma, Q, \delta^c, q_0, F^c \rangle$ where $\delta^c(q, a) = (\delta(q, a))^c$, the dual of a positive Boolean formula is defined by the following clauses:

$$\begin{array}{lcl} x^c & = & x \\ \mathsf{t}^c & = & \mathsf{f} \\ \mathsf{f}^c & = & \mathsf{t} \\ (\phi \wedge \psi)^c & = & \phi^c \vee \psi^c \\ (\phi \vee \psi)^c & = & \phi^c \wedge \psi^c \end{array}$$

and the dual of the set of final states is given by $F^c = Q \backslash F$.

2.2.6 Lemma (Complementation)**.** *Let $\mathcal{A}$ be a (co-)trivial automaton, then $\mathcal{L}(\mathcal{A}^c) = \mathcal{L}(\mathcal{A})^c$.*

Proof. The proof can be found in the paper of Muller and Schupp [1987]. □

Model checking problems

An instance of the model checking problem that we are interested in consists of two components: a recursion scheme $\mathcal{G}$ over a signature Σ which specifies a particular (possibly infinite) $\Sigma^\perp$-labelled tree $\mathsf{Tree}(\mathcal{G})$ and a (co-)trivial automaton $\mathcal{A}$ over Σ specifying the class of trees that are acceptable.

Hence, both the scheme $\mathcal{G}$ and the automaton $\mathcal{A}$ are built over the same signature Σ, but the tree defined by $\mathcal{G}$ may contain $\perp$. Kobayashi and Ong [2009] worked under the assumption that the particular recursion schemes they considered did not generate trees containing any occurrence of bottom. They note that it is always possible to manufacture such a scheme from an arbitrary one in such a way that the two are equivalent with respect to the modal mu-calculus model checking problem. We prefer a more direct approach, introduced in [Kobayashi et al., 2010], in which the tree may contain bottom and its treatment by the property is implicit in the problem. This approach requires the following two auxiliary definitions.

2.2.7 Definition. Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ be a trivial automaton whose alphabet Σ does not include any symbol $\perp$. Then define $\mathcal{A}^\perp = \langle \Sigma^\perp, Q, \delta \cup \{(q, \perp) \mapsto \mathsf{t} \mid q \in Q\}, q_0 \rangle$.

2.2.8 Definition. Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ be a co-trivial automaton whose alphabet Σ does not include any symbol $\perp$. Then define $\mathcal{A}_\perp = \langle \Sigma^\perp, Q, \delta \cup \{(q, \perp) \mapsto \mathsf{f} \mid q \in Q\}, q_0 \rangle$.

The idea is that, given a specification represented by a trivial automaton $\mathcal{A}$, the specification $\mathcal{A}^\perp$ is maximally liberal with respect to occurrences of $\perp$. This is consistent with the idea that we will use trivial automata to represent safety properties: if some part of the

scheme degenerates to unproductive non-termination, then certainly nothing “bad” ever comes from it. The definition for co-trivial automata is complimentary.

2.2.9 Proposition. *The following are obvious:*

- *For all trivial automata* $\mathcal{A}$*:* $(\mathcal{A}^c)_\perp = (\mathcal{A}^\perp)^c$.
- *The complement of a trivial automaton is a co-trivial automaton and visa versa.*

Using these two auxiliary definitions we are able to describe the corresponding model checking problems.

2.2.10 Definition. We define the following model checking problems.

(i) The *alternating trivial automaton model checking problem for higher-order recursion schemes* is, given a scheme $\mathcal{G}$ and a trivial automaton $\mathcal{A}$ over the same signature, to determine $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$.

(ii) The *alternating co-trivial automaton model checking problem for higher-order recursion schemes* is, given a scheme $\mathcal{G}$ and a co-trivial automaton $\mathcal{A}$ over the same signature, to determine $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}_\perp)$.

The decidability of the problems is a corollary of the landmark result of Ong [2006], who also showed, in joint work with Kobayashi [Kobayashi and Ong, 2011], that the (co-)trivial automaton model checking problem for order-n recursion schemes is complete for n-EXPTIME. Let us remind the reader that a problem belonging to this class can only be solved, in the worst case, in time $\mathcal{O}(\mathsf{exp}_n(x))$ for x a measure of the size of the input, where:

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

However, the analyses of Kobayashi [2009a] and Kobayashi and Ong [2009] showed that the problem is fixed-parameter polynomial time in the size of the scheme since, if the type-theoretic order of the scheme, the maximum arity of any of the symbols and the number of states of the automaton are fixed, there exist decision procedures whose worst-case time complexity is $\mathcal{O}(|\mathcal{G}|)$ with $|\mathcal{G}|$ the size of the scheme (the number of normalised rules).

2.3 The intersection type characterisation

Kobayashi [2009a] introduced an approach to higher-order model checking based on a characterisation of the problem in an intersection type system. The approach became highly influential, with many of the subsequent algorithms, including the one that we will present in Chapter 4, being grounded in the intersection type approach. The key is a certain characterisation theorem for the trivial automaton model checking problem:

> *Fix a recursion scheme $\mathcal{G}$ and an alternating trivial automaton $\mathcal{A}$. Then* $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A})$ *if, and only if, there exists a $(\mathcal{G}, \mathcal{A})$-consistent*[1] *intersection type environment Γ such that $\Gamma \vdash S : q_0$.*

In other words, model checking is shown to be equivalent to type assignment in a certain intersection type system. The theorem was soon generalised by Kobayashi and Ong [2009] so as to capture the full modal mu-calculus model checking problem via alternating parity automata, but since our main concern is to prepare the reader for the rest of this dissertation, we will restrict ourselves to the trivial automaton fragment. The rest of this section introduces the content of the theorem through a description of the intersection type system, type assignment and the consistency condition. However, the definitions that we describe in this section should be considered informal: they have been included only to aid the explanation and, since the rest of the dissertation will be based on our extension to Kobayashi's type system in Chapter 3, they will be superseded by the formal material of that chapter.

Intersection types

Intersection types, first studied in the late seventies by Coppo and Dezani [1978, 1980] and, independently, Sallé [1978], arose from of a desire to extend Curry's *basic functionality* to meaningfully assign types to a larger set of lambda terms. In particular, fixed point combinators, such as $Y = \lambda x. (\lambda y. x\,(y\,y))(\lambda y. x\,(y\,y))$, which were of obvious interest, did not receive types in Curry's system and, perhaps even more unsatisfactorily, neither did some normal forms such as $\lambda x. xx$. Intersection type systems rectified these deficiencies in a particularly strong way, providing the first type-theoretic characterisations of the strongly normalising terms and, later, all normalising terms. However, intersection type systems proved to be much more than just a new way to classify terms. Starting with the seminal paper of Barendregt, Coppo, and Dezani-Ciancaglini [1983], intersection types have been shown to be an important material with which to construct and compare models of untyped lambda calculus. An excellent introduction is given by Hindley [1992] and a comprehensive exposition can be found in the third part of the recently published handbook of Barendregt, Dekkers and Statman [Barendregt et al., 2013].

The intersection type discipline added to Curry's arrow type former also the intersection operator. The idea is that, whenever σ_1 and σ_2 are types describing some particular aspects of the behaviour of a term t, then to assign to t a type such as $\sigma_1 \wedge \sigma_2$ is to assert that t behaves *both* in the way described by σ_1 *and* in the way described by σ_2. If we view σ_1 and σ_2 as being representations of two sets, then the statement $t : \sigma_1 \wedge \sigma_2$ represents the fact that t is an element of the intersection of the set represented by σ_1 and the set represented by σ_2; and from this the name of the typing discipline is given. It is by using the intersection that terms like $\lambda x. x\,x$ can be given a type, since types of the

[1] Kobayashi originally called such type environments *complete*, but later referred to them simply by a judgement form $\vdash_{\mathcal{A}} \mathcal{G} : \Gamma$. We prefer the name *consistent* for reasons that will become clear shortly.

form $((\sigma_1 \to \sigma_2) \wedge \sigma_1) \to \sigma_2$ can be used to express the fact that the bound variable x is used *both* as a function *and* as an argument to that very same function.

Kobayashi's system is parametrised by a trivial automaton, so let us fix automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ for the remainder of this section. The types of his system are stratified in such a way that intersection cannot occur in the tail of any arrow. His intersection types are defined by the following grammar:

$$\begin{array}{rcl} \tau & ::= & q \mid \sigma \to \tau \\ \sigma & ::= & \bigwedge_{i=1}^{n} \tau_i \end{array}$$

in which q is drawn from the set of states Q of the automaton and $n \geqslant 0$. Conventions such as the fact that the arrow associates to the right and the intersection operator binds tightest are standardly adopted. When $n = 2$, intersection is often written infix and when $n = 0$, so that the intersection is empty, the type is simply written as $\top$.

The restriction on intersections implied by the stratification of the grammar is only for convenience, and for developments in so far as higher-order model checking is concerned intersections could be allowed in tail position without any problems. In fact, if one views Kobayashi's types as a fragment of one of the larger intersection type languages in which intersection is allowed arbitrarily, such as that of Barendregt et al. [1983], then it is often the case that the associated subtype theory identifies all types of the form $\sigma_1 \to \tau_1 \wedge \tau_2$, which do not fit into this framework, with those of the form $(\sigma_1 \to \tau_1) \wedge (\sigma_1 \to \tau_2)$, which do.

The characterisation of the model checking problem for recursion schemes was given without reference to any subtype theory on the intersection types in [Kobayashi, 2009a] and [Kobayashi and Ong, 2009], though we should mention two points. First, in those type systems there is a kind of limited amount of subtyping "built in" which we shall point out when we relate the system shortly. Second, it quickly became clear that a subtype theory could be fruitfully exploited by decision procedures, such as Kobayashi's *hybrid* algorithm [Kobayashi, 2009b] where it is used to reduce the space of types to some smaller selection of canonical forms which stand as representatives of their subtype equivalents. This observation prompted this author and Ong to investigate the consequences of adding subtyping to the intersection type system for higher-order model checking, the results of which are reported as part of Chapter 3.

Intersection type assignment

Types are assigned to terms by way of a system of typing judgements which are derived according to certain rules. The system is very simple, consisting of only five rules corresponding to the five basic term formations, namely: variables, function symbols, constants, application and abstraction. There are no special rules to handle intersection itself, with the formalisation of this operator pushed into the rule for application.

An *intersection type environment*, Γ, is a finite, partial function mapping variables and function symbols to intersection types. An *intersection type judgement* is an expression of shape $\Gamma \vdash t : \tau$. Judgements are provable in the following system, where the notation $S|_i$

is used to denote the set $\{q \mid (i, q) \in S\}$. Recall that $S \models \phi$ asserts that S is a satisfying assignment to the positive Boolean formula ϕ.

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.\, t : \sigma \to \tau}$$

$$\frac{}{\Gamma, x : \bigwedge_{i=1}^{n} \tau_i \vdash x : \tau_i} \qquad \frac{}{\Gamma, F : \bigwedge_{i=1}^{n} \tau_i \vdash F : \tau_i}$$

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

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

Although the intersection operator is only formalised implicitly, in the rule for application, it is still powerful enough to assign a type to e.g. $\lambda x.\, x\, x$; let $q \in Q$:

$$\frac{\dfrac{}{x : (q \to q) \wedge q \vdash x : q \to q} \qquad \dfrac{}{x : (q \to q) \wedge q \vdash x : q}}{\dfrac{x : (q \to q) \wedge q \vdash x\, x : q}{\vdash \lambda x.\, x\, x : (q \to q) \wedge q \to q}}$$

As mentioned earlier, there is a kind of subtyping associated with the application rule because arguments are allowed to have strictly more types than are required by the functions that are being applied to them, e.g. with $\Gamma = \{x : q_1 \wedge q_2 \to q,\ y : q_1 \wedge q_2 \wedge q_3\}$:

$$\frac{\dfrac{}{\Gamma \vdash x : q_1 \wedge q_2 \to q} \qquad \dfrac{}{\Gamma \vdash y : q_1} \qquad \dfrac{}{\Gamma \vdash y : q_2}}{\Gamma \vdash x\, y : q}$$

However, this does not extend to the types *within* arrows. By this we mean that, for example, $x : (q_1 \wedge q_2 \to q) \to q,\ y : q_1 \to q \nvdash x\, y : q$, despite the fact that $q_1 \to q$ is, "morally", a stronger type than $q_1 \wedge q_2 \to q$ (since it requires less from its arguments). We will see this deficiency rectified in the type system presented in Chapter 3, which incorporates an explicit notion of subtyping.

The type system is induced by the property automaton, which features in the premise to the rule for constants. This rule acts to give a meaning to intersection types that can be thought of as follows. Each base type q is the type of all terms t that generate (via infinite reduction) trees that are accepted by the automaton from state q. An intersection, such as $q_1 \wedge q_2$, is the type of all terms that generate trees that are accepted *both* from state q_1 and from state q_2. Finally, an arrow such as $q_1 \wedge q_2 \to q$ is the type of those terms which, when applied to a term that generates a tree accepted from q_1 and q_2, will, as an application, generate a tree accepted from state q. For example, if $\delta(q, a) = (1, q_1) \wedge (1, q_2)$ for rank 1 constant a then any tree rooted at a will be accepted by $\mathcal{A}$ from state q just if the child of a is accepted from state q_1 and also accepted from state q_2. Thinking of trees as terms this means that the argument to a had better be accepted from state q_1 and from state q_2. This is reflected in the rule for constants, where $\vdash a : q_1 \wedge q_2 \to q$. Similarly, if $\delta(q, b) = (2, q_1) \vee (2, q_2)$ for rank 2 constant b, then since there are two possible ways of satisfying the transition function, there are two types assigned by the system, which are $\vdash b : \top \to q_1 \to q$ and $\vdash \top \to q_2 \to q$.

The kinding relation

An important observation made by Kobayashi was that the full power of intersection is not required for the purposes of higher-order model checking. We can make do with only those intersection types of a certain shape, and that shape should reflect the construction of the kinds.

The relation :: between intersection types and kinds is defined inductively by the following rules:

$$\frac{\sigma :: \kappa_1 \qquad \tau :: \kappa_2}{\sigma \to \tau :: \kappa_1 \to \kappa_2} \qquad\qquad \frac{}{q :: o} \qquad\qquad \frac{\tau_i :: \kappa \quad (\forall i \in [1..n])}{\bigwedge_{i=1}^{n} \tau_i :: \kappa}$$

Furthermore, the relation is extended to environments by writing $\Gamma :: \Delta$ for some intersection type environment Γ and some kind environment Δ just if, for all $F \in \mathsf{dom}(\Gamma)$, $\Gamma(F) :: \mathcal{N}(F)$.

Kobayashi's insight was that, when finding a type θ to assign to some term $t : \kappa$, it suffices to consider only those intersection types θ for which $\theta :: \kappa$. In other words those intersection types that *refine* κ, or conversely, those intersection types that are *well-kinded* by κ. Restricting the intersection types under consideration to only those which refine simple types removes a lot of power from the system. For example, it is now not possible to express the type $(q \to q) \wedge q \to q$ since the intersection $(q \to q) \wedge q$ underneath the arrow is not well kinded: $q \to q :: o \to o$ but $q :: o$.

Type environment consistency

In a recursion scheme, where terms are defined with respect to a given set of equations, type assignment should respect those equations. Since function symbols appear in the environment, the types assigned to them represent assumptions about the context. A typing $F : \sigma$ in an environment is an assumption about the behaviour of the function represented by F and so it should be justified by the equation that defines that function. In particular, it should not be so strong as to assert things about F that are not true. For example, if $\delta(a, q) = (1, q)$ for unary constant a and $\mathcal{R}(F) = \lambda x.\, a\ (F\ (a\ x))$ then to assign a type $\top \to q$ to F is unjustifiable, because the function $\lambda x.\, a\ (F\ (a\ x))$ cannot convert arbitrary inputs (those terms that satisfy all the types in the empty intersection $\top$) to trees accepted from state q, it can only convert those inputs that are themselves accepted from state q. We shall describe those type environments whose constituent type assignments are justified by the rules of $\mathcal{G}$ as being *consistent* with $\mathcal{G}$.

A type environment Γ is $(\mathcal{G}, \mathcal{A})$*-consistent*, just if $\Gamma :: \mathcal{N}$ and, for all $F \in \mathsf{dom}(\Gamma)$ and $\tau \in \Gamma(F)$, $\Gamma \vdash \mathcal{R}(F) : \tau$. By way of an example, consider the scheme of Example 2.1.24 and let $\mathcal{A}$ contain the transition function defined by:

$$\begin{array}{rcl} \delta(q_0, a) & = & (1, q_1) \wedge (2, q_0) \\ \delta(q_1, a) & = & (1, q_1) \wedge (2, q_1) \\ \delta(q_1, b) & = & (1, q_1) \\ \delta(q_1, c) & = & \mathsf{t} \end{array}$$

Then the environment $\Gamma = \{S : \top, F : (q_0 \wedge q_1 \to q_0) \wedge (q_1 \to q_1)\}$ is consistent since all the assignments can be justified; in other words, both of the following judgements are derivable:

$$\Gamma \vdash \lambda x.\, a\, x\, (F\, (b\, x)) : q_0 \wedge q_1 \to q_0$$
$$\Gamma \vdash \lambda x.\, a\, x\, (F\, (b\, x)) : q_1 \to q_1$$

However, the environment $\Gamma' = \{S : \top, F : q_1 \to q_0\}$ is not consistent, since the typing $F : q_1 \to q_0$ is not justifiable.

Note that there is a kind of circularity in the definition, since a typing $F : \tau$ can be used as part of its own justification. One consequence of this is that a function like Ω defined by the equation $\Omega = \Omega$ can be justifiably given any type at all, since $\Omega : \tau \vdash \Omega : \tau$ for any τ. Since, for ground kind Ω, $\mathsf{Tree}(\Omega) = \bot$, we might describe the situation by saying that it is consistent to assert that bottom inhabits every type. It is this kind of circularity that allows the system to type (terms that generate) *infinite* trees.

In the work of Kobayashi and Ong [2009], the consistency condition is generalised. A great insight of their work is that the parity condition of an alternating parity automaton (APT) can be captured in the type system by varying the notion of the consistency of environments. The automata generalise (co-trivial) automata, which can be seen as instances of APT in which the finitely many priorities number exactly one. However, since we are not concerned with automata with general parity conditions in this dissertation, let us just remark that the consistency condition corresponding to co-trivial acceptance requires every typing in the environment to be justified in just the same way as for trivial acceptance, but now there is an additional finiteness restriction on the sum of all the justifications. We shall see a formalisation of the condition in the next chapter.

Characterisation

Let us return to the characterisation theorem with which we started the section:

> *Fix a recursion scheme $\mathcal{G}$ and an alternating trivial automaton $\mathcal{A}$. Then $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ if, and only if, there exists a $(\mathcal{G}, \mathcal{A})$-consistent intersection type environment Γ such that $\Gamma \vdash S : q_0$.*

To summarise, the assertion $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ is equivalent to the existence of a type environment, consistent with the rules in $\mathcal{G}$ which moreover types S with q_0.

An immediate benefit of the characterisation theorem is that, through it, consistent type environments that type $S : q_0$ can be seen as certificates for yes-instances of the model checking problem. Given an instance $\langle \mathcal{G}, \mathcal{A} \rangle$ and a potential certificate Γ, the certificate can be verified simply by confirming that Γ is $(\mathcal{G}, \mathcal{A})$-consistent and $\Gamma \vdash S : q_0$, in other words: by type-checking. Consider the following example:

2.3.1 Example. *Consider the scheme from Example 2.1.6 and the automaton from Example 2.2.4. That the tree generated by the scheme is accepted by the automaton is witnessed by*

the type environment Γ*:*

$$S : q_0$$
$$F : (q_1 \to q_0) \wedge (q_3 \to q_0) \wedge (q_1 \to q_2)$$

by virtue of the provability of the typing judgements $\Gamma \vdash S : q_0$ *and, for consistency:*

$$\Gamma \vdash F\ c : q_0$$
$$\Gamma \vdash \lambda x.\, a\, x\, (F\, (b\, x)) : q_1 \to q_0$$
$$\Gamma \vdash \lambda x.\, a\, x\, (F\, (b\, x)) : q_3 \to q_0$$
$$\Gamma \vdash \lambda x.\, a\, x\, (F\, (b\, x)) : q_1 \to q_2$$

Computing consistent environments

To conclude this brief tour of the intersection type approach to recursion scheme model checking, let us point out that a major advantage of considering only those types that are well-kinded is that there are only finitely many. Given any simple type κ of order n and arity k, there are fewer than $\mathsf{exp}_n(k|Q|)$ intersection types that refine κ. Consequently, there is also only a finite number of intersection type environments that refine any given simple type environment. This means that it is possible to search for those environments that satisfy a (decidable) property in a finite amount of time. The search for a consistent type environment can be organised as follows.

Note that type environments can be ordered according to the typings that they contain. We write $\Gamma_1 \Subset \Gamma_2$ just if, for all $F \in \mathsf{dom}(\Gamma_1)$, $\tau \in \Gamma_1(F)$ implies $\tau \in \Gamma_2(F)$. Fix a recursion scheme. The collection of all type environments Γ with $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\mathcal{N})$ and $\Gamma :: \mathcal{N}$, when ordered as above, forms a complete lattice. Hence, the following monotone function Shrink from environments to environments has a greatest fixed point:

$$\mathsf{Shrink}(\Gamma)(F) = \bigwedge\{\tau \in \Gamma(F) \mid \Gamma \vdash \mathcal{R}(F) : \tau\}$$

Since, by definition, every fixed point of Shrink is a consistent type environment, the greatest fixed point $\nu\mathsf{Shrink}$ is the *largest consistent type environment.* As usual, it can be computed by repeatedly applying Shrink to the maximum type environment Γ_{max}, which assigns every intersection type refining $\mathcal{N}(F)$ to every $F \in \mathsf{dom}(\mathcal{N})$. Hence, there exists a $(\mathcal{G}, \mathcal{A})$-consistent type environment Γ with $\Gamma \vdash S : q_0$ iff $\nu\mathsf{Shrink} \vdash S : q_0$.

2.4 Algorithms for higher-order model checking

Due to the promise of interesting practical applications, researchers have been interested in developing efficient algorithms with which to solve recursion scheme model checking problems. Although the extremely high theoretical worst-case complexity seems like a barrier to any practical application, there are two points that should be considered. The first is that the applications to which model checking is to be applied typically belong to

the classes far worse than n-EXPTIME. For example, many applications in higher-order software verification can be phrased in terms of reachability problems for functional programs. Such problems are intrinsically Σ^0_1. In other words, *any* algorithm that attempts to solve, for example, the reachability problem for functional programs must also solve it for those *particularly simple* functional programs that are themselves recursion schemes and thus ultimately deal with the n-EXPTIME difficulty. The second point is that, in practice, the kinds of higher-order programs that people wish to verify have rather small, human-bearable maximum arity and order and, when we consider these parameters to be fixed, the problem is linear in the size of scheme. Hence, many authors have tried to invent or improve on efficient model checking algorithms; we survey them in this section.

Algorithms from proofs. The first higher-order model checking algorithm was Ong's witness to the decidability of the alternating parity tree automaton (APT) model checking problem for recursion schemes [Ong, 2006]. Ong's algorithm reduces the problem of deciding whether the value tree of the scheme is accepted by the property automaton to one of whether a certain regular tree, called the *computation tree* of the scheme, is accepted from a certain APT, called the *traversal simulating APT*. The computation tree of the scheme is the value tree of an order-0, defunctionalisation of the scheme. This tree makes explicit all of the intensional structure of the computation, laying out all the beta-redexes that contribute to it without actually contracting any of them. The *traversals* on the computation tree recover the dynamics of the computation, they can be seen as an evaluation of the scheme under linear-head reduction. The traversal simulating APT is constructed so as to accept the computation tree iff the property APT would accept the value tree and hence reduce the problem to a finite parity game, which can be solved by an algorithm such as that of Jurdziński [2000]. In joint work with Ong, Broadbent later extended the algorithm to solve the so called *global* version of the model checking problem, which requires the construction of a representation of the value tree at which the property holds [Broadbent and Ong, 2009]. Due to the complexity of some of the transformations and its reliance on game semantics, the algorithm is considered difficult to understand. Furthermore, it is unlikely to be practical, since the parity game constructed in this way has a number of nodes which is n-exponential in the order of the scheme.

Kobayashi's greatest fixpoint computation from [Kobayashi, 2009a] is a much simpler algorithm for the case when the property is specified by a trivial automaton. This algorithm was introduced as part of the intersection type characterisation of the trivial automaton fragment of the model checking problem. The computation is exactly the calculation of $\nu\mathsf{Shrink}$ as described in the previous section. Unfortunately, since computing the greatest fixed point of Shrink requires constructing the maximum type environment Γ_{max} whose cardinality is $\mathcal{O}(\mathsf{exp}_n(c))$ (with n the order), this algorithm also suffers from an n-exponential running time for *all* order-n inputs. The algorithm arising from the joint work of Kobayashi and Ong [2009] in lifting the type based characterisation to all mu-calculus definable properties generalises this greatest fixpoint computation to the solution of a parity game. However, the parity game is played over a graph containing every type environment and so the all-instances hyper-exponential running time is not avoided.

The hybrid algorithm. The three algorithms that we have discussed so far were designed to be witnesses to decidability results rather than efficient procedures suitable for model checking practice. Indeed, to the best of our knowledge, none of the previous three algorithms has ever been implemented. The first algorithm that was developed with the explicit goal of being implemented and used was the so-called *hybrid* algorithm of Kobayashi [2009b]. The hybrid algorithm is a decision procedure for the deterministic trivial automaton model checking problem. It attempts to address the specific problem of how to compute a consistent type environment without having to construct the maximum one Γ_{max}. The algorithm is based on the thesis that, in practice, the sort of recursion schemes that people are interested in checking will behave in a uniform manner and thus there will exist consistent environments that are much smaller than Γ_{max}. The idea is, given a recursion scheme $\mathcal{G}$ and a deterministic trivial automaton $\mathcal{A}$, to infer types from a finite, partial evaluation of the scheme which will, in a sense, reveal how it really behaves.

The algorithm is structured as a loop consisting of three stages. In the first stage the *configuration graph* is constructed. This graph is a finite prefix of a kind of product construction between the reduction relation of the scheme and the transition function of the automaton. The size of the finite prefix constructed can be considered as a parameter to the algorithm. The graph can be thought of as a partial evaluation of the scheme in which the ground-kind terms t (reducts of S) are annotated by the state of the automaton in which $\mathsf{Tree}(t)$ is required to be accepted. By keeping track of this state information, it is possible to identify counter-examples, which occur when the partial evaluation reveals a pair $(a\ s_1\ \cdots\ s_n,\ q)$ for which $\delta(q,\ a) = \mathsf{f}$. If a counter-example is detected in this way, then the algorithm terminates and the input is a no-instance. Otherwise, in the second stage, types are extracted from the configuration graph following a procedure adapted from [Kobayashi and Ong, 2009]. Since the configuration graph is only a prefix of the product, the types that are inferred from it are often incomplete and so the algorithm employs a guessing strategy in order to try to complete them; the result is a candidate type environment Γ. However, these guesses may be incorrect and so in the third stage, types that are not consistent are filtered out by computing the greatest fixed point Γ' of Shrink *below* Γ. Γ' is guaranteed to be a consistent type environment but it may not type S with q_0. In case $\Gamma' \vdash S : q_0$ then the algorithm terminates and the input is a yes-instance. Furthermore, the environment Γ' acts a certificate. Otherwise, the algorithm returns to the first stage, extends the configuration graph by a further finite amount and repeats the process.

The hybrid algorithm overcomes the main obstacle to usability inherent in the previous algorithms by avoiding the n-exponential bottleneck in at least some cases (hopefully in all those cases that are interesting). Of course, the algorithm is still hyper-exponential in the worst case. An implementation, TRecS, provided some evidence of the performance on typical inputs and the results were very surprising. Despite the extreme worst-case complexity, TRecS could solve order-3, order-4 and order-5 instances consisting of a few tens of lines in only a few milliseconds. This alone set the algorithm apart from those previously discussed, for which this sort of result would be impossible. TRecS has been a very successful higher-order model checker, and until only recently its strong performance has made it the most popular choice for applications in verification.

The GTRecS algorithm. Despite its surprisingly good results on very small but high-order examples, the hybrid algorithm was felt unsatisfactory because it did not achieve the theoretical lower-bound on worst-case time complexity. In particular, it was not linear in the size of the recursion scheme and so could not be expected to scale reliably beyond these small inputs. The GTRecS algorithm was Kobayashi's response to these problems [Kobayashi, 2011], it is a decision procedure for the trivial automaton model checking problem. The algorithm achieves a linear worst-case time complexity with respect to the size of the scheme by working exclusively with intersection types, no potentially large auxiliary structures are constructed (such as the configuration graph). As in the hybrid algorithm, the idea is to compute a seed Γ with which to relativise the greatest fixed point computation of Shrink which is (hopefully) much less expensive than the canonical choice Γ_{max}.

The algorithm takes the form of a sequential composition of two fixed point computations. The first fixed point computation constructs a seed Γ and the second constructs the greatest fixed point of Shrink below Γ as before. The first computation calculates the least fixed point of an increasing function of type environments, starting with the environment $\{S : q_0\}$. The idea is that from a typing $F : \tau$ the algorithm adds new typings to the environment based on an inspection of the right hand side of F. The new typings may not be consistent and this motivates the second fixed point computation. The new types are guessed according to an intuition from game semantics that is made formal by a relation between types (and between environments).

The algorithm was implemented in a tool called GTRecS. An empirical evaluation demonstrated that whilst GTRecS did outperform TRecS on a selection of new examples (of roughly the same size), TRecS was considerably more efficient at processing many others, including all the examples that had been used as the benchmarks in [Kobayashi, 2009b].

The TravMC algorithm. The TravMC algorithm of Neatherway, Ong, and Ramsay [2012] was an attempt to develop an efficient decision procedure for the deterministic trivial automaton model checking problem for recursion schemes with cases. Recursion schemes with cases, which we reviewed briefly at the end of Section 2.1, extend the syntax of recursion schemes with a case analysis construct over a finite enumeration. Of course, every recursion scheme is itself a recursion scheme with cases, so the algorithm can also be viewed as a decision procedure for the more restricted problem.

Like the previous two algorithms, the TravMC algorithm also follows the intersection type approach, but the extension by cases requires a complimentary extension to the underlying type system. This consisted of a restricted form of union types, in which type unions can be formed only at ground kind and eliminated only by use of the case construct. The overall structure of the procedure is quite similar to the hybrid algorithm: there is a main loop in which a finite structure is extended, types are extracted from this structure and then a test determines termination. However, there are a number of important differences. The most important is that the TravMC algorithm attempts a more direct construction of a type environment by building finite prefixes of the consistency

proof itself rather than finite prefixes of the configuration graph. The way that the finite prefixes are extended is by following a projection of the traversals, first seen in Ong's decidability proof [Ong, 2006], on the typing derivation. The central consequence of this approach is that, unlike type extraction in the hybrid algorithm, the types extracted from these prefixes of derivations are guaranteed to be consistent.

The algorithm was implemented in the tool TravMC. Since the algorithm is designed to check recursion schemes with cases, much of the empirical evaluation focused on problem instances within that class. However, a direct comparison was made on a small number of vanilla recursion schemes, which showed that, whilst TravMC was generally more efficient than GTRecS, it was not quite able to match TRecS.

The C-SHORe algorithm. Following a completely different approach, the C-SHORe algorithm of Broadbent, Carayol, Hague, and Serre [2012] solves the alternating co-trivial automaton model checking problem for recursion schemes via an equivalent presentation using collapsible pushdown automata. Furthermore, whereas the type based algorithms that we have discussed so far have been *forward* algorithms, in the sense of starting from the initial configuration of the system and inferring the consequences, the C-SHORe algorithm is a *backward* algorithm, in the sense of starting from a set of error configurations and inferring the necessary causes. Like the GTRecS algorithm, this procedure is fixed-parameter tractable, running in polynomial time in the size of the scheme even in the worst case.

The algorithm is in the classical, saturation style that was introduced by Bouajjani, Esparza, and Maler [1997] for pushdown automata. In the case of pushdown automata, the set of configurations that can reach some regular set can be recognized by a finite state automaton. In the case of an order-n collapsible, higher-order pushdown automata, this role is played by what the authors call an order-n stack automaton. The idea is to start from such an automaton characterising the set of error configurations, which represent property violations, and then *saturate* the automaton by successively adding more and more states and transitions until a fixed point is obtained. This fixed point stack automaton is the characterisation of the set of all configurations that can reach an error configuration.

The authors implemented a variation of the algorithm in a tool, C-SHORe [Broadbent et al., 2013]. A crucial part of their implementation was to add a preliminary, cheap control flow analysis which helps to prune away a large number of irrelevant configurations from the search space — without this step their tool times out on all but one of the benchmarks. The results of a comparison with the tools already mentioned showed mixed results. Whilst C-SHORe ran fastest on five of the twenty-six benchmarks, it ran slowest on six. TRecS ran quickest on the majority of examples. However, when compared in isolation with GTRecS, the only other fixed-parameter tractable algorithm, C-SHORe was generally superior. Of course, it should be noted that the C-SHORe algorithm is designed for collapsible pushdown automata and a comparison of its performance on recursion scheme benchmarks (which must first be translated into automata suited to its input) should be taken with a pinch of salt.

The HorSat algorithm. The HorSat algorithm of Broadbent and Kobayashi is an attempt to transfer the backwards, saturation approach of the C-SHORe algorithm from its setting of collapsible, higher-order pushdown automata and order-n stacks to the setting of higher-order recursion schemes and intersection types. The HorSat algorithm solves the alternating co-trivial automaton model checking problem and its partner algorithm HorSatT solves the alternating trivial automaton model checking problem. Like the GTRecS and C-SHORe algorithms, the HorSat algorithm is fixed-parameter polynomial time in the worst case.

The idea of the algorithm is, in spirit, the same as that of the C-SHORe algorithm. However, rather than representing a set of stack configurations using an order-n stack automaton, the HorSat algorithm represents a set of terms using a type environment. The algorithm is structured as a single least fixed point computation. Starting from an environment that types all those terms that represent violations of the property — i.e. those terms t for which $t^{\perp}$ is rejected by the automaton — the environment is iteratively expanded according to a certain monotone function. The function adds a new typing $F : \sigma_1 \to \cdots \to \sigma_n \to q$ to the environment just if the typing is consistent and there exist closed terms that inhabit each of the σ_i. The algorithm, like ours presented in Chapter 4, is (independently) novel in its use of types to reason about rejection.

The co-trivial version of the algorithm is implemented in the tool HorSat and the trivial version in the tool HorSatT, though since for each alternating trivial automaton model checking instance there is an equivalent alternating co-trivial automaton model checking instance, both tools can be used for both kinds of problem. The results of an evaluation of the tools on a selection of benchmarks are very good with at least one of the two new algorithms proving to be the fastest of all the algorithms that we have discussed so far. In particular, on some of the slightly larger benchmarks, typically of a few hundred lines or so, HorSat outperforms TRecS convincingly.

2.5 Applications in verification

A large part of the interest in recursion scheme model checking arises from the promise of highly accurate analysis and verification tools for higher-order programs that can be built on top of it. Despite the fact that robustly efficient algorithms for the decision problem are only just emerging, there have already been several proposals that describe ways in which concrete verification problems can be reduced to recursion scheme model checking.

The first such suggestion came from Kobayashi, Tabuchi, and Unno [2010]. The idea of the work is to model a program as a kind of tree transducer, and they introduce a model of programs called higher-order multi-parameter tree transducers (HMTT) for this purpose. One can think of an HMTT as a first order function over trees which is itself defined in terms of a number of interacting higher-order functions. The verification problem they study is, given a program represented as an HMTT, a regular representation of the allowed set of inputs to the program and a regular representation of the set of allowable outputs, does the program satisfy the specification? Their method is to reduce the problem to RSFD model checking (recall that RSFD are a variant of recursion schemes that we discussed at the end of Section 2.1), by building an RSFD whose finite data domain is an abstraction constructed from the regular specification. They show how it is possible, in principle, to solve many interesting problems in areas such as XML-translation and string analysis, as well as certain problems in program verification more generally. The common thread is that programs from these fields can be viewed naturally as first-order functions over trees, but in practice are often defined in terms of mutually recursive systems of higher-order functions. Furthermore, they present some promising results based on preliminary experiments and the method is extended in later work reported in [Unno et al., 2010].

Following the observation that recursion schemes are, in a precise sense, the higher-order analogues of Boolean programs, Kobayashi, Sato, and Unno [2011] presented an approach to program verification based on predicate abstraction and higher-order model checking. Their approach is a very natural lifting of the first-order Boolean programs framework of the Microsoft SLAM project [Ball and Rajamani, 2002], which has been highly successful for the analysis of device drivers written in C. The construction of a higher-order Boolean program (represented as a recursion scheme) from a given functional program is via an assignment of dependent types, built over the atomic predicates, which acts to specify the Boolean transfer functions involved. The combination of functional program and dependent typing are together converted to a recursion scheme by a type-directed transformation. As usual, if this abstract model satisfies a certain reachability property then the original program will be known to satisfy the property also. If the abstract model is found to violate the property, new predicates can be inferred from the counterexample through a dependent type assignment algorithm. The potential of the work was shown through a large number of small but complicated examples that can be accurately analysed. Scalability of the method remains an open problem which is actively under research; some recent progress is reported in [Sato et al., 2013].

The great precision that can be achieved through building analyses upon higher-order model checkers is demonstrated to good effect by Tobita, Tsukada, and Kobayashi [2012] in

their exact flow analysis by model checking. They study the control flow analysis problem for a call-by-value, simply typed lambda calculus with recursion and non-determinism, and show that it is possible to construct an analyser that is completely accurate by reducing the problem to the trivial automaton model checking problem for recursion schemes. They are not the first to propose an exact analysis for a language of this kind, since Mossin [2003] had also developed an algorithm, but since Mossin's algorithm has an enormous constant factor, it is considered impractical even on very small examples. In contrast, by reducing to higher-order model checking, Tobita *et al*'s work is able to leverage recent advances there, and report encouraging experimental results.

Finally, we look at our own analysis of functional programs with pattern matching and algebraic data types, which was joint work with Ong reported in [Ong and Ramsay, 2011]. This paper introduced PMRS as a model of such functional programs and reduced a certain verification problem concerning them to the model checking problem of their weak cousin, wPMRS (recall the variants discussed at the end of Section 2.1). To give the reader a bit more of a flavour of how recursion scheme model checking can be used in order to help solve verification problems, we look at this example in more detail.

In [Ong and Ramsay, 2011], we studied the following verification problem. Given a correctness property ϕ, a functional program $\mathcal{P}$ (*qua* deterministic PMRS) and a regular set $\mathcal{I}$ of input terms, does every term that is reachable from $\mathcal{I}$ under rewriting by $\mathcal{P}$ satisfy ϕ? It is straightforward to see that the problem is undecidable.

By way of an example, consider the PMRS $\mathcal{P}$ which, when started from *Main*, takes as input a list of natural numbers and returns the same list with all occurrences of the number zero removed. The defining rules of $\mathcal{P}$ are given by:

$$
\begin{array}{lll}
\mathit{Main}\,m & = & \mathit{Filter}\,\mathit{Nz}\,m \\[1em]
\mathit{If}\,a\,b\,\mathtt{true} & = & a \\
\mathit{If}\,a\,b\,\mathtt{false} & = & b \\[1em]
\mathit{Nz}\,\mathtt{z} & = & \mathtt{false} \\
\mathit{Nz}\,(\mathtt{s}\,n) & = & \mathtt{true} \\[1em]
\mathit{Filter}\,p\,\mathtt{nil} & = & \mathtt{nil} \\
\mathit{Filter}\,p\,(\mathtt{cons}\,x\,xs) & = & \\
\quad \mathit{If}\,(\mathtt{cons}\,x\,(\mathit{Filter}\,p\,xs))\,(\mathit{Filter}\,p\,xs)\,(p\,x) & &
\end{array}
$$

The input set $\mathcal{I}$ is given by a regular tree grammar which simply specifies all possible lists of natural numbers. The correctness property ϕ is: "any outcome of the program is a list containing no zeros", which is easily expressible as a trivial automaton.

Our work presents a sound but incomplete semi-algorithm for solving the problem, which is based on a counterexample-guided abstraction refinement (CEGAR) loop [Kurshan, 1994, Clarke et al., 2000]. The input to the algorithm consists of a PMRS $\mathcal{P}$ representing the program, a regular tree grammar $\mathcal{G}$ representing the set $\mathcal{I}$ of possible

inputs to the program and a trivial tree automaton $\mathcal{A}$ representing a specification ϕ of good behaviour. There are four steps to the CEGAR loop, which we describe as follows.

In the first step we compute a sound abstraction of the behaviour of $\mathcal{P}$ when started from terms in $\mathcal{I}$. From an order-n PMRS $\mathcal{P}$ and a regular tree grammar $\mathcal{G}$, we build an order-n *weak pattern-matching recursion scheme* (wPMRS) which over-approximates the set of terms that are reachable from $\mathcal{I}$ under rewriting by $\mathcal{P}$. Recall from Section 2.1 that a wPMRS is similar to a PMRS, except that its pattern matching mechanism is only able to determine control flow; it is unable to decompose data structure.

Our method is to construct a finite set Ξ of variable-term bindings such that, for every variable x (formal parameter of rewrite rule), every term that is ever bound to x during the computation is derivable from Ξ. In other words, Ξ is a finite representation of, for each formal parameter x, an over-approximation to the set of terms that are bound to x during any run of the program. In the second stage, we use the set Ξ to build rules for the over-approximating wPMRS. These rules model the bindings of all non-pattern-matching (including all higher-order) variables *precisely*; they only approximate the binding behaviours of the pattern-matching variables. This is in contrast to typical control flow analyses for functional programs, such as that of Jones and Andersen [2007] (our inspiration for this method), which builds a regular tree grammar that over-approximates the binding set of *every* variable. For an order-n PMRS, our algorithm produces an order-n wPMRS $\widetilde{\mathcal{P}_{\mathcal{G}}}$ as an abstraction, which is a tighter approximation of the order-n PMRS being analysed than regular tree grammars (which are equivalent to order-0 wPMRS).

The weakened pattern-matching mechanism of wPMRS makes it possible to decide a model checking problem for it, which is the content of the second step. Given a wPMRS $\mathcal{W}$, a closed term t and a trivial automaton $\mathcal{A}$, we decide if every (possibly infinite) tree generated by $\mathcal{W}$ on input t is accepted by $\mathcal{A}$.

If the model-checker does not find any violation of the property then, since the approximating wPMRS $\widetilde{\mathcal{P}_{\mathcal{G}}}$ defines a superset of the terms reachable under $\mathcal{P}$ from $\mathcal{I}$, the CEGAR loop will terminate because $\mathcal{P}$ satisfies $\mathcal{A}$ on $\mathcal{I}$. However, if the model-checker reports a counterexample, then it may be that $\mathcal{P}$ also violates the property (for some term in $\mathcal{I}$), but it may also be that the counterexample is an artefact of an inaccuracy in the abstraction. To determine which of these possibilities is the case, in the third step we analyse the non-determinism introduced in the abstraction to see whether it behaves well for this particular counter-example.

In the fourth step the abstraction process is refined. Due to the fact that the abstractions only ever approximate the (first-order) pattern matching variables, whilst remaining faithful to all the others, there is a simple notion of automatic abstraction-refinement, whereby patterns are "unfolded" to a certain depth in the PMRS $\mathcal{P}$, forming a new PMRS $\mathcal{P}'$. In the abstraction $\widetilde{\mathcal{P}'_{\mathcal{G}}}$ of $\mathcal{P}'$, the rules that define the approximation will be more accurate and, in particular, the spurious counterexample will no longer be present. Since any rule in a wPMRS abstraction $\widetilde{\mathcal{P}'_{\mathcal{G}}}$ is *perfectly* accurate whenever the pattern parameter contains no free variables, this method of unfolding gives rise to a semi-completeness property. Given any no-instance of the PMRS verification problem, the CEGAR loop eventually terminates with the answer "No".

Returning to our earlier example, whilst performing the first step we obtain an over-approximation of the binding behaviour of the variables in the program Ξ. This set contains, amongst others, the bindings: $x \mapsto N$ and $xs \mapsto ListN$. From this we construct an approximating wPMRS $\widetilde{\mathcal{P}_{\mathcal{G}}}$, whose rule-set contains the following:

$$
\begin{aligned}
& \mathit{Filter}\, p\, \mathtt{nil} = \quad \mathit{Nil} \\
& \mathit{Filter}\, p\, (\mathtt{cons}\, x\, xs) = \\
& \quad \mathit{If}\ (\mathit{Cons}\ X\ (\mathit{Filter}\, p\, XS))\ (\mathit{Filter}\, p\, XS)\ (p\, X)
\end{aligned}
$$

$$
\begin{aligned}
X &= \mathtt{z} \\
X &= \mathtt{s}\, X \\
XS &= \mathtt{nil} \\
XS &= \mathtt{cons}\, X\, XS
\end{aligned}
$$

together with, amongst others, all the $\mathcal{P}$ rules from the example except those for *Filter*. Unfortunately the wPMRS is too coarse to be useful: there are trees (representing lists) that are obtained by rewriting from '*MainS*' that are not accepted by the trivial automaton $\mathcal{A}$. However, these are spurious counterexamples. For an illustration, consider the error trace in the wPMRS:

$$
\begin{aligned}
& \mathit{Main}\ S \\
\rhd^* \ & \mathit{Main}\, (\mathtt{cons}\, (\mathtt{s}\, \mathtt{z})\, \mathtt{nil}) \\
\rhd^* \ & \mathit{Filter}\ \mathit{Nz}\, (\mathtt{cons}\, (\mathtt{s}\, \mathtt{z})\, \mathtt{nil}) \\
\rhd \ & \mathit{If}\ (\mathit{Cons}\ X\ (\mathit{Filter}\ \mathit{Nz}\ XS))\ (\mathit{Filter}\ \mathit{Nz}\ XS)\ (\mathit{Nz}\ X) \\
\rhd^* \ & \mathit{If}\ (\mathtt{cons}\, \mathtt{z}\, (\mathit{Filter}\ \mathit{Nz}\, \mathtt{nil}))\ (\mathit{Filter}\ \mathit{Nz}\ XS)\ (\mathit{Nz}\, (\mathtt{s}\, \mathtt{z})) \\
\rhd^* \ & \mathtt{cons}\, \mathtt{z}\, (\mathit{Filter}\ \mathit{Nz}\, \mathtt{nil}) \\
\rhd^* \ & \mathtt{cons}\, \mathtt{z}\, \mathtt{nil}
\end{aligned}
$$

The problem can be traced to the second clause of *Filter* in the wPMRS: when replacing the *variable* x by the *non-terminal* X, the connection between the two occurrences of x in the right-hand side is lost, as the reduction of one occurrence of X is independent of that of the other.

The refinement algorithm produces a new, unfolded PMRS $\mathcal{P}'$ that replaces the two defining rules of *Filter* by five new rules. The two rules that cover the case when the list

is a singleton are shown below:

$$\begin{aligned}
&\mathit{Filter}\ p\ (\texttt{cons}\ \texttt{z}\ \texttt{nil}) = \\
&\qquad \mathit{If}\ (\texttt{cons}\ \texttt{z}\ (\mathit{Filter}\ p\ \texttt{nil}))\ (\mathit{Filter}\ p\ \texttt{nil})\ (p\ \texttt{z}) \\
&\mathit{Filter}\ p\ (\texttt{cons}\ (\texttt{s}\ v_2)\ \texttt{nil}) = \\
&\qquad \mathit{If}\ (\texttt{cons}\ (\texttt{s}\ v_2)\ (\mathit{Filter}\ p\ \texttt{nil}))\ (\mathit{Filter}\ p\ \texttt{nil})\ (p\ (\texttt{s}\ v_2))
\end{aligned}$$

Applying the approximation algorithm to PMRS $\mathcal{P}'$ (and input grammar $\mathcal{G}$), we obtain a wPMRS $\widetilde{\mathcal{P}'_{\mathcal{G}}}$ that accurately captures the set of reachable terms. We showed that this algorithm is *semi-complete*: whenever its input is a no-instance of the verification problem, this method is guaranteed to terminate and identify it as such.

Chapter 3

An Intersection Refinement Type System with Subtyping

In this chapter we introduce an extension of Kobayashi's intersection type system for characterising the trivial automaton model checking problem for recursion schemes [Kobayashi, 2009a] that we surveyed in Chapter 2. The extension consists of integrating a carefully restricted form of subtyping into the system. The work, which we carried out together with Ong, was a response to Kobayashi's proposal in [Kobayashi, 2009b] to use subtyping as a kind of symmetry breaking optimisation in the search for type assignments which forms part of his *hybrid* algorithm. In particular, we were interested in using subtyping to allow more liberal type derivations without losing (i) the preservation of kind refinement and (ii) an efficient type checking algorithm. In the following we will present the intersection types, their subtype theories and the associated type assignment system. We show that the system meets our specification by preserving refinement and admitting an efficient type checking algorithm, and we show that, despite the greater permissiveness of the system, it still characterises the (co-)trivial automaton model checking problem.

This system is the basis for the work in Chapters 4 and 5. In Chapter 4, the argument for the correctness for our decision procedure depends upon the additional freedom granted by the subtyping extension, see e.g. Lemmas 4.3.13 and 4.3.18. In Chapter 5, subtyping is used to formalise the underlying precision of abstract domains.

3.1 Intersection types

We follow Kobayashi and also the work of van Bakel [2011], from which we take the name "strict", in stratifying our types into those with an outermost intersection and those without.

3.1.1 Definition (Intersection types)**.** Fix a set of atomic *base types* $(q \in)\ Q$. The *intersection types* over Q are defined simultaneously with the *strict types* over Q by the following grammar:

$$\begin{array}{lrcl} (\textsc{Strict Types}) & \tau & ::= & q \mid \sigma \to \tau \\ (\textsc{Intersection Types}) & \sigma & ::= & \bigwedge_{i=1}^{n} \tau_i \end{array}$$

in which $q \in Q$ and $n \geqslant 0$. We will use parentheses to disambiguate such expressions and assert that arrows associate to the right and that intersection binds tightest. An *intersection type environment*, Γ, is a finite, partial function mapping variables and function symbol to intersection types. We will write $\Gamma_1 \uplus \Gamma_2$ for the operation sometimes called *type environment multiplication*, which is just the pointwise combination of environments defined by:

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

and write $\Gamma_1 \Subset \Gamma_2$ just if there is some Γ' and $\Gamma_1 \uplus \Gamma' = \Gamma_2$. We will use τ and σ to denote strict and intersection types respectively. When we are agnostic about whether a particular expression is either a strict type or an intersection type, we will say it is simply a *type* and denote it by θ.

Notation Since intersection types play a major role in this work, we will use a number of notational conventions (abuses) to make working with them as intuitive as possible. We will typically write $\top$ for the empty intersection $\bigwedge \varnothing$, an intersection $\bigwedge_{i=1}^{2} \tau_i$ containing two elements infix as $\tau_1 \wedge \tau_2$ and an intersection $\bigwedge\{\tau\}$ of the singleton set containing τ simply as τ. Whenever convenient, we will identify an intersection type $\sigma = \bigwedge_{i=1}^{n} \tau_i$ with the set of its conjuncts $\{\tau_i\}_{i \in [1..n]}$ and, for example, write $\tau \in \sigma$ just in case there is some $i \in [1..n]$ such that $\tau = \tau_i$. Although the intersection operator is defined only for strict types, this shall not deter us from writing types such as $\bigwedge X$ for arbitrary set of types X; by which is meant $\bigwedge\{\tau \mid \tau \in X \text{ or there is some } \sigma \in X \text{ and } \tau \in \sigma\}$.

Intersection type theories, which formalise the subtype relation, have been well studied in the literature. In particular, intersection type theories have been very successful as a logical approach to studying various denotational models of lambda calculi [Coppo et al., 1984]. Since intersection types are those expressions freely generated from some atoms by means of the intersection operator and the arrow, much of the interesting expressive power of the typing discipline comes from defining interesting subtype relationships over specially selected sets of atoms. Since there is now a comprehensive guide to the literature in this area [Barendregt et al., 2013], let us just remark on one prototypical example so as to give the reader an idea of the general flavour. Coppo, Dezani-Ciancaglini, and Zacchi [1987] were able to classify terms by their normalisation properties using a system built over two atoms: ϕ_* representing the set of all terms that have a normal form, and $\phi_\top$ representing the "normalisability preserving" arguments, i.e. those terms t such that $s\,t$ is normalisable whenever s is. This can be formalised using a subtype theory which sets $\phi_\top \leqslant \phi_*$ and makes equivalent ϕ_* and $\phi_\top \to \phi_*$ and also $\phi_\top$ and $\phi_* \to \phi_\top$. Briefly, these axioms assert that normalisability preserving arguments are necessarily normalisable, that normalisable

terms, when applied to normalisability preserving arguments deliver normalisable results and that normalisability preserving arguments, when applied to normalisable arguments are again normalisability preserving.

We shall not be so liberal as this in our type theory, since we will not allow the relating of atoms and compound constructions (such as arrow types). The theory that we adopt here is quite close to the classical theory of Barendregt et al. [1983]. Just as the intersection types themselves are generated from a set of atomic base types, so the intersection type theory is generated from a preordering of the base types.

3.1.2 Definition (Intersection type theory)**.** An *intersection type theory* is a set of *base types* Q equipped with a preorder Θ. The *subtype relation* over an intersection type theory Q, written $\leqslant$, is the least relation on types that validates the following rules:

(Q-Bas) if $(q_1, q_2) \in \Theta$ then $q_1 \leqslant q_2$

(Q-Arr) if $\sigma_2 \leqslant \sigma_1$ and $\tau_1 \leqslant \tau_2$ then $\sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2$

(Q-Fun) if $\bigwedge_{i=1}^n \tau_i \leqslant \tau$ then $\bigwedge_{i=1}^n (\sigma \to \tau_i) \leqslant \sigma \to \tau$

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

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

(Q-Trs) if $\theta_1 \leqslant \theta_2$ and $\theta_2 \leqslant \theta_3$ then $\theta_1 \leqslant \theta_3$

When two types θ_1 and θ_2 are mutual subtypes, i.e. $\theta_1 \leqslant \theta_2$ and $\theta_2 \leqslant \theta_1$, we say they are *subtype equivalent* and write $\theta_1 \equiv \theta_2$. We extend the ordering to type environments , writing $\Gamma_1 \leqslant \Gamma_2$ just if, for all $\xi \in \mathsf{dom}(\Gamma_2)$, $\Gamma_1(\xi) \leqslant \Gamma_2(\xi)$.

When $\theta_1 \leqslant \theta_2$ we will often say that θ_1 is *stronger* than θ_2. The rule (Q-Bas) imports the underlying preordering of the atoms. The rule (Q-Arr) is the usual formalisation of subtyping with respect to the arrow: a type $\sigma_1 \to \tau_1$ is stronger than a type $\sigma_2 \to \tau_2$ just if σ_2 is a stronger requirement on the argument than σ_1 and τ_1 delivers a stronger guarantee about the result than τ_2. The rule (Q-Fun) reflects the intended interpretation of the arrow as a set of functions. If a function has each of an intersection of types $\bigwedge_{i=1}^n (\sigma \to \tau_i)$ then, given an argument inhabiting σ the result will inhabit $\bigwedge_{i=1}^n \tau_i$. Consequently, if it is also known that $\bigwedge_{i=1}^n \tau_i$ is stronger than some type τ then also $\bigwedge_{i=1}^n (\sigma \to \tau_i)$ is stronger than $\sigma \to \tau$. The rules (Q-Prj) and (Q-Glb) formalise the role of intersection as a meet (modulo subtype equivalence). Finally the rule (Q-Trs) simply expresses the transitivity of the preorder (note that reflexivity follows from the reflexivity of Θ).

Observe that, due to the way we have generated the subtype relation from the preorder on the base types, we have not forced meets at base type to coincide with the intersections that were added. This could be done without any serious difficulty, but we prefer to have stronger inversion theorems, which we shall see next. These theorems come in very useful since they allow to deduce the conditions that necessarily hold whenever it is known that two intersection types are in the subtype relation. The first one concerns the particular situation at base type.

3.1.3 Lemma (Inversion at base types)**.** *$\bigwedge_{i=1}^{n} q_i \leqslant \bigwedge_{j=1}^{m} q'_j$ iff for all $j \in [1..m]$, there is some $i \in [1..m]$ such that $q_i \leqslant q'_j$.*

Proof. In the forward direction we proceed by induction on the derivation of the premise, note that (Q-Arr) and (Q-Fun) are not applicable.

(Q-Bas) In this case $n = m = 1$ and the result is trivial.

(Q-Prj) In this case $m = 1$ and there is some $i \in [1..n]$ such that $q_i = q'_1$. The result is then clear.

(Q-Glb) In this case the immediate premises have $\bigwedge_{i=1}^{n} q_i \leqslant q'_j$ for all $j \in [1..m]$. Therefore the result follows from the induction hypothesis.

(Q-Trs) In this case there is some intermediate type of the form $\bigwedge_{k=1}^{p} q''_k$. It follows from the induction hypothesis that for each $j \in [1..m]$ there is some $k \in [1..p]$ such that $q''_k \leqslant q'_j$ and by the induction hypothesis a second time there is some $i \in [1..n]$ such that $q_i \leqslant q''_k$. Hence the result follows from the transitivity of subtyping.

In the backward direction we use (Q-Glb) so let $j \in [1..m]$, then by the assumption there is some $i \in [1..n]$ such that $q_i \leqslant q'_j$ and the result follows from (Q-Prj). □

The second theorem concerns the situation at arrow types. Here the conditions are slightly weaker than at base type because of the influence of the rule (Q-Fun).

3.1.4 Lemma (Inversion at arrow types)**.** *$\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i) \leqslant \bigwedge_{j=1}^{m}(\sigma'_j \to \tau'_j)$ iff for all $j \in [1..m]$, there is some $N \subseteq [1..n]$ such that $\forall i \in N.\, \sigma'_j \leqslant \sigma_i$ and $\bigwedge_{i \in N} \tau_i \leqslant \tau'_j$.*

Proof. The forward direction is by induction on the derivation of the assumption.

(Q-Bas) Not applicable.

(Q-Arr) In this case, necessarily $n = m = 1$ and the result follows trivially.

(Q-Fun) In this case, necessarily $m = 1$ and for all $i, j \in 1..n$, $\sigma_i = \sigma_j = \sigma'$ and $\bigwedge_{i=1}^{n} \tau_i \leqslant \tau'$. The result is immediate.

(Q-Prj) In this case, necessarily $m = 1$ and there is an $i \in 1..n$ such that $\sigma' \to \tau' = \sigma_i \to \tau_i$, hence this type is the singleton witness.

(Q-Glb) In this case, the immediate subderivation shows the LHS to be a lower bound, i.e. for each $j \in 1..m$, $\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i) \leqslant \sigma'_j \to \tau'_j$. Hence, the result follows from the induction hypothesis.

(Q-Trs) In this case, necessarily there is some intermediate type $\bigwedge_{k=1}^{p} \sigma''_k \to \tau''_k$ with subderivations showing that this type is ordered between the original two. It follows from the induction hypothesis that (1) for each $j \in 1..m$ there is some $P_j \subseteq 1..p$ such that $\forall k \in P_j.\, \sigma'_j \leqslant \sigma''_k$ and $\bigwedge_{k \in P_j} \tau''_k \leqslant \tau'_j$ and (2) for each $k \in P_j$, there is $N_k \subseteq 1..n$ such that $\forall i \in N_k.\, \sigma''_k \leqslant \sigma_i$ and $\bigwedge_{i \in N_k} \tau_i \leqslant \tau''_k$. To see the result, let

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

Figure 3.1: System of kind assignment.

$j \in 1..m$, then take as witness $W = \bigcup\{N_k \mid k \in P_j\}$. There are two requirements to be met. Let $i \in W$, it follows that $i \in N_k$ for some $k \in P_j$, hence by (2) $\sigma''_k \leqslant \sigma_i$ and by (1) $\sigma'_j \leqslant \sigma''_k$ so the first condition is met by transitivity. To see that the second condition is met, observe that by (2), for each $k \in P_j$, $\bigwedge_{i \in W} \tau_i \leqslant \bigwedge_{i \in N_k} \tau_i \leqslant \tau''_k$ and hence $\bigwedge_{i \in W} \tau_i \leqslant \bigwedge_{k \in P_j} \tau''_k$. Then the desired conclusion follows from (1).

In the backward direction we use (Q-Glb), so let $j \in [1..m]$. It follows from the second conjunct of the assumption that $\bigwedge_{i \in N}(\sigma_j \to \tau_i) \leqslant \sigma_j \to \tau_j$ by (Q-Fun) and by the first conjunct and (Q-Arr), $\bigwedge_{i \in N}(\sigma_i \to \tau_i) \leqslant \sigma_j \to \tau_j$. To see the result simply observe that also $\bigwedge_{i \in [1..n]}(\sigma_i \to \tau_i) \leqslant \bigwedge_{i \in N}(\sigma_i \to \tau_i)$. □

Intersection refinement types. We now consider those intersection types that observe a kinding discipline. We shall call such types *intersection refinement types*, because they refine the kinds with which they are associated.

3.1.5 Definition (Kind assignment)**.** A *kinded type judgement* is an expression $\theta :: \kappa$, with κ a kind. Such a judgement is pronounced "θ refines κ", and we say that the type θ is an *intersection refinement type*. Provability of judgements is defined by the system of kind assignment in Figure 3.1. Kinding can be lifted to type environments by writing $\Gamma :: \Delta$ just if $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Delta)$ and, for all $F \in \mathsf{dom}(\Gamma)$, $\Gamma(F) :: \Delta(F)$.

3.2 Intersection type assignment

The associated type assignment system is very similar to the one we surveyed in Section 2.3, except on three points. The most important is the mention of the subtype relation which appears on the rule for typing applications. So that the system is applicable to situations other than characterising the (co-)trivial automaton model checking problem for recursion schemes such as those in Chapter 5, there is a more general form of the rule for handling constants. Finally, the system is more closely associated with the underlying kinding of the terms that it types, since we require a kinding relation to hold of the types assigned to variables in the rule for abstractions and the types assigned to the basic constants are required to respect the kinds induced by the signature. For this reason we say that the system is an intersection *refinement* type system. The system is parametrised over the underlying set of base types, the particular types assigned to the term constants and the type theory.

$$\frac{\mathsf{type}(c) = \bigwedge_{i=1}^{n} \tau_i}{\Gamma \vdash c : \tau_i} \text{ (T-Cst)}$$

$$\frac{\Gamma,\, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : q \qquad \sigma_i :: \kappa_i \ (i \in [1..n])}{\Gamma \vdash \lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.\, t : \sigma_1 \to \cdots \to \sigma_n \to q} \text{ (T-Abs)}$$

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

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

Figure 3.2: Type assignment system.

3.2.1 Definition (Intersection refinement type system)**.** A *intersection refinement type system*, $\mathcal{T}$, is a triple $\langle \Sigma, Q, \mathsf{type} \rangle$ in which Σ is a sigmature, Q is an intersection type theory and type is an assignment of an intersection type σ to each constant $c \in \Sigma$. We require that the assignment of types to the basic constants is *kind respecting*, that is:

$$\forall c \in \Sigma.\, \mathsf{type}(c) :: \mathsf{kind}(c)$$

An *intersection type judgement* is an expression of the form $\Gamma \vdash t : \tau$ whose derivations are defined inductively by the system for intersection type assignment shown in Figure 3.2. As usual, the notation $\Gamma,\, x : \sigma$ stands for $\Gamma \cup \{x : \sigma\}$ whenever $x \notin \mathsf{dom}(\Gamma)$. We write $\bigwedge \mathbb{T}(\Gamma)(t)$ for the intersection $\bigwedge \{\tau \mid \Gamma \vdash t : \tau\}$ of all types assignable to t in Γ.

We will sometimes have cause to introduce two separate type systems $\mathcal{T}_1$ and $\mathcal{T}_2$ in the same context and hence, in such a situation, we will sometimes differentiate between the two by annotating the typing relations $\vdash_{\mathcal{T}_1}$ and $\vdash_{\mathcal{T}_2}$.

In Kobayashi's intersection type system, the types assigned to constants were fixed according to the particular property automaton under consideration, and these types were known to refine the corresponding kinds of the tree constructors. Here, since we allow an arbitrary choice of types for the constants, we have imposed a condition to the same effect. The idea is that refinement type systems preserve refinement:

3.2.2 Lemma. *The following is true in any intersection refinement type system:*

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

Proof. By induction on the derivation of the first premise.

- When concluded by (T-Var) or (T-Fun), the result follows from the second premise.

- When concluded by (T-Cst), the result follows from compatibility of the components.
- When concluded by (T-App), necessarily t is of the form $u\,v$ and $\Gamma \vdash u : \sigma \to \tau$ for some σ. It follows from the third premise that $\Delta \vdash u : \kappa_1 \to \kappa$ for some κ_1. Hence, it follows from the induction hypothesis that $\sigma \to \tau :: \kappa_1 \to \kappa$ and, by definition, $\tau :: \kappa$.
- When concluded by (T-Abs), necessarily t is of the form $\lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.\,u$ and τ is of the form $\sigma_1 \to \cdots \to \sigma_n \to q$ with $\sigma_i :: \kappa_i$ for each $i \in [1..n]$. It follows from the third premise that $\Delta \vdash t : \kappa_1 \to \cdots \to \kappa_n \to o$ and hence the result is immediate.

□

Our type system only allows the substitution of a type by something stronger in the type assignment rule for applications. The more standard choice for incorporating subtyping into a type system is through the addition of a *subsumption rule* such as:

$$\frac{\Gamma \vdash s : \tau \qquad \tau \leqslant \tau'}{\Gamma \vdash s : \tau'}$$

or something equivalent such as van Bakel's extension of the variable rule [van Bakel, 2011]. Note that this rule is not admissible in our system since, e.g. $x : \top \to q \vdash x : \top \to q$ but $x : \top \to q \nvdash x : q \to q$. Admitting general subsumption is undesirable for two reasons.

The first is that the system is no longer amenable to straightforward, efficient type checking. The reason is that the space of typing derivations for any given judgement $\Gamma \vdash s : \tau$ is considerably larger, due to the fact that the structure of typing derivations is much less constrained. Consider the fact that not only can the subsumption rule appear anywhere in a typing derivation but also, more seriously, the types that appear in the conclusions of judgements within the derivation can be arbitrary. This is contrasted with the case for our type system, in which the choice of rule is completely determined by the desired conclusion and in which the type appearing in any conclusion of a judgement within a derivation must already appear as part of one of the types in the root. In other words, as we shall show in the following section, our system is *syntax directed*[1] and has the *subformula property*, which allows us to give an efficient type checking algorithm. Since our original motivation for studying the system is for use in optimising algorithms which, inevitably, contain some non-trivial amount of type checking, this is essential.

The second reason is that the system no longer preserves the well-kindedness of types, as stated in Lemma 3.2.2. In other words, with general subsumption, it is possible to start from assumptions about the free variables in a term t which are consistent with the kinding of those variables and yet infer a type for t that is inconsistent with its kind. This is because the subtype relation itself does not preserve well-kindedness of types as demonstrated by, e.g. the fact that $q \to q :: o \to o$ is stronger than $\top \to q :: (o \to o) \to o$. This kind of

[1]However, note that the exact shape of the tree is not determined by the syntax of the term appearing in the conclusion since the number of premises is a function of the types involved.

unpleasantness can be avoided by moving to a system in which, e.g. intersection types are explicitly kinded, but doing so loses much of the flavour of the approach.

On the other hand, there are distinct disadvantages to our choice. For example, the set of types that can be assigned to a given term is not upward closed and, as we shall see presently, many of the associated results are only true to "within a subtype". However, we start with two results that follow as expected. The first is that type assignment is monotonic in the environment.

3.2.3 Lemma. *Let Γ and Γ' be such that $\Gamma \leqslant \Gamma'$. It follows that if $\Gamma' \vdash t : \tau$ then there is some τ' such that $\Gamma \vdash t : \tau'$ and $\tau' \leqslant \tau$.*

Proof. By induction on the type derivation.

- When the derivation is concluded by (T-Fun) or (T-Var) then t is some symbol ξ and it follows that $\tau \in \Gamma'(\xi)$. Since $\Gamma \leqslant \Gamma'$, by definition, $\Gamma(\xi) \leqslant \Gamma'(\xi)$ and hence the result follows.
- When the derivation is concluded by (T-Cst), it is independent of the environment and so take witness τ' to be τ.
- When the derivation is concluded by (T-App), t is of the form $u\,v$ and there are two intersection types σ_1 and σ_2 such that $\Gamma' \vdash u : \sigma_1 \to \tau$, $\Gamma' \vdash v : \tau''$ for each $\tau'' \in \sigma_2$ and $\sigma_2 \leqslant \sigma_1$. It follows from the induction hypothesis that there is some type $\sigma_3 \to \tau'$ with, by (Q-Arr), $\sigma_2 \leqslant \sigma_3$ and $\tau' \leqslant \tau$ and also $\Gamma \vdash u : \sigma_3 \to \tau'$. It follows that $\Gamma \vdash u\,v : \tau'$ as required.
- When the deriviation is concluded by (T-Abs), t is of the form $\lambda x_1 \cdots x_n.\, u$ and τ has shape $\sigma_1 \to \cdots \to \sigma_n \to q$. It follows from the induction hypothesis that there is some q' such that $q' \leqslant q$ and $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash u : q'$. Hence, $\Gamma \vdash t : \tau'$.

□

3.2.4 Corollary (Monotonicity of type assignment)**.** *Let Γ and Γ' be such that $\Gamma \leqslant \Gamma'$. Then, for all terms t, $\bigwedge\mathbb{T}(\Gamma)(t) \leqslant \bigwedge\mathbb{T}(\Gamma')(t)$.*

A straightforward modification of the proof of Lemma 3.2.3 is all that is required in order to give a proof of weakening. That is, if term t can be assigned type τ in environment Γ then by pointwise adding types to Γ to obtain Γ' one has also $\Gamma' \vdash t : \tau$.

3.2.5 Lemma (Weakening)**.** *Let Γ and Γ' be such that $\Gamma \Subset \Gamma'$. It follows that if $\Gamma \vdash t : \tau$ then $\Gamma' \vdash t : \tau$.*

Proof. By induction on the type derivation. □

It is expected that typing is preserved with respect to (weak) β-reduction but, in fact, here is one place in which the lack of subsumption makes life slightly more difficult. Types may not necessarily be preserved by reduction, but they can only get stronger. Consider the following case: the judgement $y : \top \to q \vdash (\lambda x.\, x)\; y : q \to q$ is derivable,

but $y : \top \to q \vdash y : q \to q$ is not. As usual, we start with a preliminary result about substitution and then use this in order to prove subject reduction.

3.2.6 Lemma (Subject substitution)**.** *Let $\Gamma, x : \sigma \vdash s : \tau$, $\bigwedge_{i=1}^{n} \tau_i \leqslant \sigma$ and, for all $i \in [1..n]$, $\Gamma \vdash t : \tau_i$. Then there is some type τ' such that $\tau' \leqslant \tau$ and $\Gamma \vdash s[t/x] : \tau'$.*

Proof. By induction on the derivation of the first premise.

(T-Var) There are two cases. When $s = x$ and (hence) $\tau \in \sigma$ then $s[t/x] = t$ and it follows from the assumption that $\Gamma \vdash s[t/x] : \tau_i$ for each $i \in [1..n]$. It follows from Lemma 3.1.3 that necessarily there is some $i \in [1..n]$ such that $\tau_i \leqslant \tau$, from which the result follows. Otherwise $s[t/x] = s$ and the result is trivial.

(T-Cst) Then $s[t/x] = s$ and the result follows immediately.

(T-App) Then s is of the form $u\,v$ and necessarily there are derivations of $\Gamma, x : \sigma \vdash u : \sigma' \to \tau$ and $\Gamma, v : \tau_j'$ for $j \in [1..m]$ and $\bigwedge_{j=1}^{m} \tau_j' \leqslant \sigma'$. Furthermore $s[t/x] = u[t/x]\,v[t/x]$ and it follows from the induction hypothesis that there is some stronger type $\sigma'' \to \tau'' \leqslant \sigma' \to \tau''$ such that $\Gamma \vdash u[t/x] : \sigma'' \to \tau''$ and some stronger types $\tau_j'' \leqslant \tau_j'$ such that $\Gamma \vdash v[t/x] : \tau_j''$. Hence, $\bigwedge_{j=1}^{m} \tau_j'' \leqslant \bigwedge_{j=1}^{m} \tau_j' \leqslant \sigma \leqslant \sigma'$ and so $\Gamma \vdash (u\,v)[t/x] : \tau''$ and the result follows.

(T-Abs) Then s is of the form $\lambda y_1 \cdots y_n.\, u$ and $\Gamma, x : \sigma, y_1 : \sigma_1, \ldots, y_n : \sigma_n \vdash u : q$. It follows from the induction hypothesis that there is some type $q' \leqslant q$ and $\Gamma, y_1 : \sigma_1, \ldots, y_n : \sigma_n \vdash u : q'$. Hence, the result follows.

□

Subject reduction with respect to weak β then follows easily, but only up to a subtype.

3.2.7 Lemma (Subject weak β-reduction)**.** *Let $s \rhd_\beta t$ with $\Gamma \vdash s : \tau$, then there is some type τ' such that $\tau' \leqslant \tau$ and $\Gamma \vdash t : \tau'$.*

Proof. By induction on the proof of $s \rhd_\beta t$.

- If the reduction is as a result of contracting a weak β-redex, then it follows that s is of the form $(\lambda x_1 \cdots x_n.\, u)\, v_1 \cdots v_n$ and $t = u[v_1/x_1, \ldots, v_n/x_n]$. Due to the second premise, it must be that $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash u : \tau$ and $\bigwedge \mathbb{T}(\Gamma)(v_i) \leqslant \sigma_i$ for each $i \in [1..n]$. Hence, by Lemma 3.2.6, there is some type q' such that $\Gamma \vdash u[v_1/x_1, \ldots, v_n/x_n] : \tau'$.

- In case the reduction is on the left of an application, s is of the form $u\,v$ and an immediate premise has $u \rhd_\beta u'$. Necessarily there is some type $\sigma \to \tau$ such that $\Gamma \vdash u : \sigma \to \tau$ and some types τ_i ($i \in [1..n]$) such that $\Gamma \vdash v : \tau_i$ and $\bigwedge_{i=1}^{n} \tau_i \leqslant \sigma$. It follows from the induction hypothesis that there is some stronger type $\sigma' \to \tau' \leqslant \sigma \to \tau$ such that $\Gamma \vdash u' : \sigma' \to \tau'$ and hence $\Gamma \vdash u'\,v : \tau'$ as required.

- In case the reduction is on the right of an application s is of the form $u\,v$ and an immediate premise has $v \rhd_\beta v'$. Necessarily there is some type $\sigma \to \tau$ such that $\Gamma \vdash u : \sigma \to \tau$ and some types τ_i ($i \in [1..n]$) such that $\Gamma \vdash v : \tau_i$ and $\bigwedge_{i=1}^n \tau_i \leqslant \sigma$. It follows from the induction hypothesis that there are some stronger types $\tau_i' \leqslant \tau_i$ ($i \in [1..n]$) such that $\Gamma \vdash v' : \tau_i'$ and hence $\Gamma \vdash u\,v' : \tau$ as required.

□

3.3 Intersection type checking

We now turn our attention to the method of algorithmically verifying that a given typing judgement is derivable, i.e. type checking. In other words, given a type environment Γ, a term t and a strict type τ, the method of verifying that there is a proof of $\Gamma \vdash t : \tau$. Type checking is an important part of several of the algorithms that we surveyed in Section 2.4 and also our own contribution in Chapter 4. Due to the constrained format of recursion scheme rules, the terms that occur within schemes are in general of the form $\lambda x_1 \cdots x_n.\, u$ for $n \geqslant 0$ and some applicative term u. A judgement $\Gamma \vdash \lambda x_1 \cdots x_n.\, u : \tau$ is derivable just if $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash u : \tau$ is derivable, so we concentrate on type checking applicative terms. We show that a good strategy for type checking a judgement $\Gamma \vdash u : \tau$ involving an applicative term u is to forget about τ, and instead compute the set of all strict types assignable to u in Γ. The type checking problem is then reduced to membership. In this section, we show that there is a straightforward and efficient algorithm for achieving this.

Due to the restriction on the use of subtyping, derivations in the type system have a shape that is largely determined by the judgement being derived. To be precise, we can prove that when searching for a derivation, it is only necessary to consider types that are suffixes of types already occurring in the conclusion. Say that a strict type τ is a suffix of a strict type τ' just if there is some $n \in \mathbb{N}$ such that τ' is of the form $\sigma_1 \to \cdots \to \sigma_n \to \tau$. We prove a subformula property for our system:

3.3.1 Lemma (Subformula property)**.** *The following is true in any intersection type system. If $\Gamma \vdash t : \tau$ is an applicative term then there is some strict type τ' such that τ is a suffix of τ' and either:*

(i) there is some constant c such that $\tau' \in \mathsf{type}(c)$, or

(ii) there is some variable or function symbol ξ and $\tau' \in \Gamma(\xi)$.

Proof. By induction on the derivation $\Gamma \vdash t : \tau$.

- When concluded by (T-Var) or (T-Fun), it follows that t is a symbol ξ, $\Gamma(\xi)$ is of the form $\bigwedge_{i=1}^n \tau_i$ and $\tau = \tau_i$ for some i. Hence, let $\tau' = \tau = \tau_i$ and (i) follows.

- When concluded by (T-Cst), it follows that t is a constant c, $\mathsf{type}(c)$ is of the form $\bigwedge_{i=1}^n \tau_i$ and τ is exactly τ_i for some i. Hence, let $\tau' = \tau = \tau_i$ and (ii) follows.

- When concluded by (T-App), it follows that t is of the form $u\,v$ and there is some intersection σ and $n \in \mathbb{N}$ such that $\Gamma \vdash u : \sigma \to \tau$. It follows from the induction hypothesis that $\sigma \to \tau$ is the suffix of some τ' about which (i) or (ii) holds. Since τ is a suffix of $\sigma \to \tau$, the same τ' is a witness to the result.

□

Type assignment algorithm. The system is syntax directed in the sense that each judgement $\Gamma \vdash t : \tau$ completely determines the rule used to conclude a typing derivation rooted at that judgement, though it does not uniquely determine the judgements that will form the premises to the instance of that rule. This ensures that we can give a very simple and efficient algorithm for type checking. Given environment Γ and applicative term t, we define the set of strict types $\mathbb{T}(\Gamma)(t)$ by the following clauses:

$$\begin{array}{lcl}
\mathbb{T}(\Gamma)(x) & = & \Gamma(x) \\
\mathbb{T}(\Gamma)(F) & = & \Gamma(F) \\
\mathbb{T}(\Gamma)(c) & = & \mathsf{type}(c) \\
\mathbb{T}(\Gamma)(s\,t) & = & \{\tau \mid \exists\,\sigma \to \tau \in \mathbb{T}(\Gamma)(s).\ \bigwedge \mathbb{T}(\Gamma)(t) \leqslant \sigma\}
\end{array}$$

Under the assumption that the subtype relation can be decided efficiently, this recursive procedure is also efficient, since it only performs a single pass over the structure of the term t deciding a number of subtype inequalities at each application node. More precisely, at each application node $u\,v$, the algorithm decides a number of subtype inequalities which is bounded by the number of types assignable to u which, due to the subformula property, is itself bounded by the number of suffixes of types available. Hence, the number of subtype inequalities that must be decided, in total, is bounded by the product of the size of the term, the size of the environment, the size of the types assigned to constants and the maximum arity of any of the types therein.

3.3.2 Lemma (Correctness)**.** *The following is true of any applicative term t in any intersection type system:*

$$\Gamma \vdash t : \tau \quad \textit{iff} \quad \tau \in \mathbb{T}(\Gamma)(t)$$

Proof. By induction on the structure of t.

- When t is a constant c we argue as follows. In the forward direction, if $\Gamma \vdash t : \tau$ then necessarily $\mathsf{type}(c) = \bigwedge_{i=1}^{n} \tau_i$ and there is some $i \in [1..n]$ such that $\tau = \tau_i$, hence $\tau \in \mathsf{type}(c)$. Conversely, if $\tau \in \mathsf{type}(c)$, necessarily $\mathsf{type}(c)$ is of the form $\bigwedge_{i=1}^{n} \tau_i$ and the derivation is completed by (T-Cst).

- When t is a variable or function symbol ξ we argue as follows. In the forward direction, if $\Gamma \vdash t : \tau$ then necessarily $\Gamma(\xi) = \bigwedge_{i=1}^{n} \tau_i$ and τ is exactly τ_i for some i; hence $\tau \in \Gamma(\xi)$. Conversely, if $\tau \in \Gamma(\xi)$, necessarily $\Gamma(\xi)$ is of the form $\bigwedge_{i=1}^{n} \tau_i$ and so the result follows by (T-Var) or (T-Fun) as appropriate.

- When t is an application $u\ v$, we argue as follows. In the forward direction, if $\Gamma \vdash t : \tau$ then necessarily there is some intersection type σ and some $n \in \mathbb{N}$ such that $\Gamma \vdash u : \sigma \to \tau$ and $\Gamma \vdash v : \tau_i$ for each $i \in [1..n]$ and $\bigwedge_{i=1}^{n} \leqslant \sigma$. It follows from the induction hypothesis that $\sigma \to \tau \in \mathbb{T}(\Gamma)(u)$ and $\tau_i \in \mathbb{T}(\Gamma)(v)$ for each $i \in [1..n]$. It follows that $\bigwedge \mathbb{T}(\Gamma)(v) \leqslant \bigwedge_{i=1}^{n} \tau_i$ and hence $\tau \in \mathbb{T}(\Gamma)(u\,v)$. Conversely, if $\tau \in \mathbb{T}(\Gamma)(u\ v)$ then it follows that there is some intersection type σ such that $\sigma \to \tau \in \mathbb{T}(\Gamma)(u)$ and $\bigwedge \mathbb{T}(\Gamma)(t) \leqslant \sigma$. It follows from the induction hypothesis that $\Gamma \vdash u : \sigma \to \tau$ and, for all $\tau' \in \mathbb{T}(\Gamma)(v)$, $\Gamma \vdash v : \tau'$. Hence, it follows by (T-APP), that $\Gamma \vdash u\,v : \tau$.

□

3.4 Consistency of type environments

Recall from Section 2.3 that a type environment Γ was said to be consistent with a scheme just if, for each non-terminal F in the domain of Γ, each type assignment $\tau \in \Gamma(F)$ is justified by a derivation of $\Gamma \vdash \mathcal{R}(F) : \tau$. We give a slightly different formalisation of Kobayashi's definition which also extends smoothly to a definition of co-consistent environments. These are the environments that will characterise co-trivial automaton model checking. Of course, both conditions are derived from the more general definition of Kobayashi and Ong's parity game over type environments [Kobayashi and Ong, 2009]. The following is a simplification for the (co-)trivial case.

3.4.1 Definition (Type environment consistency)**.** Fix a recursion scheme $\mathcal{G}$. We say that an intersection type environment Γ is $\mathcal{G}$-*consistent* just if there is a *possibly infinite* witness, rooted at $\ggcurly \Gamma$, and built according to the following system:

$$\frac{\Gamma \vdash \mathcal{R}(F) : \tau \qquad \ggcurly \Gamma}{\Gamma \ggcurly F : \tau}\ (\textsc{Con-Strict})$$

$$\frac{\Gamma_1 \Subset \Gamma \quad \Gamma_1 \ggcurly F : \tau_1 \quad \cdots \quad \Gamma_n \Subset \Gamma \quad \Gamma_n \ggcurly F : \tau_n}{\Gamma \ggcurly F : \bigwedge_{i=1}^{n} \tau_i}\ (\textsc{Con-Intersect})$$

$$\frac{\{F_1 : \sigma_1, \ldots, F_m : \sigma_m\} \ggcurly F_1 : \sigma_1 \quad \cdots \quad \{F_1 : \sigma_1, \ldots, F_m : \sigma_m\} \ggcurly F_m : \sigma_m}{\ggcurly \{F_1 : \sigma_1, \ldots, F_m : \sigma_m\}}\ (\textsc{Con-Env})$$

Similarly, we say that an intersection type environment Γ is $\mathcal{G}$-*co-consistent* just if there is a *strictly finite* witness, rooted at $\ggcurly \Gamma$, and built from the above system.

Notice that any $\mathcal{G}$-co-consistent type environment is itself $\mathcal{G}$-consistent.

From the work of Kobayashi and Ong, it is well known what the meaning of consistency and co-consistency is with respect ground kind terms and the automaton that gives rise to the type system. A ground kind term typable in a consistent environment generates a

tree for which there exists a run-tree and a ground kind term typable in a co-consistent environment generates a tree for which there exists a finite run-tree. However, consistency and co-consistency are notions that are defined here independently of any automaton, so a natural question is what does it mean for a type environment to satisfy these conditions just in terms of the type system itself. In the case of consistent environments, a good answer is already given by the proof of soundness in [Kobayashi, 2009a], types assigned in consistent environments are preserved by reduction; in our case we have the result up to a subtype:

> *Fix a recursion scheme $\mathcal{G}$. If $\Gamma \vdash s : q$ for applicative term s and Γ is $\mathcal{G}$-consistent, then for all terms t such that $s \rhd_w^* t$, there is a type $q' \leqslant q$ such that $\Gamma \vdash t : q'$*

However, to the best of our knowledge, there is no analogous result that sheds light on the role played by co-consistency. The most obvious finiteness condition, namely that the tree generated by a ground kind term typable in a co-consistent environment is finite, is not true since even when the type system is generated by an automaton such trees may be infinite yet the run-trees over them finite. Instead, we suggest that the following condition is a kind of dual to subject reduction:

> *Fix a recursion scheme $\mathcal{G}$. If $\Gamma \vdash s : q$ for applicative term s and Γ is $\mathcal{G}$-co-consistent, then there is some term t such that $s \rhd_w^* t$ and there is a type $q' \leqslant q$ such that $\vdash t : q'$.*

These two facts will allow us to give a quite natural and compact proof of the characterisation of the (co-)trivial automaton model checking problem for recursion schemes in the final section of this chapter.

The rest of this section is devoted to proving these two facts, with the latter somewhat more difficult to obtain than the former. We use the decomposition of scheme reduction into δ- and weak β-reduction that we outlined in Section 2.1. In particular, we suggest that δ is the right notion to connect the concepts of (co-)consistency and reduction. The idea is that the derivation of (co-)consistency can be seen as a strategy for how to δ-reduce a term. We start with consistency.

3.4.2 Lemma (Consistent subject δ-reduction)**.** *Fix a scheme $\mathcal{G}$. If $s \rhd_\delta t$ and $\Gamma \vdash s : \tau$ with Γ $\mathcal{G}$-consistent, then $\Gamma \vdash t : \tau$.*

Proof. The proof follows the same inductive structure as for the proof of subject weak β-reduction. The only interesting case is when s is a non-terminal F, whence necessarily $t = \mathcal{R}(F)$. It follows from the second premise that $\bigwedge_{i=1}^{n} \tau_i \in \Gamma$ with some $i \in [1..n]$ such that $\tau = \tau_i$. By consistency, there is some $\Gamma' \Subset \Gamma$ such that $\Gamma' \vdash \mathcal{R}(F) : \tau$ and hence the result follows from Weakening (Lemma 3.2.5). ☐

3.4.3 Lemma (Consistent subject scheme reduction)**.** *Fix a recursion scheme $\mathcal{G}$. If $\Gamma \vdash s : q$ for applicative term s and Γ is $\mathcal{G}$-consistent, then for all terms t such that $s \rhd_w^* t$, there is a type $q' \leqslant q$ such that $\Gamma \vdash t : q'$*

Proof. Since each $\rhd_w$ step can be decomposed into a $\rhd_\delta$ step followed by a $\rhd_\beta$ step, the result follows from Lemmas 3.4.2 and 3.2.7. ☐

For the dual result regarding co-consistency, the difficulty just lies in extracting a terminating strategy for δ-reduction from the derivation witnessing that a particular environment is co-consistent. This is the first goal, which leads to a proof of co-consistent subject δ-reduction. The extraction of the strategy is achieved by two functions, suggestively named move and play. The first, move extracts the strategy from a single finite derivation of $\Gamma \ggg F : \tau$, which shows how to δ-reduce a given term s typable in Γ to a new term s' that is typable in an environment where $F : \tau$ has been removed.

3.4.4 Definition (Move function)**.** Let us assume, without loss of generality, that no witnessing derivation tree of a co-consistency judgement contains a node labelled by a judgement $\Gamma \uplus \{F : \tau\} \ggg F : \tau$ (i). We define a function move which, given a type environment Γ and a typing $F : \tau$ such that $\Gamma \ggg F : \tau$ (ii), yields a function which given a derivation T of typing judgement $\Gamma \vdash s : \tau'$ returns a type derivation of $(\Gamma \uplus \Gamma')\backslash\{F : \tau\} \vdash s' : \tau'$ for some s' with $s \rhd_\delta^* s'$. Throughout the definition, which is by induction on the derivation T, let us write Γ'' as a shorthand for $(\Gamma \uplus \Gamma')\backslash\{F : \tau\}$.

- When T is concluded by $\Gamma \vdash \xi : \tau'$ with ξ any symbol, there are two cases. In case $\xi = F$ and $\tau' = \tau$ then set $\mathsf{move}(\Gamma')(F : \tau)(T)$ to the derivation of $\Gamma'' \vdash \mathcal{R}(F) : \tau$ which is guaranteed to exist by (i) and (ii). In every other case, set $\mathsf{move}(\Gamma')(F : \tau)(T)$ to $\Gamma'' \vdash \xi : \tau'$ whose derivability is immediate.

- When T is concluded by an abstraction $\Gamma \vdash \lambda x.\, u : \sigma \to \tau'$ then necessarily there is some subderivation T' of $\Gamma, x : \sigma \vdash u : \tau'$. It follows from the induction hypothesis that $\mathsf{move}(\Gamma')(F : \tau)(T')$ is some derivation of $\Gamma'', x : \sigma \vdash u' : \tau'$ and hence set $\mathsf{move}(\Gamma')(F : \tau)(T)$ to:

$$\frac{\mathsf{move}(\Gamma')(F : \tau)(T')}{\Gamma'' \vdash \lambda x.\, u' : \sigma \to \tau''}$$

- When the derivation is concluded by an application $\Gamma \vdash u\, v : \tau'$ then necessarily there is some subderivation T' of $\Gamma \vdash u : \sigma \to \tau'$ and subderivations T_i of $\Gamma \vdash v : \tau_i$ for $i \in [1..n]$. It follows from the induction hypothesis that $\mathsf{move}(\Gamma')(F : \tau)(T')$ is some derivation of $\Gamma'' \vdash u' : \sigma \to \tau'$ and each $\mathsf{move}(\Gamma')(F : \tau)(T_i)$ is some derivation of $\Gamma'' \vdash v' : \tau_i$. Then set $\mathsf{move}(\Gamma')(F : \tau)(T)$ to:

$$\frac{\mathsf{move}(\Gamma')(F : \tau)(T') \qquad \mathsf{move}(\Gamma')(F : \tau)(T_i)\ (i \in [1..n]) \qquad \bigwedge_{i=1}^{n} \tau_i \leqslant \sigma}{\Gamma'' \vdash u'\, v' : \tau'}$$

It is clear from the definition that if T is a derivation of $\Gamma \vdash s : \tau'$ then there is some s' such that $\mathsf{move}(\Gamma')(F : \tau)(T)$ is a derivation of $\Gamma'' \vdash s' : \tau$ and $s \rhd_\delta^* s'$.

The second function, play, extracts a strategy from an entire, finite $\ggcurly \Gamma$ derivation for the purposes of δ-reducing a given term s typable in Γ to a term s' typable in the empty environment. It uses the previously defined function move in order specify what is extracted from sub-derivations of the form $\Gamma' \ggcurly F : \tau$.

3.4.5 Definition (Play function)**.** We define function play which takes a finite derivation C of a co-consistency judgement and a derivation T of a type judgement $\Gamma \vdash s : \tau$ and returns a derivation of $\Gamma'' \vdash s' : \tau$ such that $s \rhd_\delta^* s'$ and:

(i) If C is a derivation of $\ggcurly \Gamma'$ then $\Gamma'' = \Gamma \backslash \Gamma'$.

(ii) If C is a derivation of $\Gamma' \ggcurly F : \sigma$ then $\Gamma'' = \Gamma \backslash \{F : \sigma\}$.

(iii) If C is a derivation of $\Gamma' \ggcurly F : \tau'$ then $\Gamma'' = \Gamma \backslash \{F : \tau'\}$.

The construction is by induction on the derivation C of co-consistency.

- If C is concluded by $\ggcurly \varnothing$ then set $\mathsf{play}(C)(T) = T$. Property (i) is trivially satisfied.
- If C is concluded by $\ggcurly \{F_1 : \sigma_1, \ldots, F_m : \sigma_m\}$ for $m > 0$, then necessarily there are subderivations C_i of $\{F_1 : \sigma_1, \ldots, F_m : \sigma_m\} \ggcurly F_i : \sigma_i$ $(i \in [1..m])$. Set $\mathsf{play}(C)(T) = \mathsf{play}(C_m)(\cdots(\mathsf{play}(C_1)(T))\cdots)$. The required property then follows from the induction hypothesis and the transitivity of reduction.
- If C is concluded by $\Gamma'' \ggcurly F : \bigwedge_{i=1}^n \tau_i$ then necessarily there are subderivations C_i of $\Gamma_i \ggcurly F : \tau_i$ for some $\Gamma_i \Subset \Gamma$ $(i \in [1..n])$. Set $\mathsf{play}(C)(T) = \mathsf{play}(C_n)(\cdots(\mathsf{play}(C_1)(T))\cdots)$. The required property then follows from the induction hypothesis and the transitivity of reduction.
- If C is concluded by $\Gamma'' \ggcurly F : \tau'$ then necessarily there is some subderivation C' of $\ggcurly \Gamma''$. Set $\mathsf{play}(C)(T) = \mathsf{play}(C')(\mathsf{move}(\Gamma'')(F : \tau)(T))$. Since $\mathsf{move}(\Gamma'')(F : \tau)(T)$ is a derivation of $(\Gamma \uplus \Gamma'') \backslash \{F : \tau\} \vdash s' : \tau$ for some s' such that $s \rhd_\delta^* s'$, the required property follows from the induction hypothesis and the transitivity of reduction.

Co-consistent subject δ-reduction is then an easy corollary.

3.4.6 Lemma (Co-consistent subject δ-reduction)**.** *Fix a recursion scheme $\mathcal{G}$. If $\Gamma \vdash s : \tau$ and Γ is $\mathcal{G}$-co-consistent, then there is some term t such that $s \rhd_\delta^* t$ and $\vdash t : \tau$.*

Proof. Necessarily there is a derivation C of $\ggcurly \Gamma$ and a derivation T of $\Gamma \vdash s : \tau$. Then, by construction, $\mathsf{play}(C)(T)$ is a derivation of $\vdash t : \tau$ for some t such that $s \rhd_\delta^* t$. ☐

This result has allowed us to deduce the existence of a term t that is reachable via δ-reductions and which is, moreover, typable in the empty environment. However, in order to convert this δ-reduction sequence into a scheme reduction sequence, we will need to eliminate all of the new weak β-redexes. Unfortunately, we cannot just δ-expand such redexes, since this will obviously not preserve the property of typability under the empty environment, so we will need to β-reduce at least some of them (it may be possible to

δ-expand some redexes which are not "inspected" by the typing derivation). Hence, we intend to weak β-reduce as many as possible by reducing to a weak β-normal form. Any such normal form can be folded up again without losing empty-environment typability.

3.4.7 Lemma. *If $\vdash n : q$ is a weak β-normal form, then $\vdash n^+ : q$.*

Proof. We show, for all weak β-normal forms t, by induction on the long form of t, that if $\vdash t : q$ is headed by a terminal symbol then $\vdash t^+ : q$. Necessarily, due to the typing assumption and the assumption on being a normal form, t is of shape $a\, s_1 \cdots s_n$ for normal forms s_i ($i \in [1..n]$). It follows from the assumption that $\vdash a : \sigma_1 \to \cdots \to \sigma_n \to q$ and, for all such i, $\bigwedge \mathbb{T}(\varnothing)(s_i) \leqslant \sigma_i$. For each i, there are two cases. In case s_i is also headed by a terminal symbol, then it follows from the induction hypothesis that $\bigwedge \mathbb{T}(\varnothing)(s_i^+) \leqslant \sigma_i$. Otherwise s_i must be headed by a variable or non-terminal symbol, but since the type environment is empty, necessarily $\bigwedge \mathbb{T}(\varnothing)(s_i) = \top$ and so $\sigma_i = \top$. Hence, also in this case, $\bigwedge \mathbb{T}(\varnothing)(s_i^+) \leqslant \sigma_i$. Hence $\vdash a\, s_1^+ \cdots s_n^+$ as required. □

The method we described above is now played out in the final lemma.

3.4.8 Lemma (Co-consistent weak subject reduction). *Fix a recursion scheme $\mathcal{G}$. If $\Gamma \vdash s : q$ for applicative term s and Γ is $\mathcal{G}$-co-consistent, there is some term t and type $q' \leqslant q$ such that $s \rhd_w^* t$ and $\vdash t : q'$.*

Proof. It follows from Lemma 3.4.6 that there is some term t' such that $s \rhd_\delta^* t'$ and $\vdash t' : q$. Since t' is simply typed (kinded), it follows that it has a weak β-normal form n and, by Lemma 3.2.7, there is a type $q' \leqslant q$ such that $\vdash n : q'$. Finally, by Corollary 2.1.17, therefore $s = s^+ \rhd_w^* n^+$ and, by Lemma 3.4.7, $\vdash n^+ : q'$. □

3.5 Higher-order model checking is type inference

Since our system is more permissive than Kobayashi's, in the sense that it types more terms, it is not at all clear that it characterises the (co-)trivial automaton model checking problem for recursion schemes. Before we can consider this further, we should first define formally the notion of a type system induced by an automaton:

3.5.1 Definition (Type system induced by an automaton). Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ the type system induced by $\mathcal{A}$ is the system $\langle \Sigma, Q, \mathsf{type} \rangle$ in which:

$$\mathsf{type}(c) = \bigwedge \{ \bigwedge (S|_1) \to \cdots \to \bigwedge (S|_n) \to q \mid S \models \delta(q, c) \}$$

for all $c \in \Sigma$ and $(q_1, q_2) \in \Theta$ iff $q_1 = q_2$. Recall that $S|_i$ stands for $\{q' \mid (i, q') \in S\}$.

We will not usually introduce the type system induced by a given automaton explicitly but assume the reader can infer it from the introduction of the automaton. Hence, in the following we will typically introduce an instance of the higher-order model checking problem, i.e. a scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ and a (co-)trivial automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ but we will freely make use of the implicitly introduced refinement type system specified

by the signature Σ and the induced type system. Furthermore, we will often just refer to the type system induced by $\mathcal{A}$ simply as $\mathcal{A}$ when the context is clear enough.

Now, let us phrase the statement of characterisation in our framework as follows:

> *Fix a recursion scheme $\mathcal{G}$ and an alternating (co-)trivial automaton $\mathcal{A}$. Then* $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$ *if, and only if, there exists a $\mathcal{G}$-(co-)consistent intersection type environment Γ such that $\Gamma :: \mathcal{N}$ and $\Gamma \vdash S : q_0$ in $\mathcal{A}$.*

Now, since any type environment that is (co-)consistent and types $S : q_0$ in Kobayashi's system is also (co-)consistent and types $S : q_0$ in our system, the completeness direction of this statement follows immediately. However, soundness is not so obvious and so we establish it in the rest of this section. For the trivial automaton case, we follow the same proof structure as Kobayashi [2009a]. For the co-trivial automaton case, which was previously only shown by a quite intricate argument covering all parity automata in [Kobayashi and Ong, 2009], we give a direct proof in the same style as the trivial case, which is made possible by Lemma 3.4.8.

We first make clear the relationship between typability of ground, closed applicative terms and the acceptance of the trees they represent. In both the trivial and co-trivial cases, typability implies acceptance.

3.5.2 Lemma. *Fix a recursion scheme $\mathcal{G}$ and a trivial automaton $\mathcal{A}$ over the same signature. If $\Gamma \vdash t : q$ is a closed applicative term then $t^\perp \in \mathcal{L}(\mathcal{A}^\perp, q)$.*

Proof. By induction on the (long) form of t. If t is headed by a non-terminal symbol, then $t^\perp = \perp$ and hence $t^\perp \in \mathcal{L}(\mathcal{A}^\perp, q)$. Otherwise $t = a\, s_1 \,\cdots\, s_n$ and $t^\perp = a\, s_1^\perp \cdots s_n^\perp$. Since t is well typed by q, it follows that $\Gamma \vdash a : \sigma_1 \to \cdots \to \sigma_n \to q$ for some types σ_i and $\bigwedge\mathbb{T}(\Gamma)(s_i) \leqslant \sigma_i$ ($i \in [1..n]$). It follows that $S = \{(i, q') \mid q' \in \sigma_i\}$ is a satisfying assignment. By Lemma 3.1.3, for each $(i, q') \in S$, there is some $q'' \in \bigwedge\mathbb{T}(\Gamma)(s_i)$ such that $q'' \leqslant q'$ but, since the type system induced by the automaton is discrete it follows that $q'' = q'$. By the induction hypothesis, $s_i^\perp \in \mathcal{L}(\mathcal{A}^\perp, q')$ and hence $a\, s_1^\perp \cdots s_n^\perp \in \mathcal{L}(\mathcal{A}, q)$. ☐

3.5.3 Lemma. *Fix a recursion scheme $\mathcal{G}$ and a co-trivial automaton $\mathcal{A}$ over the same signature. If $\vdash t : q$ is a closed applicative term then $t^\perp \in \mathcal{L}(\mathcal{A}_\perp, q)$.*

Proof. By induction on the (long) form of t. Since t is typable in the empty environment, it must be that t is of the form $a\, s_1 \,\cdots\, s_n$ and necessarily $\vdash a : \sigma_1 \to \cdots \to \sigma_n \to q$ for some types σ_i and $\bigwedge\mathbb{T}(\varnothing)(s_i) \leqslant \sigma_i$ ($i \in [1..n]$). It follows that $S = \{(i, q') \mid q' \in \sigma_i\}$ is a satisfying assignment. By Lemma 3.1.3, for each $(i, q') \in S$, there is some $q'' \in \mathbb{T}(\Gamma)(s_i)$ such that $q'' \leqslant q'$ but, since the type system induced by the automaton is discrete, $q'' = q'$. By the induction hypothesis, $s_i^\perp \in \mathcal{L}(\mathcal{A}, q')$ (since necessarily any such s_i must be headed by a terminal symbol) and hence $a\, s_1^\perp \cdots s_n^\perp \in \mathcal{L}(\mathcal{A}_\perp, q)$. ☐

The soundness of consistent type assignment then follows. Since the tree defined by the scheme is the limit of its finite approximants, and each approximant derives from $S : q_0$ it follows that each approximant is a tree accepted by the automaton from state q_0.

3.5.4 Lemma (Soundness of consistent type assignment)**.** *Fix a recursion scheme $\mathcal{G}$ and a trivial automaton $\mathcal{A}$ over the same signature. If there exists a $\mathcal{G}$-consistent Γ such that $\Gamma \vdash S : q_0$, then $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$.*

Proof. Assume for contradiction that $\mathsf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^\perp)$. Then there is some finite witness t such that $S \rhd_w^* t$ and $t^\perp \notin \mathcal{L}(\mathcal{A}^\perp)$. However, since $\Gamma \vdash S : q_0$ it follows from subject reduction (Lemma 3.4.3) that there is some stronger type $q \leqslant q_0$ such that $\Gamma \vdash t : q$ but, since the system induced by the automaton is discrete, $q = q_0$. Hence, by Lemma 3.5.2, $t^\perp \in \mathcal{L}(\mathcal{A}^\perp)$ which contradicts the original assumption. ☐

The soundness of co-consistent type assignment follows because, if a co-consistent environment types $S : q_0$, then we can always find some reduct t of S that represents a tree $t^\perp$ already known to be accepted by the automaton (and hence have a complete, finite run-tree).

3.5.5 Lemma (Soundness of co-consistent type assignment)**.** *Fix a recursion scheme $\mathcal{G}$ and a co-trivial automaton $\mathcal{A}$ over the same signature. If there exists a $\mathcal{G}$-co-consistent Γ such that $\Gamma \vdash S : q_0$, then $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}_\perp)$.*

Proof. Since Γ is co-consistent it follows from subject reduction (Lemma 3.4.8) that there is some term t such that $S \rhd^* t$ and some state q' such that $q' \leqslant q_0$ and $\vdash t : q'$. Since the type system induced by the automaton is discrete, $q' = q_0$. By Lemma 3.5.3, it follows that $t^\perp \in \mathcal{L}(\mathcal{A}_\perp)$. Since $\perp$ is rejected from every state, it follows that any tree which extends $t^\perp$ is also accepted, hence $\mathsf{Tree}(t) = \mathsf{Tree}(S) = \mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}_\perp)$. ☐

To obtain the completeness parts of the characterisations we can either appeal to the existing work of Kobayashi and Ong [2009] or, alternatively, appeal to the correctness proof of our algorithm in Chapter 4. The results are summarised by the following theorem:

3.5.6 Theorem (Characterisation of model checking)**.** *Fix a recursion scheme $\mathcal{G}$ and trivial automaton $\mathcal{A}$ over the same signature. Then the following are equivalent:*

(i) $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$

(ii) there exists some $\mathcal{G}$-consistent Γ such that $\Gamma :: \mathcal{N}$ and $\Gamma \vdash_{\mathcal{A}} S : q_0$

(iii) there does not exist any $\mathcal{G}$-co-consistent Γ such that $\Gamma :: \mathcal{N}$ and $\Gamma \vdash_{\mathcal{A}^c} S : q_0$

(iv) $\mathsf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^c_\perp)$

Proof. To see (i) implies (iv), observe that $\mathcal{L}(\mathcal{A}^c_\perp) = \mathcal{L}((\mathcal{A}^\perp)^c) = \mathcal{L}(\mathcal{A}^\perp)^c$. That (iv) implies (iii) follows from the contrapositive of Lemma 3.5.5. It follows from Lemma 4.5.14 that either there exists a $\mathcal{G}$-consistent environment $\Gamma_\exists$ such that $\Gamma_\exists :: \mathcal{N}$ and $\Gamma_\exists \vdash_{\mathcal{A}} S : q_0$ or there exists a $\mathcal{G}$-co-consistent environment $\Gamma_\forall$ such that $\Gamma_\forall :: \mathcal{N}$ and $\Gamma_\forall \vdash_{\mathcal{A}^c} S : q_0$, and hence (iii) implies (ii). Finally, that (ii) implies (i) is the content of Lemma 3.5.4. ☐

Chapter **4**

Model Checking via Type Directed Abstraction Refinement

The following is work conducted jointly with Ong and Neatherway and a version is due to appear in the conference proceedings of POPL '14 [Ramsay et al., 2014].

4.1 Introduction

In Section 2.5 we presented the case for the use of recursion schemes in program verification and surveyed some of the recent work in this direction. However, for these approaches to be successful, there need to exist efficient decision procedures for the model checking problem that work well in practice. Hence, this interest in verification has driven the development of several ingenious algorithms [Kobayashi, 2009b, 2011, Neatherway et al., 2012, Broadbent et al., 2012, Broadbent and Kobayashi, 2013] whose aim is to solve higher-order model checking problems for many "practical" instances. However, the state of this effort is summarised well by Broadbent and Kobayashi [2013]:

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

This is the goal of this chapter. Our main contribution is a new algorithm for higher-order model checking and a large body of evidence to show empirically that it scales well to HORS consisting of several thousand rules. In contrast, the largest instances considered in the literature to date are of the order of several hundred rules. By way of an example, the order-2 benchmark $\mathcal{G}_{2,10000}$ of Kobayashi [2009b], which consists of 10006 rules, can be processed by our prototype implementation in less than one minute.

Our algorithm, which decides the HORS model checking problem with respect to alternating trivial tree automata, has been designed to be scalable. Since the inherent

worst-case complexity of HORS model checking is extreme, to have any chance at all of solving non-trivial instances, one has to work in the belief that those instances that are met in practice are not pathological. It is thus essential to ensure that only work that is *relevant* to deciding the particular instance at hand is actually computed. To help achieve this goal, our algorithm is designed in the abstraction refinement paradigm [Clarke et al., 2000]. A relatively cheap but coarse-grained approximation to the problem is processed initially and, as much as possible, detail is only added by successive iterations where the problem instance necessitates it. Moreover, it can be shown that our algorithm is *fixed-parameter polynomial time* in the size of the scheme; the parameters that are fixed are the order and arity of the scheme, and the size of the tree automaton.

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

In order to gain more information, and thus populate the context, each iteration consists of constructing a sound abstraction of the scheme's configuration graph [Kobayashi, 2009b]. Since recursion schemes have no facility to inspect the data that they operate over, the behaviours of the scheme arise from the complex interactions between higher-order functions. Hence, we have designed this abstraction around a traditional control flow analysis (CFA) [Jones, 1981], but with an important twist: in our abstraction, parameters to function calls are distinguished according to the intersection types that they inhabit, in other words, according to the properties that they satisfy. This is, in turn, a function of the context and hence, as the algorithm progresses and the size of the context increases, so the abstractions become more precise, as they are able to distinguish more instances of function calls.

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

We have implemented the algorithm in a tool, PREFACE, and evaluated its performance over the several hundred problem instances that are now either recorded in the literature or which have resulted from verification tools for higher-order programs. These instances range from a few tens of rules to several thousands and from first order schemes up to fifth order, and a few beyond. The results show very clearly that, whilst PREFACE is sometimes

a little slower than other model checkers on examples of up to around one hundred rules, its great strength lies in solving examples of many hundreds of rules, where it consistently outperforms other model checkers, and several thousands of rules, which are typically instances that it alone can solve. To help analyse the weaknesses of the algorithm, we have also constructed a family of examples which are deliberately difficult for our tool to solve (yet sometimes much easier for other tools reported in the literature).

Outline The rest of the chapter is structured as follows. In Section 4.2 we give an informal outline of the algorithm by means of an example. In Section 4.3 the algorithm is formally defined and, in Section 4.4, the reader is guided through a run of the procedure on a problem instance. In Section 4.5 we show that the algorithm is correct and always terminates. In Section 4.6 we discuss our prototype implementation and present a digest of the empirical evaluation and associated analysis. Finally, in Section 4.7, we discuss related work.

4.2 Type directed abstraction refinement

The starting point for the algorithm is the characterisation of the trivial automaton model checking problem for recursion schemes given in Theorem 3.5.6. The theorem states that $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ just if (a) there is some $\mathcal{G}$-consistent environment Γ such that $\Gamma :: \mathcal{N}$ and $\Gamma \vdash_{\mathcal{A}} S : q_0$ and that $\mathsf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^{\perp})$ just if (b) there is some $\mathcal{G}$-consistent environment Γ such that $\Gamma :: \mathcal{N}$ and $\Gamma \vdash_{\mathcal{A}^c} S : q_0$. Our algorithm tries to prove (a) and (b) simultaneously, by iteratively constructing two type environments $\Gamma_\exists$ and $\Gamma_\forall$ which are possible witnesses to (a) and (b) respectively. Since an invariant of the algorithm is that $\Gamma_\exists$ is always $\mathcal{G}$-consistent in $\mathcal{A}$ and $\Gamma_\forall$ is always $\mathcal{G}$-co-consistent in $\mathcal{A}^c$, it follows that at most one of the two environments can prove the type assignment $S : q_0$; in fact, upon termination, exactly one of the two will do so. Since the two type systems $\mathcal{A}$ and $\mathcal{A}^c$ share the same underlying set of types, let us call an intersection type an *acceptance type* when we regard it as part of the system $\mathcal{A}$ and let us call it a *rejection type* when we regard it as part of the system $\mathcal{A}^c$.

The algorithm starts with $\Gamma_\exists$ and $\Gamma_\forall$ empty, which are trivially (co-)consistent. On each iteration, new type assignments are inferred which will be added to one or the other of the environments. The way that new acceptance types and new rejection types are inferred is by, on each iteration, constructing and interrogating an auxiliary structure, called the *abstract configuration graph*. As its name suggests, this graph is an abstraction and its precision is a function of the size of the type environments $\Gamma_\exists$ and $\Gamma_\forall$. The more precise the abstraction the more useful the type information that can be deduced by interrogating it. Hence, starting from the empty *context*, $C_0 = \langle \Gamma^0_\exists, \Gamma^0_\forall \rangle$, the abstraction refinement cycle continues as follows, a diagram is shown in Figure 4.1. In iteration

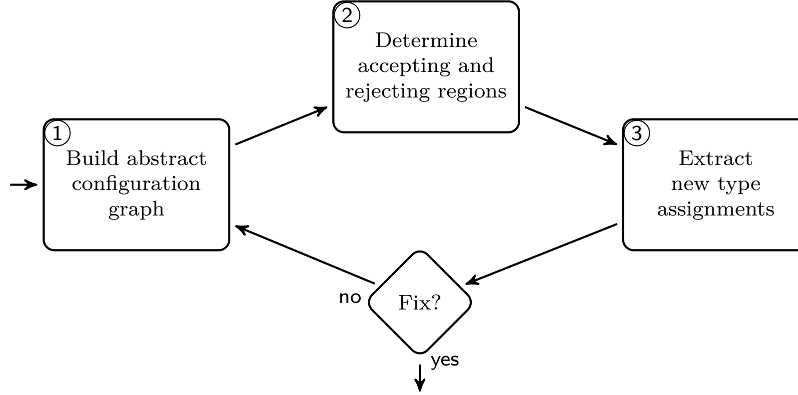


Figure 4.1: Overview of the algorithm.

$i+1$ with context $C_i = \langle \Gamma^i_\exists, \Gamma^i_\forall \rangle$, the abstract configuration graph is constructed. Two subgraphs are then carved out called the accepting and rejecting regions. These regions are the parts of the graph from which it is possible to obtain consistent acceptance typings and co-consistent rejection typings respectively. Types are then extracted from the regions and added to context C_i to form a strictly larger context $C_{i+1} = \langle \Gamma^{i+1}_\exists, \Gamma^{i+1}_\forall \rangle$. If one of these two environments can already type $S : q_0$ then the algorithm terminates since, by construction, it will not be possible to extract any new information on the next iteration and hence a fixed point has been reached. Otherwise the cycle repeats and, since C_{i+1} is strictly larger, the abstract configuration graph constructed in iteration $i+2$ will be strictly more precise, and strictly more type information will be deduced. This process cannot go on forever because there are only finitely many intersection refinement types associated with a given scheme. Hence, after a finite amount of time one of the environments will be exhausted and thus witness the corresponding assertion (a) or (b).

As we explained in Section 2.4, the configuration graph of a model checking problem instance was first introduced by Kobayashi [2009b] as a simplification of the kinds of trees used in the proof of completeness in the characterisation theorem of his work with Ong [Kobayashi and Ong, 2009]. It is a kind of product construction, pairing up the reduction relation of the scheme and the transition function of the property automaton and it takes the shape of a rooted, directed graph. Since we are solving the alternating trivial automaton model checking problem for recursion schemes, the configuration graphs that we are interested in are something in between the simple ones of [Kobayashi, 2009b], which are appropriate to deterministic trivial automaton properties, and the much more complicated ones of [Kobayashi and Ong, 2009], which are appropriate to alternating parity automaton properties. Moreover, we shall not be interested in the configuration graphs themselves, which are in general infinite, but rather in our finite abstractions which we shall outline through an example.

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

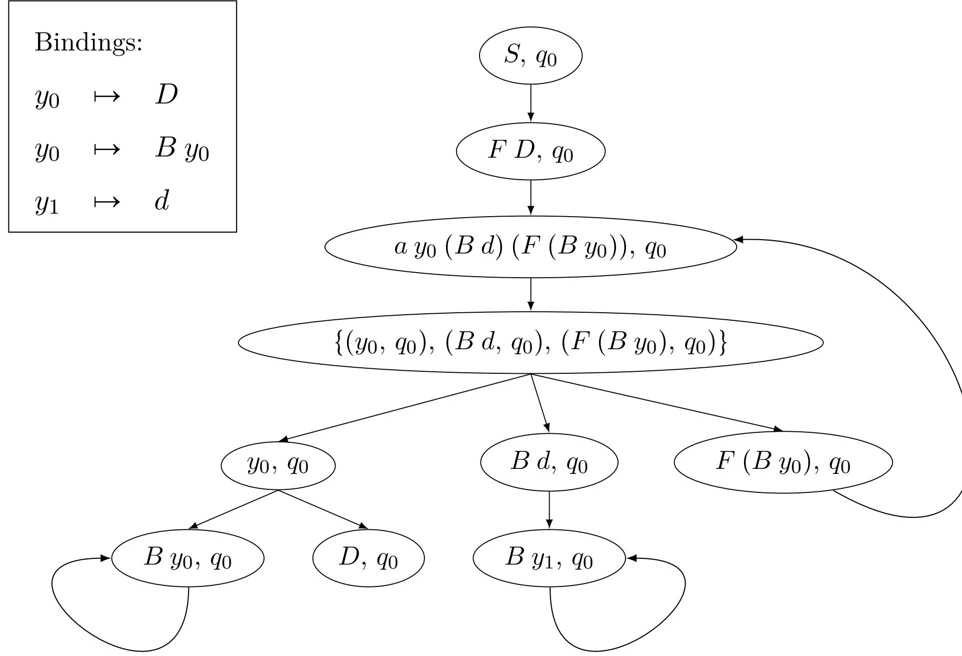


Figure 4.2: An abstract configuration graph.

Recursion scheme $\mathcal{G}$:

$$\begin{array}{lcl} S & = & F\,D \\ F & = & \lambda x.\, a\,x\,(B\,d)\,(F\,(B\,x)) \\ B & = & \lambda z.\, B\,z \\ D & = & d \end{array}$$

Property automaton $\mathcal{A}$:

$$\begin{array}{lcl} \delta(q_1, d) & = & \mathsf{t} \\ \delta(q_0, a) & = & (1, q_0) \wedge (2, q_0) \wedge (3, q_0) \end{array}$$

As described above, the algorithm begins with the initial context $C_0 = \langle \Gamma^0_\exists, \Gamma^0_\forall \rangle$ consisting of $\Gamma^0_\exists = \varnothing$ and $\Gamma^0_\forall = \varnothing$. In order to infer new type assignments it constructs the abstract configuration graph $\mathsf{ACG}(C_0)$, which is given in full in Figure 4.2. The construction of the graph starts at the root, which a vertex labelled (S, q_0). Every vertex of the graph is either labelled by a pair (t, q) of a ground kind term and a state, called a *configuration*, or by a finite set of such configurations. If there is a vertex (t, q) in a graph $\mathsf{ACG}(\langle \Gamma_\exists, \Gamma_\forall \rangle)$, then it should be read as "neither $\Gamma_\exists \vdash_\mathcal{A} t : q$ nor $\Gamma_\forall \vdash_{\mathcal{A}^c} t : q$ is provable". Hence, the starting point for the graph $\mathsf{ACG}(C_0)$ is the fact that $S : q_0$ is not provable in $\varnothing$.

The shape of the successors of a vertex, if it has any, depend upon whether the vertex is a configuration or a set and in case of the former, the syntactic class of which its head symbol is a member. Here the root is a configuration and its head symbol is S which is a non-terminal and hence the term part is a redex. In such cases, the vertex has at most one successor and that successor *represents* the contraction of the redex. The contraction of

the redex is $F\,D$ and, because neither $\Gamma^0_\exists \vdash_\mathcal{A} F\,D : q_0$ nor $\Gamma^0_\forall \vdash_{\mathcal{A}^c} F\,D : q_0$ is provable, it has a successor, which is labelled by $(F\ D,\ q_0)$. Since this vertex is of the same form, it also has a single successor which represents the contraction of the redex, however the situation is more complicated because this redex involves parameters. It is along such edges that the abstraction happens: rather than substituting actual parameters for formals in the successor, formals are substituted by special variables which are used by the abstraction to represent sets of terms. So we create a new variable y_0 and make a note $y_0 \mapsto D$ that one of the possible instantiations of y_0 is D. In this way, we are always able to recover the real contraction $a\ D\ (B\ d)\ (F\ (B\ D))$ from the abstract one $a\ y_0\ (B\ d)\ (F\ (B\ y_0))$ by rewriting occurrences of the variables y_0 using the *bindings* $y_0 \mapsto D$. This ensures we are building a *sound* abstraction, in the sense of incorporating all the behaviours of the original. This kind of abstraction is, in essence, a traditional control flow analysis [Jones, 1981], but here we go one step further, which is critical in order to obtain completeness. The extra step that we take is to record the acceptance type and rejection type of the new variable y_0. The acceptance type (respectively rejection type) of y_0 is just the intersection of types assignable to the term that it has been introduced to represent, in this case D i.e. $\bigwedge\mathbb{T}_\mathcal{A}(\Gamma^0_\exists)(D) = \top$ (respectively $\bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma^0_\forall)(D) = \top$). The types of such variables will later determine when new variables should be created in order to represent actual parameters, as y_0 was here, or if variables created previously should be reused.

The frontier of our construction so far consists of a terminal headed configuration. The successors of a configuration $(a\,s_1\ \cdots\ s_n,\ q)$ depend upon the satisfying assignments to $\delta(q,\ a)$. For each satisfying assignment $\mathcal{S}$ there is one successor, which is the set of configurations $\{(s_i,\ q') \mid (i,\ q') \in \mathcal{S}\}$. In this case, $\delta(q,\ a)$ is satisfied just if the first argument of a is accepted from q_0, the second argument is accepted from q_0 and the third argument is accepted from q_0. Consequently, the set is $\{(y_0,\ q_0),\ (B\,d,\ q_0),\ (F\,(B\,y_0),\ q_0)\}$. Such a set $\{(t_1,\ q_1), \ldots, (t_n,\ q_n)\}$ should be read as "there is some i such that $\Gamma_\exists \vdash_\mathcal{A} t_i : q_i$ is not provable". The successors of the set are those configurations $(t,\ q)$ in the set which are provable in neither $\Gamma_\exists$ nor $\Gamma_\forall$.

So let us now consider the frontier configuration $(F\ (B\ y_0),\ q_0)$. Since it is a non-terminal headed configuration, if it has a successor then the successor abstractly represents the contraction in the same way as before. However, since the actual parameter $B\,y_0$ has the same acceptance type and rejection type as the previous one that we considered, C (and is the same kind), we will not create a new variable to represent By_0, but we will *reuse* the variable y_0 for D. So the abstract contraction is the term $a\,y_0\ (B\,d)\ (F\ (B\,y_0))$ and we form a loop in the graph. To ensure the abstraction remains sound we note that another possible instantiation of y_0 is $B\ y_0$, but observe that this merging of vertices resulting from reusing y_0 has caused some loss of information. Now the term $a\,y_0\ (B\ d)\ (F\ (B\ y_0))$ represents many different concrete instances, including some that are not possible in the original problem, such as $a\,D\,(B\,d)\,(F\,(B\,(B\,D)))$ which results from rewriting the leftmost occurrence of y_0 to D and the rightmost occurrence to $B\ D$. The actual parameters D and $B\ y_0$ have become confused because, according to the current context C_0, they have the same acceptance and rejection types. In this way, *types direct the abstraction* and refinement will occur because type information in the context increases.

By contrast, the vertex $(B\ d,\ q_0)$ has an actual parameter whose acceptance and re-

jection types are non-trivial. The terminal d has acceptance type $\bigwedge\mathbb{T}_{\mathcal{A}}(\Gamma^0_\exists)(d) = q_1$ and rejection type $\bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma^0_\forall)(d) = q_0$. We have not yet created a new variable with those types, so we do so now, assigning the new variable y_1 the acceptance type q_1 and the rejection type q_0. We record that d is a possible instantiation of y_1 and label the successor $(B\, y_1,\, q_0)$. Following the same method, this new vertex has itself as successor. Officially we ought to note that y_1 is a possible instance of y_1, but such trivial circularities will make no difference to the outcome so we will omit it for brevity.

Let us now consider the frontier vertex $(y_0,\, q_0)$, which is variable headed. At variable headed configurations the consequences of the abstraction are felt. A vertex of the form $(y\, s_1\, \cdots\, s_n,\, q)$ has a successor $(t\, s_1\, \cdots\, s_n,\, q)$ for each binding $y \mapsto t$. For reasons that we have already discussed, its child $(B\, z_0,\, q_0)$ has itself as a successor. Its other child is the vertex $(D,\, q_0)$ whose contraction is $(d,\, q_0)$, however, because $\Gamma^0_\forall \vdash_{\mathcal{A}^c} d : q_0$ is *not* provable, we do not add vertex $(d,\, q_0)$ to the graph and so $(D,\, q_0)$ has no successors. We say that $(D,\, q_0)$ is a *rejecting leaf*. This completes the construction of the abstract configuration graph.

We now consider any rejecting leaves in the graph. Rejecting leaves are configurations $(F\, s_1\, \cdots\, s_n,\, q)$ for which the corresponding type assignment $(F\, s_1\, \cdots\, s_n,\, q)$ is not provable in either $\Gamma_\exists$ or $\Gamma_\forall$, yet the type assignment associated with the contraction is provable in $\Gamma_\forall$. This points directly to a weakness in the context. In our example, $\Gamma^0_\forall \vdash_{\mathcal{A}^c} d : q_0$ is provable, so d is a tree rejected by $\mathcal{A}^\perp$. Now D reduces to d in one step, yet $\Gamma_\forall \vdash_{\mathcal{A}^c} D : q_0$ is not provable! The rejection environment $\Gamma_\forall$ *ought* to type $D : q_0$, since D generates a tree that is rejected by $\mathcal{A}^\perp$, but it doesn't. Hence, we have discovered a new rejection type assignment. In this case, the parent of $(D,\, q_0)$ is $(y_0,\, q_0)$ but, because y_0 represents many different terms, it doesn't follow that because $\mathsf{Tree}(D)$ is rejected by $\mathcal{A}^\perp$ from state q_0 then also *every* instance of y_0 is rejected from the same state; but if the parent of $(D,\, q_0)$ had have been, for example, another non-terminal headed node $(t',\, q_0)$, then we could repeat the same reasoning and deduce that also $t' : q_0$ ought to have been rejection typable. The vertices that we classify as representing typing assignments that ought to have been provable under the rejecting environment are collectively called the *rejecting region* and from this region new rejection types are extracted. Where the rejecting region is, roughly speaking, all the vertices that "definitely" can reach rejecting leaves, the *accepting region* is all the vertices that "definitely" cannot reach rejecting leaves. New acceptance types are extracted from the accepting region. In this case, the rejecting region is the single vertex $(D,\, q_0)$ and the accepting region consists of all the B headed configurations.

Type extraction from the regions follows a similar approach to that defined in [Kobayashi and Ong, 2009]. Briefly, a type is assigned to each prefix s of the term component of each vertex $(s\, t_1\, \cdots\, t_n,\, q)$ in the accepting region. The type assigned to s is constructed by recursively computing the type σ assigned to t_1 by considering all the ways in which t_1 is used within the region (the vertices where t_1 is itself a prefix), recursively computing the type τ assigned to the prefix $s t_1$ and then forging the arrow $\sigma \to \tau$. The base case is where the entire component, considered as a trivial prefix, is assigned the type q. The rejecting region is handled similarly, but care must be taken to only consider uses of t_1 in vertices that are strictly closer to the leaves so that a well founded co-consistency argument can be given in the end. The type assignments to prefixes that are themselves non-terminal

symbols are extracted and added to the appropriate environments. In this case, there is only one vertex in the rejecting region, so its only prefix is assigned the type $D : q_0$. There are three vertices in the accepting region, of which the prefix B of $(B\,d,\, q_0)$ is assigned acceptance type $q_1 \to q_0$ since d is known to have type q_0, prefix B of $(B\,y_0)$ is assigned type $\top \to q_0$ since there are no uses of y_0 in the region and, for the same reason, prefix B of $(B\,y_1)$ is assigned the type $\top \to q_0$.

The new context is therefore $C_1 = \langle\{B : (\top \to q_0) \wedge (q_1 \to q_0)\},\, \{D : q_0\}\rangle$. Since the new context neither types $S : q_0$ as accepting nor as rejecting, the process repeats, but this time with more type information, so more of the parameters to calls will be distinguished. The second iteration starts in a similar way to the first by contracting $(S,\, q_0)$, but now the actual parameter D in vertex $(F\,D,\, q_0)$ has a non-trivial type. According to C_1, $\bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma^1_\forall)(D) = q_0$. Consequently, when a variable y_3 is created to represent D in the abstract contraction, it is assigned rejection type q_0. However, this means that the contraction $(a\,y_3\,(B\,d)\,(F\,(B\,y_3)),\, q)$ can already be seen to be rejecting, in the sense that $\Gamma^1_\forall,\, y_3 : q_0 \vdash_{\mathcal{A}^c} a\,y_3\,(B\,d)\,(F\,(B\,y_3)) : q_0$, and hence the corresponding configuration is not added to the graph. Since there are no more vertices to be considered, this signals the end of the construction of the graph. Since every vertex can "definitely" reach a rejecting vertex, the rejecting region contains every vertex in the graph, including the root $(S,\, q_0)$. Consequently, $S : q_0$ is added to the rejecting environment and the algorithm terminates, correctly deducing that the input is a no-instance.

4.3 A decision procedure for model checking

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

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

Construction of abstraction

The abstraction is based on a traditional CFA [Jones, 1981], in the sense of over-approximating reduction using an abstract environment. An important twist on the usual formulation is that here every variable in the environment is associated with a kind and a pair of intersection types.

Typed variables. The main mechanism for abstraction will be a set of typed variables. By means of an abstract environment (to be described shortly) each variable represents

the set of terms that can be obtained from it by repeated substitution. Each variable has three pieces of associated type information: an acceptance type, a rejection type and a kind. An essential part of the algorithm is in ensuring that type information is invariant across the abstraction, i.e. if a variable abstracts a set of terms, then every term in the set shares the same acceptance type, rejection type and kind as the variable. The association of a variable with its type information is made precise in the following.

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

Type context. The algorithm is ultimately concerned with constructing a pair of type environments $\langle \Gamma_\exists, \Gamma_\forall \rangle$ such that $\Gamma_\exists$ is $(\mathcal{G},\mathcal{A})$-consistent and $\Gamma_\forall$ $(\mathcal{G},\mathcal{A}^c)$-co-consistent. We will speak of $\Gamma_\exists$ as the "acceptance" type environment and $\Gamma_\forall$ as the "rejection" type environment. Furthermore, we stipulate that every such pair of environments, which we shall call a *type context*, understands the basic assumptions we have made about the typed variables, i.e. the type information contained in A and R (as defined in Definition 4.3.1, but viewed as type environments for the typed variables) is also contained in $\Gamma_\exists$ and $\Gamma_\forall$ respectively.

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

(i) $\Gamma_\exists :: \mathcal{N} \cup \mathsf{K}$

(ii) $\Gamma_\forall :: \mathcal{N} \cup \mathsf{K}$

(iii) For all $y \in \mathcal{Y}$, $\Gamma_\exists(y) = \mathsf{A}(y)$ and $\Gamma_\forall(y) = \mathsf{R}(y)$

Abstract configurations. As mentioned in the previous section, the abstraction itself is a finite representation of the possibly infinite *configuration graph*, as defined by Kobayashi and Ong [2009]. In this *concrete* configuration graph, the configurations are pairs of a *closed* term (a reduct of the start symbol of the scheme) and a state of the automaton, and the edges that connect them must respect the constraints of both the reduction relation of the scheme and of the transition function of the automaton. The configurations (t, q) should be viewed as a kind of primitive assertion (which may or may not hold), that term t generates a tree which is accepted from state q. The abstraction is always rooted at (S, q_0) and the exploration of the state space from this point corresponds to computing the necessary requirements, phrased in terms of configurations, that must be satisfied in order that S generate a tree accepted from state q_0. In the *abstract* configuration graph, defined shortly, configurations are still pairs of term and state, but now the term is abstract, which in our setting means that it can contain free occurrences of typed variables.

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

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

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

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

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

4.3.5 Definition (Abstract configuration graph)**.** An *abstract configuration graph* A is a tuple $\langle V, E, B \rangle$ in which $\langle V, E \rangle$ is a directed graph and B is a set of mappings from variables $y \in \mathcal{Y}$ to terms t that may contain occurrences of them. Each vertex $v \in V$ is either (i) an abstract configuration or (ii) a finite set of abstract configurations; and edges $E \subseteq V \times V$ are unlabelled. Given a typing context C, the *abstract configuration graph of* C, denoted $\mathsf{ACG}(C)$, is the abstract configuration graph $\langle V_C, E_C, B_C \rangle$ defined inductively by the system in Figure 4.3.

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

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

(G1) Whenever all the following are true: then all the following are also true:

- (i) $(S,\, q_0)$ is C-unknown
- $(S,\, q_0) \in V_C$

(G2) Whenever all the following are true: then all the following are also true:

- (i) $(F\, s_1\, \cdots\, s_n,\, q) \in V_C$
- (ii) $\mathcal{R}(F) = \lambda x_1\, \cdots\, x_n.\, t$
- (iii) $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_n \to o$
- (iv) $(t[y_1/x_1, \cdots, y_n/x_n],\, q)$ is C-unknown
- (v) for each $i \in [1..n]$, y_i is exactly: $\mathsf{var}(\bigwedge \mathbb{T}_{\mathcal{A}}(\Gamma_\exists)(s_i),\, \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i),\, \kappa_i)$

then:

- $(t[y_1/x_1, \ldots, y_n/x_n],\, q) \in V_C$
- $\langle (F s_1 \cdots s_n, q), (t[y_1/x_1, \ldots, y_n/x_n], q) \rangle \in E_C$
- for each $i \in [1..n]$: $y_i \mapsto s_i \in B_C$

(G3) Whenever all the following are true: then all the following are also true:

- (i) $(a\, s_1\, \cdots\, s_n,\, q) \in V_C$
- (ii) $S \models \delta(q,\, a)$
- (iii) for all $(i,\, q') \in S$, $(s_i,\, q')$ is not C-rejected

then:

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

(G4) Whenever all the following are true: then all the following are also true:

- (i) $\{(s_1,\, q_1), \ldots, (s_n, q_n)\} \in V_C$
- (ii) $i \in [1..n]$
- (iii) $(s_i,\, q_i)$ is not C-accepted

then:

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

(G5) Whenever all the following are true: then all the following are also true:

- (i) $(y\, s_1\, \cdots\, s_n,\, q) \in V_C$
- (ii) $y \mapsto t \in B_C$

then:

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

Figure 4.3: Abstract configuration graph construction.

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

Rule (G2) simulates the contraction of a redex, but it does so in an abstract way. To apply the rule requires that a configuration $(F\, s_1\, \cdots\, s_n,\, q)$ containing a redex occurs in the graph. The consequence is that an abstraction of the contraction of that redex is added as a new configuration. However, it is abstract because, rather than substituting actual parameters for formals, typed variables are substituted for the formals. These

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

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

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

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

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

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

4.3.8 Lemma. *Let C be a type context. Then the size of V_C is bounded by a polynomial function of the size of the scheme (assuming all other parameters are fixed).*

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

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

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

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

Note that a rejecting leaf is not itself C-rejecting, by Lemma 4.3.6 since it is in the graph at all it is necessarily C-unknown, but its contractum is C-rejecting. Similarly accepting leaves are not themselves C-accepting, but the contractum of an accepting leaf is C-accepting.

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

The rejecting region

Region of rejection. The construction of an ACG from a given type context C is a method for analysing the type context. By constructing the graph it is possible to see where the information in the type context is deficient, and the main tools for identifying and correcting deficiencies are the regions and region type extraction respectively. Consider a rejecting leaf v of the form $(F\, s_1 \cdots s_n,\ q)$. By definition, the contraction of this configuration using (G2) would yield a configuration $(t[y_1/x_1,\ \ldots,\ y_n/x_n],\ q)$ which is already C-rejecting. In other words, the tree generated by *any* term of the form $t[t_1/y_1,\ \ldots,\ t_n/y_n]$ such that $\bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(t_i) \leqslant \bigwedge \mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(y_i)$ for each i is sure to be rejected from state q. Assuming that $\Gamma_\forall$ is co-consistent, this follows because necessarily $\Gamma_\forall \vdash_{\mathcal{A}^c} t[t_1/y_1,\ \ldots,\ t_n/y_n] : q$. Recalling Proposition 4.3.7, one such sequence of t_i are the actual parameters of the term component of the rejecting leaf we started with: v. Hence, because we know that the contractum of the term part of v generates a tree that is rejected from state q_0, necessarily the term part of v itself generates a tree that is rejected from state q_0. So we have identified that v should be classified as C-rejecting (but is not currently). Through analogous reasoning (and remembering the conjunctive and disjunctive interpretations of the child relation in the graph), it is possible to identify other such vertices which are necessarily rejecting. The collection of all such is called the rejecting region.

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

(R1) If c is a rejecting leaf then $c \in \mathsf{RR}(C)$.

(R2) If $\{(s_1,\ q_1), \ldots, (s_n,\ q_n)\} \in V_C$ and there exists $j \in [1..n]$ and $(s_j,\ q_j) \in \mathsf{RR}(C)$ then $\{(s_1,\ q_1), \ldots, (s_n,\ q_n)\} \in \mathsf{RR}(C)$.

(R3) If $\langle (F\ s_1\ \cdots\ s_n,\ q),\ (t,\ q)\rangle \in E_C$ and $(t,\ q) \in \mathsf{RR}(C)$ then $(F\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RR}(C)$.

(R4) If $(a\, s_1\ \cdots\ s_n,\ q) \in V_C$ and, for every v, $\langle (a\, s_1\ \cdots\ s_n,\ q),\ v\rangle \in E_C$ implies $v \in \mathsf{RR}(C)$, then $(a\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RR}(C)$.

(R5) If $(y\ s_1\ \cdots\ s_n,\ q) \in V_C$ and, for all $y \mapsto t \in B_C$, $(t\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RR}(C)$ then $(y\ s_1\ \cdots\ s_n,\ q) \in \mathsf{RR}(C)$.

Unless it is the final iteration of the algorithm, the rejecting region will always be non-empty. The fact that an absence of rejecting vertices is an absence of counterexamples in the abstraction is formalised later, in Lemma 4.3.16.

Rejection type extraction. The vertices in the rejecting region are those configurations, that we have identified by constructing $\mathsf{ACG}(C)$, which *should* be classified by the context as rejecting, but *are not* – each is necessarily C-unknown, since it belongs to the graph. So the rejecting region represents a weakness in the context. To remedy it, from the region we will extract new type information to be added to the context ready for the next iteration. The idea is as follows. If the rejecting region is non-empty, then there must be some rejecting leaf $(F\,s_0\,\cdots\,s_n,\,q)$ headed by some non-terminal symbol F with definition $F = \lambda x_1\,\ldots\,x_n.\,t$. By definition, the abstract contraction $(t[y_1/x_1,\ldots,y_n/x_n],\,q)$ of this leaf must be rejection typable, in the sense that $\Gamma_\forall \vdash_{\mathcal{A}^c} t[y_1/x_1,\ldots,y_n/x_n] : q$ where, for each i, $\bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(y_i) = \bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_i)$. Therefore, the rejection type:

$$\tau = \bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_0) \to \cdots \to \bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_n) \to q$$

is a good one for F, since $\Gamma_\forall,\, x_1 : \bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_0),\,\ldots,\,x_n : \bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma_\forall)(s_n) \vdash_{\mathcal{A}^c} t : q$. For this reason, τ can be added to the known co-consistent rejection type environment $\Gamma_\forall$ to form $\Gamma'_\forall$ and this new environment will still be co-consistent. Hence, one can then consider the parent(s) of the leaf $(F\,s_0\,\cdots\,s_n,\,q)$. If some parent is, for example, another non-terminal headed vertex $(G\,t_0\,\cdots\,t_n,\,q)$, then the situation repeats. Now, the leaf, although not typable in $\Gamma_\forall$, *is* rejection typable in $\Gamma'_\forall$, and hence a type $\bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma'_\forall)(t_0) \to \cdots \to \bigwedge\mathbb{T}_{\mathcal{A}^c}(\Gamma'_\forall)(t_n) \to q$ may be taken for G. The full definition below considers all the possible cases.

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

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

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

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

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

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

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

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

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

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

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

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

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

Furthermore, whenever the rejecting region is non-empty, then genuinely new type information will be extracted. Taken together with Lemma 4.3.16, the following result is the key measure of progress in the algorithm.

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

The accepting region

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

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

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

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

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

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

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

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

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

Acceptance type extraction. To extract new type information from the accepting region we follow the approach of Kobayashi [2009b] (a simplification of work by Kobayashi and Ong [2009]), in which types are assigned to prefixes of configurations recursively based on the kind of the prefix.

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

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

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

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

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

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

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

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

Fixed point construction

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

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

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

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

Since the initial environments $\Gamma^0_\exists$ and $\Gamma^0_\forall$ are trivially $\mathcal{G}$-consistent in $\mathcal{A}$ and $\mathcal{G}$-co-consistent in $\mathcal{A}^c$ respectively and since every extension of these environments by env_A and env_R preserves this property, it follows that the limit of the sequence also enjoys the property and hence can be relied upon to decide the model checking problem. Furthermore, since progress is guaranteed by Lemma 4.3.16 and Lemma 4.3.14, and the size of rejecting environment $\Gamma_\forall$ is bounded by the number of well-kinded types, we can state the following correctness theorem:

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

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

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

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

Table 4.1: Typed variables listing.

4.4 A narrated example

By way of an example, we consider an instance of the HORS model checking problem due to Kobayashi [2009b] (though it is itself an encoding of a problem considered by Might and Shivers [2008]). Although this scheme only generates a finite tree, it has an interesting control flow structure.

Model checking instance

The instance consists of the following recursion scheme $\mathcal{G}$ over the terminal symbols $\texttt{flow} : o \to o$ and $\texttt{end} : o$. The property is specified by the deterministic trivial automaton $\mathcal{A}$ consisting of a single state q_0 and a single transition $\delta(q_0, \texttt{end}) = \texttt{t}$. The idea is that the scheme will be rejected just if a use of Lam flows to the result, i.e. if $\texttt{flow}$ appears in $\mathsf{Tree}(\mathcal{G})$. However, this is not the case, and hence $\mathsf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A})$ is determined after three iterations. The typed variables that will be used throughout this example are shown in Table 4.1. The reader may wish to periodically consult this listing whilst following the constructions described below.

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

Iteration 1

In the first iteration, the initial context C_0 just consists of $\langle \mathsf{A}, \mathsf{R} \rangle$. The graph $\mathsf{ACG}(C_0)$ is shown in Figure 4.4. Note that the accepting and rejecting regions are coloured light and dark grey respectively. The binding set for the graph is as shown below. Note that, as a small optimisation to aid readability, we will never add bindings of the form $y \mapsto y$ to the set during graph construction.

$$
y2 \mapsto Lam' \quad y2 \mapsto Lam \quad y0 \mapsto Id \quad y1 \mapsto C3 \quad y1 \mapsto C2\, y_0
$$

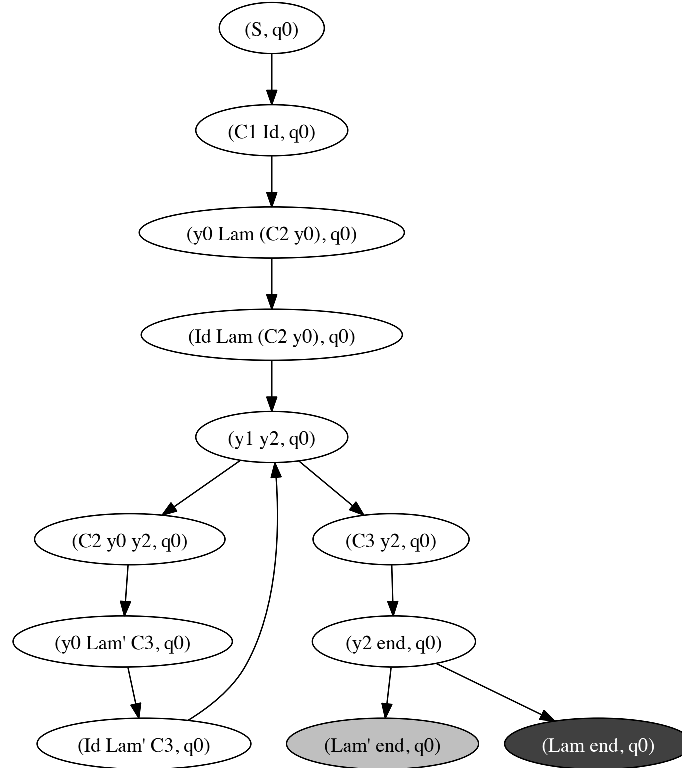


Figure 4.4: Abstract configuration graph for iteration 1.

Since there is no intersection type information for any of the non-terminals during the first iteration, there is no way to distinguish the two actual parameters Lam and Lam' which occur in two different calls to the non-terminal Id, once due to the contraction of $Id\,Lam\,(C2\,y0)$ and again due to the contraction of $Id\,Lam'\,C3$. Consequently, the typed variable $y2$, which is used to abstract arguments of type $(\top,\, \top,\, o \to o)$, confuses the two calls which leads to the undesirable (and spurious) effect of having the redex $Lam\;end$ occur in a configuration in the graph. This configuration is a rejecting leaf, because the configuration that would result from contracting the redex, which would be of the form $(\texttt{flow}\,y,\, q_0)$, is C-rejecting for any C, in essence because the tree $\texttt{flow}\,s$ is rejected by $\mathcal{A}^{\perp}$ for any tree s.

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

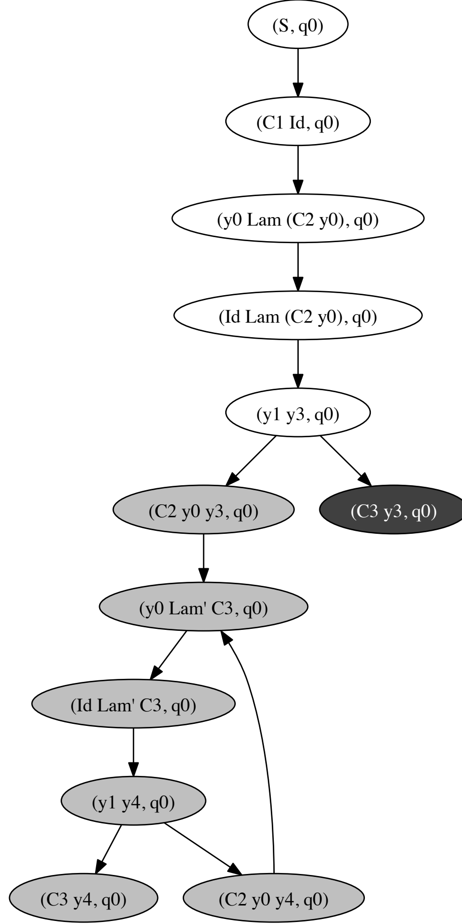


Figure 4.5: Abstract configuration graph for iteration 2.

is rejected from state q_0.

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

Hence, the context for the next iteration will be:

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

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

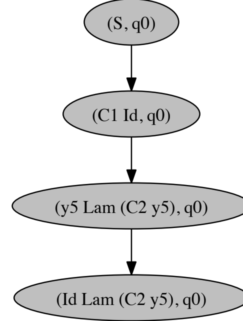


Figure 4.6: Abstract configuration graph for iteration 3.

Iteration 2

The graph for the second iteration $\mathsf{ACG}(C_1)$ is shown in Figure 4.5. The binding set for the graph is as shown below.

$$\begin{array}{lll} y4 \mapsto Lam' & y1 \mapsto C3 & y0 \mapsto Id \\ y3 \mapsto Lam & y1 \mapsto C2\,y0 & \end{array}$$

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

$$\begin{array}{rl} \mathrm{C3}: & (q_0 \rightarrow q_0) \rightarrow q_0 \\ \mathrm{C2}: & ((q_0 \rightarrow q_0) \rightarrow ((q_0 \rightarrow q_0) \rightarrow q_0) \rightarrow q_0) \rightarrow (q_0 \rightarrow q_0) \rightarrow q_0 \\ \mathrm{C2}: & ((q_0 \rightarrow q_0) \rightarrow ((q_0 \rightarrow q_0) \rightarrow q_0) \rightarrow q_0) \rightarrow \top \rightarrow q_0 \\ \mathrm{Id}: & (q_0 \rightarrow q_0) \rightarrow ((q_0 \rightarrow q_0) \rightarrow q_0) \rightarrow q_0 \end{array}$$

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

Iteration 3

The third iteration is the last (non-trivial) iteration of the algorithm when running on this instance. The graph $\mathsf{ACG}(C_2)$ is depicted in Figure 4.6. In this case, the associated binding set is simply $y5 \mapsto \text{Id}$. Here, the configuration $(Id\,Lam\,(C2\,y5),\, q_0)$ is an accepting leaf since, if it were to be contracted it would yield the configuration $(C2\,y5\,Lam)$ which is already known to be C_2-accepting. Hence the construction of the graph is halted at this point. Since there are no rejecting leaves, it follows that the accepting region is universal. Consequently, among the types extracted from the region is the typing $S : q_0$ which concludes the run of the algorithm with the answer YES.

4.5 Correctness of the decision procedure

In this section we go through the correctness proof for the algorithm. Unlike the rest of this dissertation this section is not intended to be read in sequence but should be treated as a catalogue, containing an entry for each of the assertions in Section 4.3. For a better idea of how these results fit together the reader is directed to that section.

Well kindedness of the ACG

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

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

(i) v is a configuration $(t,\, q)$ and $\mathcal{N} \cup \mathsf{K} \vdash t : o$.

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

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

(G1) By definition, $S : o \in \mathcal{N}$.

(G2) It follows from the induction hypothesis that

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

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

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

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

(G3) It follows from the induction hypothesis that:

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

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

(G4) Here (i) follows directly from the induction hypothesis.

(G5) It follows from the induction hypothesis that

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

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

□

Proof of Lemma 4.3.6

Lemma 4.3.6 is the assertion that all of the vertices in the configuration graph are C-unknown. This is guaranteed by the preconditions on the construction rules for the ACG; although not all rules have an explicit precondition, those that do not are missing one because it is unnecessary due to the way that certain edges preserve non-typability. We prove the following stronger statement:

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

- *(i) v is a configuration and v is C-unknown.*
- *(ii) v is a set of configurations and both of the following are true:*
  - *(a) there exists $c \in v$ such that c is not C-accepted*
  - *(b) for all $c \in v$, c is not C-rejected*

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

(G1) By the precondition (S, q_0) is trivially C-unknown.

(G2) By the precondition $(t[y_1/x_1, \ldots, y_n/x_n], q)$ is trivially C-unknown.

(G3) By the precondition, for each $(i, q') \in S$, (s_i, q') is not C-rejected. Assume for contradiction that, for all $(i, q') \in S$, (s_i, q') is C-accepted. It follows that then $\Gamma_\exists \vdash_{\mathcal{A}} a\, s_1 \cdots s_n : q$ which contradictions the induction hypothesis.

(G4) It follows from the induction hypothesis that, for each $j \in [1..n]$, (s_j, q_j) is not C-rejected. By the precondition also (s_i, q_i) is not C-accepted. Hence (s_i, q_i) is C-unknown.

(G5) It follows from the induction hypothesis that $(y\,s_1\,\cdots\,s_n,\,q)$ is C-unknown. Since, additionally, $\mathbb{T}(\Gamma_\forall)(y) = \mathbb{T}(\Gamma_\forall)(t)$ and $\mathbb{T}(\Gamma_\exists)(y) = \mathbb{T}(\Gamma_\exists)(t)$ it follows that also $(t\,s_1\,\cdots\,t_n,\,q)$ is C-unknown.

□

Proof of Lemma 4.3.8

Lemma 4.3.8 is the assertion that the number of vertices in any given ACG is bounded by a polynomial in the size of the scheme. We provide a slightly more detailed analysis in the following. The key observation is that all the terms that occur in the ACG have a very specific shape and the number of possible terms of that shape is not more than a polynomial in the size of the scheme. However, it is worth noting that, as a function of the size of the input as a whole, the bound is much worse: since the typed variables occur in terms in the ACG the number of nodes may be hyperexponential in the order of the scheme.

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

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

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

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

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

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

(G1) Clearly, S is compound.

(G2) It follows from the induction hypothesis that $F\,s_1\,\cdots\,s_n$ is compound, so each s_i is simple. Clearly $t[y_1/x_1, \ldots, y_n/x_n]$ is simple (and hence compound).

(G3) It follows from the induction hypothesis that $a\,s_1\,\cdots\,s_n$ is compound and hence each s_i is simple.

(G4) It follows from the induction hypothesis that, for each $j \in [1..n]$, s_j is simple and hence s_i is simple (and hence compound).

(G5) It follows from the induction hypothesis that $y\,s_1\,\cdots\,s_n$ is compound and hence each s_i is simple and also that t is simple. Hence $t\,s_1\,\cdots\,s_n$ is compound.

□

We can now prove the statement in the lemma:

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

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

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

It follows from the definition of $\mathcal{Y}$ that $|\mathcal{Y}| = O(K_{N-1}^2\,|\mathcal{S}|)$. Take an element $y \mapsto t$ of B_C. Because $\mathcal{G}$ is normalised, the (simple) term t is either a variable or has the form $F\,\overline{x}$, where the variables are elements of $\mathcal{Y}$. Thus there are $O(P\,|\mathcal{Y}|^A)$ such simple terms t. It follows from Lemma 4.5.2(i) that $|B_C| = O(P\,|\mathcal{Y}|^{A+1})$.

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

Proof of Lemma 4.3.10

Lemma 4.3.10 is the assertion that every leaf in the ACG is either accepting or rejecting. Of course, if a leaf is a non-terminal headed configuration, then this is obvious. Other possible shapes of leaf can be ruled out by a straightforward case analysis.

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

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

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

The following proof of the lemma, restated below, is due to Ong:

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

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

- If v is of the form $(F\, s_1 \cdots s_n,\, q)$ then, by (G2), it must be that v is either C-accepted or C-rejected and hence v is either an accepting or rejecting leaf (respectively).
- If v is of the form $(a\, s_1 \;\cdots\; s_n,\, q)$ then, by (G3), it must be that for every satisfying assignment S to $\delta(q, a)$ there is some $(i\; q') \in S$ such that $(s_i,\, q')$ is C-rejected. However, this is inconsistent with $v \in V_C$ since, it follows from Lemma 4.5.3 that then $\bigcup_{i=1}^{n}\{(i\; q') \mid \Gamma_\forall \vdash_{\mathcal{A}^c} s_i : q'\}$ is a satisfying assignment to $\delta^c(q, a)$. Consequently $\Gamma_\forall \vdash_{\mathcal{A}^c} a\; s_1 \;\cdots\; s_n : q$ which contradicts Lemma 4.3.6 (i).
- If v is of the form $\{(s_1,\, q_1), \ldots, (s_n,\, q_n)\}$ then, by (G3)) it must be that, for all $i \in [1..n]$, $(s_i,\, q_i)$ is C-accepted. But this contradicts Lemma 4.3.6 part (ii)(b).
- If v is of the form $(y\; s_1 \;\cdots\; s_n,\, q)$ then, by construction, there is some $y \mapsto t \in B_C$ and consequently, v being a leaf is inconsistent with the closedness of $\mathsf{ACG}(C)$ under (G5).

□

Proof of Lemma 4.3.16

Lemma 4.3.16 is the assertion that an absence of rejecting leaves implies that the whole ACG is accepting. This is because the graph is essentially *closed* in the terminology of [Kobayashi, 2009b]. Even though there may still be imprecision in the graph due to the abstraction which leads to extra, spurious behaviours, those behaviours are not problematic, in the sense of generating rejecting trees. Hence, every non-terminal headed node has a successor (unless it is already known to be accepting), every terminal headed node has a satisfying assignment, every satisfying assignment has all of its members and every variable headed node has all of its instantiations. The statement is as follows:

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

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

(A1) Assume $(F\ s_1\ \cdots\ s_n,\ q)$ is not an accepting leaf, since, by assumption, there are no rejecting leaves, it is unknown. It follows from the coinduction hypothesis that $(F\ s_1\ \cdots\ s_n,\ q) \in V_C$, hence the result follows by (G2).

(A2) It follows from the coinduction hypothesis that $(a\ s_1\ \cdots\ s_n,\ q) \in V_C$, it follows by Lemma 4.3.10 that the only leaves in $\mathsf{ACG}(C)$ are accepting or rejecting so this vertex has a successor. It follows by (G3) that necessarily this successor is a set $\{(s_i,\ q') \mid (i,\ q) \in S\}$ for some satisfying assignment S to $\delta(q,\ a)$.

(A3) It follows from the induction hypothesis that, in this case, $\{(s_1,\ q_1), \ldots, (s_n,\ q_n)\} \in V_C$. It follows from (G4) that if $i \in [1..n]$ then $(s_i,\ q_i) \in V_C$ is a child.

(A4) It follows from the coinduction hypothesis that $(y\ s_1\ \cdots\ s_n,\ q) \in V_C^{\mathsf{A}}$. Since $y \mapsto t \in B_C$, the result follows from (G5).

□

Proof of Lemma 4.3.13

Lemma 4.3.13 is the assertion that the addition of newly extracted rejection types to the rejection environment in the context preserves co-consistency of that environment. The proof is broken down into two main parts. The first part is a straightforward demonstration that the environment extracted from the witnessing tree of some configuration actually types the configuration. The second part shows that each such environment is actually co-consistent (when added to the context). Recall that, for an environment to be co-consistent, it must be the case that all of its constituent typings are well justified and, additionally that all the justifications together be finite. The key observation is that, given a typing $F : \tau$ extracted from the region, there must be a particular vertex $(F\,s_1\ \cdots\ s_n,\ q)$ that it is extracted from and then, by the definition of the region, one can justify the typing based on the fact that its successor — which is essentially the RHS of F — is already known (inductively) to be rejecting. Since the witnessing trees are well-founded, so are the justifications.

We begin the formal account by showing that the environment extracted from a witnessing tree types the configuration that it witnesses.

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

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

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

Proof. By induction on T. In cases (R1), (R3) and (R5), the result follows by definition. In case (R2), the result follows from the induction hypothesis. It remains to check case (R4). In this case v is of the form $(a\,s_1\ \cdots\ s_n,\ q)$. Let the children be in the set W. Necessarily,

for each child $w \in W$ there is a sub-proof T_w and, by (M4), $\mathcal{M}(T) = \biguplus\{\mathcal{M}(T_w) \mid w \in W\}$. It follows that, for each satisfying assignment $S \models \delta(q, a)$, either the corresponding set of configurations $\{(s_i, q') \mid (i, q') \in S\}$ is in W or is already known to be rejecting, in the sense that there is some j such that $\Gamma_\forall \vdash s_j : q'$. It follows from the induction hypothesis that, for each such set w in W, it is also the case that there is some j such that $\Gamma_\forall \uplus \mathcal{M}(T_w) \vdash s_j : q'$. Consequently, for each satisfying assignment S, there is some $(j, q') \in S$ such that $\Gamma_\forall \uplus \biguplus\{\mathcal{M}(T_w) \mid w \in W\} \vdash s_j : q'$. It follows from Lemma 4.5.3 that the set of all such pairs (j, q') is a satisfying assignment to $\delta^c(q, a)$ and hence $\Gamma_\forall \uplus \mathcal{M}(T) \vdash a\, s_1 \cdots s_n : q$ as required. □

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

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

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

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

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

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

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

such that the following conditions hold:

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

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

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

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

To this end, distinguish cases on the form of ξ. If ξ is a variable or non-terminal symbol then let τ_ξ be the type extracted from the proof of T''. Then (i) and (ii) hold by definition and (iii) is true since, in this case, S_i is exactly $\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T''))(s_i)$ and $\mathcal{M}(T') \leqslant \mathcal{M}(T'')$. Otherwise ξ is a terminal symbol c and we let τ_ξ be the type induced by $\delta(q, c)$ according to rule (T-Cst). It follows from Lemma 4.5.4 that, therefore (i) and (ii) hold and furthermore that $\bigwedge \mathbb{T}(\Gamma_\forall \uplus \mathcal{M}(T''))(s_i) \leqslant \bigwedge S_i$. Since $\mathcal{M}(T') \leqslant \mathcal{M}(T'')$, the result follows transitively. □

We can now prove the following statement, of which Lemma 4.3.13 is a corollary, since it should be clear from the definition of co-consistency that the multiplication of a finite number of co-consistent environments is again co-consistent.

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

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

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

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

Let this type be τ. Since $\Gamma_\forall$ is already co-consistent, it remains just to check that there is some $\Gamma \subseteq \mathcal{M}(T) \uplus \Gamma_\forall$ such that $\Gamma \gg F : \tau$. Take as witness $\Gamma_\forall$ since, by definition, v is a rejecting leaf and hence its successor $(\mathcal{R}(F)[y_1/x_1, \ldots, y_n/x_n], q)$ is typable in the environment $\Gamma_\forall$. Due to 4.3.7, for each $i \in [1..n]$, $\Gamma_\forall(y_i) = \bigwedge \mathbb{T}(\Gamma_\forall)(s_i)$ and so the result follows by a renaming of variables.

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

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

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

Let this type be τ. It follows from the induction hypothesis that $\Gamma_\forall \uplus \mathcal{M}(T')$ is already co-consistent and, from the above Lemma 4.5.4, that $\Gamma_\forall \uplus \mathcal{M}(T')$ is sufficient to prove $\mathcal{R}(F)[y_1/x_1, \ldots, y_n/x_n] : q$ (*). Hence, to see that there is some $\Gamma \Subset \Gamma_\forall \uplus \mathcal{M}(T)$ such that $\Gamma \gg F : \tau$ take as witness $\Gamma_W := \Gamma_\forall \uplus \mathcal{M}(T')$. It remains to show that:

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

Due to (*) it suffices to show only that, for all $i \in [1..n]$, $\bigwedge \mathbb{T}(\Gamma_W)(s_i) \leqslant \Gamma_W(y_i)$. To this end, consider some $\tau \in \Gamma_W(y_i)$, it must be that either (i) $\tau \in \Gamma_\forall(y_i)$ or (ii) $\tau \in \mathcal{M}(T')(y_i)$. In the former case, also $\tau \in \mathbb{T}(\Gamma_\forall)(s_i)$ by Proposition 4.3.7. In the latter case, observe that, by Lemma 4.5.5, $\bigwedge \mathbb{T}(\mathcal{M}(T'))(s_i) \leqslant \mathcal{M}(T')(y_i)$ and $\Gamma_W \leqslant \mathcal{M}(T')$ and the result follows transitively.

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

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

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

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

□

Proof of Lemma 4.3.14

Lemma 4.3.14 asserts that the types extracted from the rejecting environment are genuinely new to the context. The proof comes down to one observation about rejecting leaves which is that, if the rejection type extracted from one of them was not new to the context, then the leaf itself would necessarily be C-rejecting, which contradicts the fact that all vertices are known to be C-unknown. Hence, the progress guarantee really only comes down to the rejecting leaves, the rest of the rejecting region are all “added extras”. The statement of the Lemma is as follows:

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

Proof. Let $v = (F\ s_1\ \cdots\ s_n,\ q)$ be the rejecting leaf, by Lemma 4.3.6, v is itself not C-rejected. By (RR1) v is also a leaf in $\mathsf{RR}(C)$, let its witnessing proof tree be T. By the strengthened Lemma 4.3.13 proved above, $\mathcal{M}(T) \uplus \Gamma_\forall \vdash_{\mathcal{A}^c} F\, s_1\ \cdots\ s_n : q$ and by definition $\mathcal{M}(T)$ is the single binding

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

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

Proof of Lemma 4.3.18

Lemma 4.3.18 is the assertion that the action of adding extracted acceptance types to the acceptance environment in the context preserves consistency. The proof is structured in a very similar way to the rejection analogue of Lemma 4.3.13 except that there is no requirement for the constructed justifications to be finite, so things are slightly simplified. Hence, the proof is broken into two main parts. The first shows that the environment extracted from the region types every configuration in the region. The second part shows that therefore the environment is itself consistent. The proof is quite close to a proof by Kobayashi in Appendix B of [Kobayashi, 2013], the present argument is slightly more complicated due to the fact that we must take care of accepting leaves. In preparation for the proof of the lemma, it is helpful to point out some simple consequences of the definition of acceptance type extraction in certain local situations.

The situation for variables.

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

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

Proof. By definition, since in each case the prefix is applied to the same arguments. □

The situation for non-terminals.

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

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

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

Proof. By rule (A4), since necessarily $y_i \mapsto s_i \in B_C$ and further by Proposition 4.5.6. □

Since we have carefully chosen the closure conditions that define the acceptance region, the proof of Lemma 4.3.18 can follow the same format as the proof of Kobayashi for type extraction from the (infinite) concrete configuration graph. We first prove that the environment extracted is sufficient to type all the prefixes of the vertices that it is extracted from. First let us define an auxiliary environment which extends $\mathsf{env}_\mathsf{A}(C)$ by retaining the variable typings which were disregarded according to the definition in the text:

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

letting x range over variables or non-terminals.

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

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

Proof. By induction on s.

- When s is a terminal a, then v is of the form $(a\, s_1 \cdots s_n, q)$ and, by (A2), there is some satisfying assignment S to $\delta(q, a)$ and a child $v' = \{(s_i, q') \mid (i, q') \in S\}$. But then by (A3), for each $(i, q') \in S$, $(s_i, q') \in \mathsf{RA}(C)$ (*). By definition $\mathsf{extr}(v, a)$ is of the form:
$$\bigwedge S_1 \wedge \bigwedge T_1 \to \cdots \to \bigwedge S_n \wedge \bigwedge T_n \to q$$
where, for each $i \in [1..n]$, S_i is the set $\mathbb{T}(\Gamma_\exists)(s_i)$ and T_i is the set of all types $\mathsf{extr}_A(\Gamma)(v'', s_i)$ such that (v'', s_i) is prefix in $\mathsf{RA}(C)$. Notice that, by (*), for each $(i, q') \in S$, $q' \in T_i$. Consequently:
$$\bigcup_{i=1}^{n} \{(i, q) \mid q \in S_i \cup T_i\} \models \delta(q, a)$$
and the result follows from (T-Cst).

- When s is a non-terminal F, then v is of the form $(F\, s_1 \cdots s_n, q)$. By definition $\mathsf{extr}(v, F) \in \mathsf{env}^+_\mathsf{A}(C)(F)$ and the result follows from (T-Fun).

- When s is a variable y, then v is of the form $(y\, s_1 \cdots s_n, q)$. By definition $\mathsf{extr}_A(\Gamma)(v, y) \in \mathsf{env}^+_\mathsf{A}(C)(y)$ and the result follows from (T-Var).

- When s is an application $t_1\, t_2$ then (v, t_1) is also a prefix and so it follows from the induction hypothesis that $\mathsf{env}^+_\mathsf{A}(C) \vdash_\mathcal{A} t_1 : \mathsf{extr}(v, t_1)$. By definition $\mathsf{extr}(v, t_1)$ is of the form:
$$\bigwedge \mathbb{T}(\Gamma_\exists)(t_2) \;\wedge \bigwedge_{v' \in W} \mathsf{extr}(v', t_2) \quad\to\quad \mathsf{extr}(v, t_1\, t_2)$$
with W the set of all configurations that have t_2 as a prefix. For each such v', since (v', t_2) is a prefix, it follows from the induction hypothesis that $\mathsf{env}^+_\mathsf{A}(C) \vdash_\mathcal{A} t_2 : \mathsf{extr}(v', t_2)$. The result follows from (T-App).

□

We can now conclude with the proof of Lemma 4.3.18. The lemma is stated as follows:

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

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

(i) If v is an accepting leaf then, by definition, it must be that there is some choice of typed variables $\{y_i \mid i \in [1..n]\}$ such that, for all $i \in [1..n]$, $\Gamma_\exists(y_i) = \bigwedge \mathbb{T}(\Gamma_\exists)(s_i)$ and also $\Gamma_\exists \vdash t[y_1/x_1, \ldots, y_n/x_n] : q$. By definition then, $\mathsf{extr}(v, F)$ is exactly:
$$\bigwedge \mathbb{T}(\Gamma_\exists)(s_1) \to \cdots \to \bigwedge \mathbb{T}(\Gamma_\exists)(s_n) \to q$$
and the result follows by a renaming of variables.

(ii) Otherwise, by (A1), v has a child v' that is of the form $(t[y_1/x_1, \ldots, y_n/x_n], q)$ and it follows from Lemma 4.5.8 that $\mathsf{env}^+_{\mathsf{A}}(C) \vdash_{\mathcal{A}} t[y_1/x_1, \ldots, y_n/x_n] : q$. Now, for each $i \in [1..n]$, it follows from Proposition 4.5.7 that in this situation,

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

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

□

Proof of Theorem 4.3.20

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

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

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

Proof. By induction on the kind κ.

- If $\kappa = o$, then v is of the form (t, q) and $\mathsf{extr}(v, s) = q$. By definition $q :: o$.
- If κ is an arrow kind $\kappa_1 \to \kappa_2$, then v is a configuration $(s t_1 \cdots t_n, q)$ and $\Delta \vdash t_1 : \kappa_1$ and $\Delta \vdash s\, t_1 : \kappa_2$. By definition $\mathsf{extr}(v, s)$ is of the form $\bigwedge T \to \mathsf{extr}(v, s\, t_1)$. It follows from the induction hypothesis that $\mathsf{extr}(v, s\, t_1) :: \kappa_2$ (*). Now let $\tau \in T$ and consider cases. If $\tau \in \mathbb{T}(\Gamma_\exists)(t_1)$ then, by assumption and Lemma 3.2.2, $\tau :: \kappa_1$. Otherwise, there is a prefix (v', t_1) such that $\tau = \mathsf{extr}(v', t_1)$ and it follows from the induction hypothesis that $\tau :: \kappa_1$. Hence $\bigwedge T :: \kappa_1$ and with (*) the result follows.

□

It follows from Lemma 4.5.1 that every such prefix will have a kind consistent with $\mathcal{N} \cup \mathsf{K}$. Hence one can show by induction that every $\Gamma_\exists$ featured in a context is a refinement of $\mathcal{N} \cup \mathsf{K}$. The same can be said of each $\Gamma_\forall$, since:

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

Proof. By induction on T. If T is concluded by any of (M1), (M3) or (M5) then the new type is a refinement of the kind of F or y as a result of Lemma 3.2.2 and Lemma 4.5.1 and the well kindedness of the rest of the environment follows from the induction hypothesis. If T is concluded by (R4), then observe that the new type is well-kinded by definition and the rest of the environment is well kinded as a result of the induction hypothesis. Otherwise T is concluded by (M2), there is no new type extracted and so the result simply follows from the induction hypothesis. □

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

Proof. Since every type extracted from the acceptance region is well kinded according to $\mathcal{N} \cup \mathsf{K}$ (Lemma 4.5.9) and every type extracted from the rejection region is well kinded according to $\mathcal{N} \cup \mathsf{K}$ (Lemma 4.5.10), it follows that for each $i \in \mathbb{N}$, $\Gamma_{\forall i} :: \mathcal{N} \cup \mathsf{K}$ and $\Gamma_{\exists i} :: \mathcal{N} \cup \mathsf{K}$. Since there are only finitely many type environments that refine $\mathcal{N} \cup \mathsf{K}$, there is some $i \in \mathbb{N}$ such that $C_i = C_{i+1}$. □

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

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

Proof. By induction on i.

- If $i = 0$, then $\Gamma_\exists^0$ is trivially consistent and $\Gamma_\forall^0$ is trivially sound.
- If $i = k+1$, then is follows from the induction hypothesis that $\Gamma_\exists^k$ is $(\mathcal{G}, \mathcal{A})$-consistent and $\Gamma_\forall^k$ is $(\mathcal{G}, \mathcal{A}^c)$-co-consistent. The result follows from Lemma 4.3.18 and Lemma 4.3.13 respectively.

□

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

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

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

Proof. Assume for contradiction that $\mathsf{ACG}(C) \neq \varnothing$. Then, by construction, necessarily $(S, q_0) \in V_C$ and hence (S, q_0) is C-unknown whence $q_0 \notin \Gamma_\exists(S)$. We distinguish two cases. In case $\mathsf{ACG}(C)$ contains some rejecting leaf then, by Lemma 4.3.14, there is some new type introduced into $\mathsf{env_R}(C)$ which contradicts the stability of C. Otherwise $\mathsf{ACG}(C)$ contains no rejecting leaves, but then, by Lemma 4.3.16, $\mathsf{RA}(C) = \mathsf{ACG}(C)$ and so $(S, q_0) \in \mathsf{RA}(C)$. But, by Lemma 4.5.8, $\mathsf{env_A}(C) \vdash_{\mathcal{A}} S : q_0$, so $q_o \in \mathsf{env_A}(C)$ contradicting stability of C. ☐

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

4.5.14 Lemma. *For any $\mathcal{G}$, $\mathcal{A}$ and type sig. compatible with $\mathcal{A}$, the algorithm terminates and:*

- *Answer* YES *implies $\Gamma_\exists$ is a $\mathcal{G}$-consistent with $\Gamma :: \mathcal{N} \cup \mathsf{K}$ and $\Gamma_\exists \vdash S : q_0$.*
- *Answer* NO *implies $\Gamma_\forall$ is a $\mathcal{G}$-co-consistent with $\Gamma :: \mathcal{N} \cup \mathsf{K}$ and $\Gamma_\forall \vdash S : q_0$.*

Proof. Since every type context C yields an ACG which is finite (Lemma 4.3.8), and, by definition, the corresponding rejecting and accepting regions are subgraphs of the ACG, it follows that $\mathsf{env_R}(C)$ and $\mathsf{env_A}(C)$ can be computed effectively. Since the sequence of type contexts must eventually stabilize to some context $\langle \Gamma_\exists, \Gamma_\forall \rangle$ (Lemma 4.5.11), it follows that the algorithm terminates. Notice that, a consequence of Lemma 4.5.13 and (G1) is that necessarily either $q_0 \in \Gamma_\exists(S)$ or $q_0 \in \Gamma_\forall(S)$. If the answer is yes, then necessarily $q_0 \in \Gamma_\exists(S)$, by Lemma 4.5.12 $\Gamma_\exists$ is consistent and by Lemma 4.5.9 $\Gamma_\exists :: \mathcal{N} \cup \mathsf{K}$. Otherwise $q_0 \in \Gamma_\forall(S)$, by Lemma 4.5.12 $\Gamma_\forall$ is co-consistent and by Lemma 4.5.10 $\Gamma_\forall :: \mathcal{N} \cup \mathsf{K}$. ☐

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

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

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

Proof. By Lemma 4.5.14 then Lemma 3.5.4 and Lemma 3.5.5 respectively. ☐

4.6 PREFACE: a higher-order model checker

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

Implementation

To ensure efficiency we have taken a number of decisions about how to code specific aspects of the algorithm which deviate from the presentation. Rather than constructing them as part of initialisation, we build the maps A and R lazily, adding bindings as they are needed by applications of rule (G2). Further, the implementation uses a flow analysis with increased accuracy, distinguishing between instances of arguments using not just the triple of acceptance type, rejection type and kind, but additionally the formal parameter and the state component of the calling configuration. Since intersection type checking is frequently invoked as part of the decision procedure, we aim to ensure it is done as efficiently as possible. Hence, we omit subtype checking[1] and hash cons the intersection types. Finally, rather than compute all possible witnessing trees for any given vertex $v \in \mathsf{RR}(C)$, we take one representative, which is a function of the construction of the region.

Evaluation

We have evaluated the tool on the, by now, large collection of recursion scheme model checking instances found online and in the related literature. We have compared the performance of our tool with that of all other recursion scheme model checkers. The full listing of results is available in the appendix of this dissertation, here we aim to present a small representative sample in order to describe the general trends.

We have picked a number of benchmarks from three categories, which are displayed in Tables 4.2, 4.3 and 4.4 respectively. The benchmarks were run on an Intel Xeon machine with 12GB of RAM and 4 cores running at 2.4GHz, limiting the run-time of each tool on each benchmark to 2 minutes. In all cases the first four columns are, respectively, the name of the benchmark, the number of rules (equations), the order of the scheme, whether the tree is accepted (A) or rejected (R) by the property automaton. The remaining columns list the time taken by each tool from start to finish, which is either given in seconds, or marked "–" in case the tool ran out of time or some other resource.

Category 1. The first category consists of model checking instances that have arisen from OCaml verification problems via the predicate abstraction tool MoCHi [Sato et al., 2013]. Although MoCHi can solve many complex examples, the scalability of a full blown predicate abstraction tool for higher-order programs is still an open topic of research. Consequently the problem instances derived from this tool are quite small: mostly less than 100 rules. MoCHi generates a mild extension of recursion schemes called RSFD [Kobayashi et al., 2010], which are currently only supported by Preface and TRecS. Our tool Preface can typically solve each of these instances in less than 0.5 seconds, but this is already roughly an order of magnitude slower than TRecS. However, the overhead incurred by JIT compilation on Mono is a major factor; when compiled ahead of time on Windows, the time taken by Preface to solve these instances is typically less than 0.05

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

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

Table 4.2: Benchmarks of category 1.

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

Table 4.3: Benchmarks of category 2.

Benchmark	Rules	Ord	Dec	Preface	HorSat	HorSatT	C-SHORe	GTRecS	TravMC	TRecS
exp2-1600	1606	2	A	8.39	–	–	–	10.47	–	–
exp2-3200	3206	2	A	17.51	–	–	–	59.13	–	–
exp2-6400	6406	2	A	39.58	–	–	–	–	–	–
exp2-12800	12806	2	A	92.19	–	–	–	–	–	–
exp4-400	408	4	A	14.12	–	106.53	–	–	–	–
exp4-800	808	4	A	30.55	–	–	–	–	–	–
exp4-1600	1608	4	A	71.06	–	–	–	–	–	–
exp4-3200	3208	4	A	–	–	–	–	–	–	–

Table 4.4: Benchmarks of category 3.

seconds, although usually still slower than TRecS. As the benchmarks become slightly larger, towards the bottom of the table, the time taken by TRecS starts to lag behind the time taken by Preface, which is the start of a general trend in the data to follow.

Category 2. The second category consists of instances arising from a tool for performing exact flow analysis [Tobita et al., 2012]. These examples are significantly larger than those of Category 1 and, indeed, form some of the largest instances on which HORS model checkers have been evaluated in the literature as of the time of writing. Although they have fewer than 1000 rules each, due to the nature of the verification algorithm that produces them, they have high order and very high maximum arity, with `cfa-life2` being order 14 and containing arity 29 functions. Consequently, many of the tools have difficulty, but among those that are able to solve these instances, the trend observed in the Category 1 examples can be seen to continue.

Category 3. The final category consists of instances of a family of schemes due to Kobayashi [2011]. This family of instances was designed to be deliberately difficult for bounded model checking style algorithms such as the hybrid algorithm of TRecS, whilst simultaneously being a good indicator of the scalability of Kobayashi's linear time algorithm as implemented in GTRecS. Although these schemes are not "real" in the sense of arising from program analysis problems they seem a good measure of scalability since they generate hyper-exponentially sized trees (and hence use the full power of higher-order schemes), their certificates are proportional to the number of rules and they can push the model checkers much further since the family contains much larger schemes than can be produced by current verification tools. The first half of the table shows instances which are generated with order-2 schemes and the second half shows instances with order-4 schemes. The size of the schemes roughly doubles within each half by row. As expected, GTRecS does a good job at solving even relatively large examples at order-2, though it has some difficulty at higher-orders. Preface does even better and, in contrast to the other tools, can be seen to solve these examples in time which is roughly linear in the number of rules.

Analysis

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

However, although the refinement will always eventually compensate for these particular limitations of the CFA, it is possible to construct instances in which the number of iterations required is unacceptably large with respect to the characteristics of the instance. Although such examples do not seem to occur in the corpus of instances drawn from the higher-order model checking literature and associated verification tools, we have been able to construct very small and simple schemes which exhibit this bad behaviour. The examples in Table 4.5 are based on a family of Boolean programs defined by Ball and Rajamani [2000]. Each of these instances is first order and consists of a few hundred rules, but the property automaton is strictly alternating. We record the number of iterations (Rnds) and the time (in seconds) taken by PREFACE and an extension, PREFACE$^+$, in the remaining columns.

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

Table 4.5: Bad behaviour.

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

This analysis suggests that not enough type information is being recovered from the ACG at each iteration and, indeed, by extending our implementation with heuristics for extracting more types, we have solved these examples more quickly. Our extension, labelled PREFACE$^+$ in the table, exploits the *incremental* nature of the algorithm. By this we mean the following characteristics:

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

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

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

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

This extension appears to work well to solve the examples in Table 4.5, cutting the time taken to process `t1600` down from almost 3 hours to under one minute! However, it seems unlikely to scale well to higher-orders, where the possible number of types from which to guess is much larger. Consequently, we do not consider this a satisfactory solution and leave to future work a proper treatment of this problem.

Note however that this particular class of examples is not difficult for all higher-order model checkers. Since all the examples are first order, the space of types is relatively small and we expect that algorithms that perform a search of this space more directly ought to perform very well. This is certainly the case for the tool HorSat, which gives the following results (reported by Kobayashi and Terao[2]):

Benchmark	t100	t300	t500	t700
HorSat	0.26	1.00	2.05	3.77

4.7 Related work

We refer the reader to Section 2.4 for an overview of the existing algorithms that solve the same problem. Since we have already compared implementations in Section 4.6, we will comment on the relatedness of the algorithms themselves.

The TRecS algorithm. The hybrid algorithm of Kobayashi [2009b] is the most clearly related to our algorithm. Both take an intersection type assignment approach to solving the model checking problem and both operate in a kind of abstraction-refinement loop constructing approximations of the (infinite) configuration graph of the model checking instance.

However, within the constraints of this description they are very different. The most important of the differences is that, whilst our algorithm constructs finite *over*-approximations of the configuration graph using a kind of CFA-based representation, the

[2]Personal communication.

hybrid algorithm constructs finite *under*-approximations by evaluating the scheme a finite number of steps. In the under-approximation constructed by the hybrid algorithm there is therefore no corresponding notion of rejecting region (since the portion of the graph that has been so far explored is completely accurate), nor is there any guarantee that there exists any accepting region (in the sense of a subgraph from which accepting region type extraction is guaranteed to produce consistent type assignments). Despite this, our notion of type extraction from the accepting region is essentially the same as the notion of type extraction in the hybrid algorithm. Consequently, the types extracted by the hybrid algorithm may not be consistent and so the algorithm explicitly filters those out as an extra fixed point computation at the end of each round. Furthermore, completeness is only guaranteed by non-deterministically guessing additional types, based on those that were already extracted. By contrast, we have described a way to identify the rejecting and accepting subgraphs of the over-approximation of the configuration graph constructed by our algorithm, from which all types extracted are guaranteed to be (co-)consistent. An important feature of the over-approximations constructed by our algorithm is that they are constructed in such a way as to be bounded in size by a polynomial function of the size of the scheme; there is no such guarantee associated with the under-approximations constructed by the hybrid algorithm. An advantage of the under-approximations constructed by the hybrid algorithm is that the extraction of counter-examples, which we have not explored in this work, is immediate. Also, for instances whose behaviour can be characterised after only a small number of reductions (for example, because the instance is small), the algorithm performs very well, as borne out by the comparison of the tools in Section 4.6. Our construction of an over-approximation and consequent introduction of spurious behaviours motivated the use of rejection typing, which is essential to the termination of the algorithm; no such notion is needed by the hybrid algorithm. Finally, we note that our algorithm is able to check alternating (co-)trivial properties of recursion schemes, whereas the hybrid algorithm is restricted to checking deterministic trivial properties.

The TravMC algorithm. Like the hybrid algorithm, the traversal based algorithm of Neatherway *et al* takes a similar, under-approximation approach and so many of the same comparisons apply. Unlike our algorithm (and the hybrid algorithm), the traversal based algorithm under-approximates the consistency derivation itself rather than the configuration graph of the problem instance. By doing this, it can ensure that it never generates an inconsistent type assignment. Like the hybrid algorithm, the algorithm is not known to be fixed parameter tractable. Also like the hybrid algorithm, the traversal based algorithm suffers from having to "guess" how best to refine the under-approximation. On each iteration these algorithms can only extend the current best under-approximation by a finite amount, but often some choices about where to extend can be dramatically better than others. Since the traversal based algorithm was designed for higher-order recursion schemes with cases, the corresponding type system is an extension of the systems considered by all the other higher-order model checking algorithms, incorporating union types at ground kind which makes type inference significantly more difficult. Not only is the space

of ground kind types exponentially larger but also, in order to type the case constant, the algorithm must infer *all possible* valid (ground kind) type assignments for the scrutinee, which is never a requirement in the more restricted, intersection-only system.

The GTRecS algorithm. Kobayashi's game semantics inspired algorithm was the first practical fixed-parameter polynomial time algorithm and, unlike our algorithm, is actually known to be linear in the size of the scheme. Linearity is obtained by working only with the scheme and not constructing any auxilliary structure, such as a configuration graph. Types are inferred based on a reading in terms of game semantics. By contrast, our algorithm builds over-approximations of the configuration graph of the problem instance, which are non-linear in the size of the scheme (though still polynomial), and uses the information gleaned in order to inform type extraction. We would argue that Kobayashi's algorithm is consequently less well directed in its search for types. This is bourne out by the empirical evaluation, which shows that the associated tool GTRecS often does not perform well on "real" problem instances.

The C-SHORe algorithm. The C-SHORe algorithm of Broadbent, Carayol, Hague and Serre is very different from our own and, indeed, all those so far discussed. The algorithm does not operate directly on recursion schemes but instead on collapsible pushdown automata (CPDA). There is a translation from recursion schemes to CPDA but, of course, the resulting automata are certainly more complicated as a result of passing through it, which makes direct empirical comparisons more difficult. The algorithm avoids intersection types entirely and uses an automaton saturation technique in the same vein as that pioneered by Bouajjani, Esparza and Maler for pushdown systems [Bouajjani et al., 1997]. Furthermore, it takes a *backwards* approach to property checking, in the sense of starting with error configurations and successively inferring larger and larger sets of configurations which can reach them. It is therefore able to check alternating (co-)trivial properties and is, moreover, fixed parameter tractable. In contrast, our algorithm (and all those discussed so far) work in a forwards direction, starting from the initial configuration and successively inferring larger and larger sets of configurations that can be reached. To make the algorithm more practical, the implementation C-SHORe performs a kind of control flow analysis as a first step but, unlike our algorithm, it is not integrated into an abstraction refinement loop.

The HorSat algorithm. Kobayashi and Broadbent's saturation algorithm, as implemented in HorSat and HorSatT, is a very successful attempt to transfer the saturation paradigm from automata to intersection types. As such, it is unlike our algorithm as it propagates information backwards from the error configurations. In this case, error configurations are represented by certain types, called "co-trivial types", which follows the same idea as our "rejection types" (our algorithm and theirs were developed independently). Unlike our algorithm they do not also use types for acceptance, and hence there is no associated interplay. Like the C-SHORe algorithm, they employ a kind of flow analysis

to make an initial pruning of the search space but, like the C-SHORe algorithm, it is not integrated into any abstraction refinement loop.

Abstraction refinement. Counterexample-guided abstraction refinement (CEGAR) was formalised by Clarke, Grumberg, Jha, Lu, and Veith [2000] for symbolic model checking. The CEGAR loop was first applied to higher-order model checking by Ong and Ramsay [2011] and, separately, by Kobayashi, Sato, and Unno [2011]. The former addresses the undecidable problem of verifying safety properties of pattern-matching recursion schemes, using patterns to abstract properties. The latter is used in conjunction with predicate abstraction to verify simply-typed functional programs generated from infinite data domains such as integers. In both cases the abstraction refinement loop is at a higher-level than model checking, which is a black-box component. In contrast, Preface builds successively more accurate finite abstractions of the configuration graph of a model checking instance, from which potential certificates of acceptance and of rejection are derived.

Type-based flow analysis. Flow analyses were first applied to untyped languages. Jagannathan, Weeks, and Wright [1997] introduced a type-directed, polyvariant flow analysis for the predicative subset of System F, which can leverage types to analyse programs more precisely. Our algorithm uses an intersection type system for describing automaton definable properties and necessarily works in a situation in which not all type information is known; in contrast, theirs uses more standard typing (System F) and starts from a situation in which all types are known. Thus their analysis is comparable to a single iteration of our algorithm where the associated context already contains all the possible correct type information. The property of *respecting types*, which is put forward by the authors as a measure of the appropriateness of a CFA for a typed language is, for us, actually an essential technical requirement in order to extract new, valid type information.

Plevyak and Chien [1994] considered a constraint-based type inference for object-oriented programs. Like our algorithm, not all type information is known at the start, and they compute types iteratively based on a flow analysis. However, within a single iteration they do not distinguish based on type information, instead they distinguish based on clashes discovered in the previous iteration. A problem with this approach of distinguishing calls unconditionally is that there may be infinitely many counterexamples which are being distinguished one at a time. Consequently, this can lead to non-termination, and so their method cannot handle recursion in general.

Chapter **5**

Intersection Types as Exact Abstract Interpretations

5.1 Introduction

What is special about higher-order model checking that it can be completely captured by a simple intersection type system? The system has a finite, discrete set of atoms, which we happen to name by the states of the automaton (though, of course, any name will do), there are no special type constructors (save for the rather general intersection operator), and there is exactly one rule for each clause of the syntax; but despite all this ordinariness, the system is related to the model checking problem for recursion schemes in a particularly strong way. The answer to this question then, must lie in the way that the types which are assigned to the constants *faithfully* represent the behaviour of tree constructors, as far as the property automaton can tell. The study of this phenomenon is the subject of this chapter.

We will present a very natural condition under which an assignment of intersection types to a given set of constants — whether they represent tree constructors, natural numbers, conditionals or some language feature entirely more exotic — is sufficient to completely characterise safety property checking problems concerning arbitrary recursion schemes built over the constants as problems of type assignment in a type system induced by the assignment. Since we will restrict our attention to intersection refinement type systems, an immediate corollary is the decidability of such problems. The safety properties that we consider are quite general and defined independently of any particular representation such as trivial automata. Since we will also allow the consideration of higher-type properties, which are sets of *functions*, we will leave behind the traditional territory of model checking, which is usually restricted to ground type properties such as sets of *trees*. We will show, by example, how the resulting theory can be straightforwardly and fruitfully applied to computational situations that are both familiar to higher-order model checking and apart from it. Furthermore, despite the increased generality, we shall show that, all ground type property checking problems that are characterised in this way

can nevertheless be reduced to higher-order model checking, bringing them into the scope of efficient algorithms such as that in Chapter 4.

By way of a very simple example, consider the class of Boolean recursion schemes in which the ground kind o is intended to represent the Booleans. Such schemes are built over a signature consisting of two zero-arity constants `true` and `false` and an arity three constant `if` whose arguments are intended to represent the condition, the true branch and the false branch respectively. The interpretation of the constants is the standard one. Now consider an intersection type system over the two base type atoms `t` and `f` to represent truth and falsity respectively. It seems quite plausible that the following type, when assigned to the constant `if`:

$$\begin{aligned} \sigma_{\texttt{if}} &= (\mathsf{t} \to \mathsf{t} \to \top \to \mathsf{t}) \wedge (\mathsf{t} \to \mathsf{f} \to \top \to \mathsf{f}) \\ &\wedge \; (\mathsf{f} \to \top \to \mathsf{t} \to \mathsf{t}) \wedge (\mathsf{f} \to \top \to \mathsf{f} \to \mathsf{f}) \\ &\wedge \; (\mathsf{t} \wedge \mathsf{f} \to \top \to \top \to \mathsf{t}) \wedge (\mathsf{t} \wedge \mathsf{f} \to \top \to \top \to \mathsf{f}) \end{aligned}$$

should completely characterise the behaviour of this constant, assuming `t` is assigned to the constant `true` and `f` is assigned to the constant `false`. The first and second pair of conjuncts in the type represent the behaviour when the condition is true and false respectively, namely that whatever the value of the corresponding branch, that should be the value of the expression. The final pair of conjuncts represent the behaviour when the condition is somehow inconsistent, in which case the type of the expression as a whole can be shown to be anything. The intuition that this type characterises the constant is made precise by proving the following assertion:

$$\alpha(\eta([\![\texttt{if}]\!](b)(b_1)(b_2))) = \sigma_{\texttt{if}} \cdot \alpha(\eta(b)) \cdot \alpha(\eta(b_1)) \cdot \alpha(\eta(b_2))$$

which will be explained in detail in what follows but which says, informally, that the most precise intersection type that can be assigned to the result of applying `if` to three Booleans is the same as the intersection type obtained by applying the type we assigned to `if`, $\sigma_{\texttt{if}}$, to the most precise types that can be given to the three Booleans. In other words, no type information is lost by computing with `if` *entirely* using types.

More precisely and in the parlance of abstract interpretation (which is the setting for this work), the statement asserts that our choice of abstract interpretation of the constant is *exact*. The general result of this work is a guarantee that given any signature and any intersection type theory, if there is an exact abstraction of the constants of the former in terms of the intersection types of the latter then, for any recursion scheme built over those constants, all queries of the form $[\![t]\!] \in \gamma(\tau)$ will be effectively soluble in the corresponding intersection refinement type system. In other words, type assignment will characterise the property checking problem for all terms built over the given constants using higher-order, recursive functions. Higher-order model checking will be recovered in case the signature is a simple collection of tree constructors, the intersection type language is over the states of the given property automaton and the only query of interest has t the ground kind non-terminal symbol S and τ the base type q_0.

An immediate and relevant application is to characterising the language constructs used to extend recursion schemes, which we surveyed at the end of Section 2.1. These extensions, involving if statements, case analysis constructs and non-determinism which, when first introduced, each required a special variation of the intersection type system with their own *global* proofs of soundness and completeness, are here justified by much simpler *local* proofs of exactness with respect to the constants involved. Furthermore, since, as we shall show, all the *ground* queries (in which both the term t and the type τ are ground) — irrespective of the particulars of the signature, its interpretation, the intersection type theory and the properties it specifies — can be reduced to the trivial automaton model checking problem for recursion schemes, it follows that all can be solved using the algorithm of the previous chapter.

This chapter is structured as follows. In Section 5.2 we will describe the way in which a denotational semantics of domains and continuous functions can be ascribed to any given signature and the way in which a semantic interpretation can be given to intersection refinement types as certain subsets of a given domain. Here we will define the general setting, in which one has a term language defined by an interpreted signature and a property language defined by an interpreted intersection type theory; and the associated property checking problem. In Section 5.3 we bring the situation into the scope of abstract interpretation by defining the concrete properties of interest, which are safety properties, the abstract properties, which are intersection types and the relationship between concrete and abstract properties, which is one of Galois correspondence. In Section 5.4 we will state and prove a restriction of our main theorem for the case of ground queries only, and discuss its consequences. We also show that all such ground type queries are reducible to higher-order model checking problems. In Section 5.5 we will apply the theorem to give new, short proofs of the intersection type characterisations of the main features of the extensions of higher-order model checking, in terms of the constants they introduce. Then, in Section 5.6, we will prove the theorem for higher-types, thus allowing us to look beyond traditional model checking by stating and proving properties of (higher-order) functions. Finally, in Section 5.7, we will discuss related work.

5.2 Term languages, property languages and queries

In this section we describe, using completely standard techniques, how signatures, recursion schemes and intersection type theories can be given a semantics. We are then able to make precise the setting of this work, in which recursion schemes define sets of functions in a given term language and queries can be formulated within a given class of properties, specified with respect to an interpretation of intersection types. We first present the notion of interpretation of terms and recursion schemes. We then show that the purely operational notion of value tree can be completely recovered by interpreting the constants as tree constructors, i.e. that the tree-constructor interpretation is *computationally adequate*. Next, we motivate and give a precise definition of safety property in this setting and we show how intersection type theories can be interpreted with respect to a given class of such safety properties. Finally we define the notions of term language, property language

and query, which allows us to specify a generalised version of the trivial automaton model checking problem for recursion schemes.

Interpretation of recursion schemes

In the standard account of higher-order model checking, which we have given in Section 2, the constants of the signature are ascribed no behaviour. Each is quite inert with respect to the behaviour of the scheme, which is defined operationally through a notion of reduction. An equivalent view of the situation is that each constant c *is* assigned behaviour, but that it behaves as a *tree constructor* — a function that simply glues together the trees given to it as input with c at the root. Since we are interested in introducing language features beyond simple tree constructors, we first describe, in a quite standard way, how to interpret a kinded language of terms semantically. Since we are ultimately interested in a characterisation of exact abstraction that depends only on the semantics of the particular constants in the language, it had better be that the constants are assigned an interpretation independently of the recursion schemes in which they are used. For this reason we will specify a denotational semantics, and the vehicle we choose is elementary domain theory.

5.2.1 Definition. (Domains and continuous functions) For us a *domain*, D, is a directed-complete partially ordered set with a least element $\bot$. That is to say $\bot \in D$ and, given any directed subset $E \subseteq D$, the limit $\bigsqcup E$ is itself an element of D. We will typically write the order on domains as $\sqsubseteq$. In this context we say that a function f between domains D_1 and D_2 is *continuous* just if, for any directed subset $E \subseteq D_1$, $f(\bigsqcup E) = \bigsqcup\{f(e) \mid e \in E\}$. Given domains D_1 and D_2, we shall write as $D_1 \Rightarrow D_2$ the set of all continuous functions between the two.

The semantics we present will be parametrised by the specific choice of interpretation of the ground kind o and the choice of interpretation of the constants. In general, the kinds are interpreted by domains and the terms by their elements.

5.2.2 Definition (Kind interpretation)**.** An interpretation for the kinds over o is given by a choice of a domain D_o for the base kind o. This choice is then extended over general kinds recursively:

$$\llbracket o \rrbracket = D_o$$
$$\llbracket \kappa_1 \to \kappa_2 \rrbracket = \llbracket \kappa_1 \rrbracket \Rightarrow \llbracket \kappa_2 \rrbracket$$

5.2.3 Definition (Kinded term interpretation)**.** An interpretation for a signature Σ is an interpretation for the kinds D_o and a choice of continuous function $d_c \in \llbracket \mathsf{kind}(c) \rrbracket$ for each constant $c \in \Sigma$. Then the choice is extended to each well-kinded term $\Delta \vdash t : \kappa$ by assigning a continuous function:

$$\llbracket \Delta \vdash t : \kappa \rrbracket \in (\Pi_{\xi \in \mathsf{dom}(\Delta)} \llbracket \Delta(\xi) \rrbracket) \Rightarrow \llbracket \kappa \rrbracket$$

That is, a function which takes an interpretation of the free function symbols and variables in t and returns an interpretation of t. We call an interpretation of the free function symbols and variables a *semantic environment* and write a typical instance as Ξ. The interpretation of a kinded term is defined inductively over the kind derivation:

$$\begin{aligned}
[\![\Delta \vdash c\colon \kappa]\!](\Xi) &= d_c \\
[\![\Delta \vdash x\colon \kappa]\!](\Xi) &= \Xi(x) \\
[\![\Delta \vdash F\colon \kappa]\!](\Xi) &= \Xi(F) \\
[\![\Delta \vdash t_0\, t_1 \colon \kappa_2]\!](\Xi) &= [\![\Delta \vdash t_0 \colon \kappa_1 \to \kappa_2]\!](\Xi)([\![\Delta \vdash t_1 \colon \kappa_1]\!](\Xi)) \\
[\![\Delta \vdash \lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.\, s\colon \kappa]\!](\Xi)(d_1)\cdots(d_n) &= [\![\Delta, x_1\colon \kappa_1, \ldots, x_n\colon \kappa_n \vdash s\colon o]\!](\Xi \cup \{x_i \mapsto d_i\}_{i\in[1..n]})
\end{aligned}$$

with, in the final clause, the type κ standing as a short-hand for $\kappa_1 \to \cdots \to \kappa_n \to o$.

The free function symbols in any given term are interpreted with respect to an environment, which assigns to each of them a particular element of the appropriate domain. Since recursion schemes define the meaning of function symbols equationally, the interpretation of a given recursion scheme determines such an environment.

5.2.4 Definition (Scheme interpretation)**.** Given an interpretation for the signature, we interpret $\mathcal{G}$ as a semantic environment $[\![\mathcal{G}]\!] \in \Pi_{F \in \mathsf{dom}(\mathcal{N})} [\![\mathcal{N}(F)]\!]$ defined so as to map the non-terminals in $\mathcal{N}$ to the interpretations of their definitions. Hence, we define $[\![\mathcal{G}]\!]$ as the least fixed point $\mu\mathsf{Step}$ of the functional Step, which is given by:

$$\mathsf{Step}(\Xi)(F) = [\![\mathcal{N} \vdash \mathcal{R}(F) : \mathcal{N}(F)]\!](\Xi)$$

Notational convention. When in the context of a fixed scheme $\mathcal{G}$, we will feel free to elide any mention of data which is uniquely determined by $\mathcal{G}$ when writing interpretations of terms. In particular, for each closed term $\mathcal{N} \vdash t : \kappa$ we will write $[\![t]\!]$ for the function $[\![\mathcal{N} \vdash t : \kappa]\!]([\![\mathcal{G}]\!])$ since all such extra information is clear from the context.

The interpretation of a scheme $\mathcal{G}$ is, in a precise sense, a solution to the equations in $\mathcal{G}$. In particular, for each $F \in \mathsf{dom}(\mathcal{N})$, $[\![\mathcal{G}]\!](F)$ solves the equation $F = \mathcal{R}(F)$ in $[\![\mathcal{N}(F)]\!]$, in the sense that $[\![F]\!] = [\![\mathcal{G}]\!](F) = [\![\mathcal{R}(F)]\!]$ is an identity. Since the reduction relation of a scheme is defined entirely in terms of the equations, and since application is interpreted simply as function application, it should come as no surprise that any such interpretation is a sound model of reduction. The proof is quite routine; we first observe that substitution is modelled correctly, then a simple induction shows the result.

5.2.5 Proposition (Model of substitution)**.** *Fix a term signature and its interpretation.*

$$[\![\Delta \vdash s[t/x] : \kappa_2]\!](\Xi) = [\![\Delta,\, x : \kappa_1 \vdash s : \kappa_2]\!](\Xi \cup \{x \mapsto [\![\Delta \vdash t : \kappa_1]\!](\Xi)\})$$

Proof. Follows since every such interpretation is a continuous type frame, see e.g. [Barendregt et al., 2013]. □

5.2.6 Lemma (Model of reduction)**.** *Fix a term signature and its interpretation. In any recursion scheme over the signature, for all closed terms s and t:*

$$s \rhd_w t \quad \textit{implies} \quad [\![s]\!] = [\![t]\!]$$

Proof. By induction on the proof of the reduction. If the reduction is by contracting a weak redex then, necessarily there is some non-terminal F of kind $\kappa_1 \to \cdots \to \kappa_n \to o$ with $\mathcal{R}(F) = \lambda x_1 \cdots x_n.\, r$, s has shape $F\, s_1 \cdots s_n$ and $t = r[s_1/x_1, \ldots, s_n/x_n]$. We reason equationally:

$$\begin{aligned}
[\![s]\!] &= [\![\mathcal{G}]\!](F)([\![s_1]\!]) \cdots ([\![s_n]\!]) \\
&= [\![\lambda x_1 \cdots x_n.\, r]\!]([\![s_1]\!]) \cdots ([\![s_n]\!]) \\
&= [\![\mathcal{N}, x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash r : o]\!]([\![\mathcal{G}]\!] \cup \{x_i \mapsto [\![s_i]\!] \mid i \in [1..n]\}) \\
&= [\![\mathcal{N} \vdash r[s_1/x_1, \ldots, s_n/x_n] : o]\!]([\![\mathcal{G}]\!]) \\
&= [\![r[s_1/x_1, \ldots, s_n/x_n]]\!]
\end{aligned}$$

where the penultimate equation follows from Proposition 5.2.5. If the reduction is by contracting a redex on the left then s is of shape $u\, v$ and necessarily there is some term u' such that $u \rhd_w u'$ and $t = u'\, v$. It follows from the induction hypothesis that $[\![u]\!] = [\![u']\!]$, from which one can reason that $[\![u\ v]\!] = [\![u]\!]([\![v]\!]) = [\![u']\!]([\![v]\!]) = [\![u'\ v]\!]$. The case of contracting a redex on the right of an application is similar. □

We now show, first by way of an example, how the notion of tree generating recursion scheme used in higher-order model checking can be recovered by interpreting o as the set of all finite and infinite $\Sigma^\perp$-labelled trees and the constants as tree constructors.

5.2.7 Example. *Consider the recursion scheme $\mathcal{G}$ from Example 2.1.6, which is defined by the equations:*

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

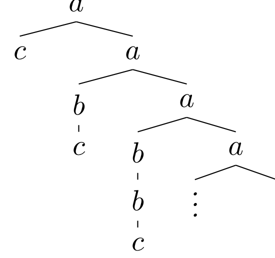
The terminal symbols are interpreted as tree constructors $\underline{a}$, $\underline{b}$ and $\underline{c}$ over the domain of finite and infinite $\Sigma^\perp$-ranked and labelled trees (recall Definition 2.1.20). We define, for each terminal symbol f of arity n, the tree constructor $\underline{f}$ by:

$$\begin{aligned}
\underline{f}(T_1) \cdots (T_n)(\epsilon) &= f \\
\underline{f}(T_1) \cdots (T_n)(i \cdot w) &= T_i(w)
\end{aligned}$$

The scheme above determines a tree $[\![S]\!]$ and a function on trees $[\![F]\!]$ which together form the least solution to the following equations:

$$\llbracket S \rrbracket = \llbracket F \rrbracket(\underline{b})$$
$$\llbracket F \rrbracket(d) = \underline{a}(d)(\llbracket F \rrbracket(\underline{b}(d)))$$

This ensures that the interpretation $\llbracket S \rrbracket$ of S is an infinite, $\Sigma^{\perp}$-ranked and labelled tree, which is depicted to the right.



That this method in general yields a complete characterisation of the trees generated by higher-order recursion schemes is the subject of the following subsection.

Computational adequacy of the tree interpretation

We aim to prove the following theorem:

5.2.8 Theorem (Adequacy of the interpretation)**.** *Fix a recursion scheme and consider the interpretation of the signature by the tree-constructor semantics of Example 5.2.7. For all closed applicative terms t of ground kind, $\llbracket t \rrbracket = \mathsf{Tree}_w(t)$.*

Although the theorem will not form the basis of any of the work that follows, it is important because it demonstrates that the setting considered here subsumes that considered by higher-order model checking. To prove the theorem, we define a logical relation between elements of the model and terms.

5.2.9 Definition (Logical relation A)**.** Fix a recursion scheme and consider the interpretation of the signature by the tree-constructor semantics. Define a binary logical relation A^{κ}, indexed by kinds κ, between elements of $\llbracket \kappa \rrbracket$ and terms of kind κ by induction on κ.

- When $\kappa = o$, define $d\ A^{o}\ t$ iff $d = \mathsf{Tree}_{\beta}(t)$.
- When $\kappa = \kappa_1 \to \kappa_2$, define $f\ A^{\kappa_1 \to \kappa_2}\ t$ iff: for all $e \in \llbracket \kappa \rrbracket$, for all terms s of type $\kappa_1 \to \kappa_2$, $e\ A^{o}\ s$ implies $d(e)\ A^{\kappa_1 \to \kappa_2}\ t\ s$.

Finally, extend the relation to kind environments Δ by, for all semantic environments $\Xi :: \Delta$ and substitutions $\rho : \Delta$, $\Xi\ A^{\Delta}\ \rho$ iff, for all $\xi : \kappa \in \Delta$, $\Xi(\xi)\ A^{\kappa}\ \rho(\xi)$.

We take the completely standard approach of showing the associated "basic lemma" (or "fundamental theorem" depending on the author) which says that, whenever a semantic environment is related to a substitution, then any term interpreted with respect to the former is related to a term under the action of the latter.

5.2.10 Lemma (Basic lemma for A)**.** *Fix a recursion scheme and consider the interpretation of the signature by the tree-constructor semantics.*

$$\Xi\ A^{\Delta}\ \rho \quad \textit{implies} \quad \llbracket \Delta \vdash t : \kappa \rrbracket(\Xi)\ A^{\kappa}\ t[\rho]$$

Proof. By induction on the shape of t.

- When t is a variable or function symbol ξ, the result follows from the assumption.
- When t is a constant c then $[\![\Delta \vdash c : \kappa]\!](\Xi) = \underline{c}$ and $c[\rho] = c$. Now κ is in general of the form $\kappa_1 \to \cdots \to \kappa_n \to o$ with each $\kappa_i = o$ and let, for each such i, $e_i\ A^o\ s_i$ for some element e_i and term s. Then, by definition, $e_i = \mathsf{Tree}_\beta(s_i)$ and hence $\underline{c}(e_1)\cdots(e_n) = \mathsf{Tree}_\beta(c\,s_1\ \cdots\ s_n)$.
- When t is an application $u\,v$, then it follows from the induction hypothesis that $[\![\Delta \vdash u : \kappa_1 \to \kappa_2]\!](\Xi)\ A^{\kappa_1\to\kappa_2}\ u[\rho]$ and $[\![\Delta \vdash v : \kappa_1]\!](\Xi)\ A^{\kappa_1}\ v[\rho]$. Hence, the result follows by definition.
- When t is an abstraction $\lambda x_1 \cdots x_n.\, s$, then necessarily κ is of the form $\kappa_1 \to \cdots \to \kappa_n \to o$ so let $e_i\ A^{\kappa_i}\ s_i$ for each such i. Then $[\![\Delta \vdash \lambda x_1\ \cdots\ x_n.\, s : \kappa]\!](\Xi)(e_1)\cdots(e_n)$ is:
$$[\![\Delta \cup \{x_i : \kappa_i \mid i \in [1..n]\} \vdash s : o]\!](\Xi \cup \{x_i \mapsto e_i \mid i \in [1..n]\})$$
and furthermore $\mathsf{Tree}_\beta((\lambda x_1 \cdots x_n.\, s)[\rho] s_1 \cdots s_n) = \mathsf{Tree}_\beta(t[\rho \cup \{x_i \mapsto s_i \mid i \in [1..n]\}])$ assuming, wlog, that $\mathsf{dom}(\rho) \cap \{x_1, \ldots, x_n\} = \varnothing$. The result then follows from the induction hypothesis.

□

The basic lemma can be lifted to environments by pointwise relating the first n-computation steps of $\mu\mathsf{Step}$ and the n-fold substitution $\mathcal{R}^n = \underbrace{\mathcal{R} \circ \cdots \circ \mathcal{R}}_{n\text{-times}}$.

5.2.11 Lemma (A-Relatedness of environments)**.** *Fix a recursion scheme and consider the interpretation of the signature by the tree-constructor semantics of Example 5.2.7. For all $n \in \mathbb{N}$:*
$$\mathsf{Step}^n(\bot)\ A^{\mathcal{N}}\ \mathcal{R}^n$$

Proof. By induction on n. If $n = 0$ then let $F : \kappa \in \mathcal{N}$, say $\kappa = \kappa_1 \to \cdots \to \kappa_n \to o$. Observe that $\bot(F) = \bot$ and $\mathcal{R}^0$ is the identity so $\mathcal{R}^0(F) = F$. Now let $e_i\ A_i^\kappa\ s_i$, then $\bot(e_1)\cdots(e_n) = \bot = \mathsf{Tree}_\beta(F\,s_1\ \cdots\ s_n)$. Otherwise $n = k + 1$ and $\mathsf{Step}(\mathsf{Step}^k(\bot))(F) = [\![\mathcal{N} \vdash \mathcal{R}(F) : \kappa]\!](\mathsf{Step}^k(\bot))$ and $\mathcal{R}^k(\mathcal{R}(F)) = \mathcal{R}(F)[\mathcal{R}^k]$ and the relatedness of these latter expressions follows from the induction hypothesis and Lemma 5.2.10. □

The two foregoing results together imply that the two limits that we want to equate are essentially built out of the same finite elements. So, we now prove the main theorem.

Proof. We want to show that $[\![\mathcal{N} \vdash S : o]\!](\mu\mathsf{Step}) = \mathsf{Tree}_w(S)$. By definition, this is equivalent to showing the equality of $\bigsqcup\{\mathsf{Step}^i(\bot)(S) \mid i \in \mathbb{N}\}$ and $\bigsqcup\{u^\bot \mid S \rhd_w u\}$.

So in the $\sqsubseteq$ direction, let $i \in \mathbb{N}$, then it follows from Lemma 5.2.11 that $\mathsf{Step}^i(\bot)(S) = \mathsf{Tree}_\beta(\mathcal{R}^i(S))$. So we show that $\bigsqcup\{w^\bot \mid \mathcal{R}^i(S) \rhd_\beta^* w\} \sqsubseteq \bigsqcup\{u^\bot \mid S \rhd_w^* u\}$. Let w be such that $\mathcal{R}^i(S) \rhd_\beta^* w$ then $S \rhd_\delta^* \mathcal{R}^i(S) \rhd_\beta^* w$. It follows from Corollary 2.1.17 that therefore there is some u such that $S \rhd_w^* u$ and $u^\bot = w^\bot$. Hence, the result follows.

In the $\sqsupseteq$ direction, let $S \rhd_w^* u$. Hence, $S \rhd_{\delta\beta}^* u$ and hence, by Lemma 2.1.16 there is some u' such that $S \rhd_\delta^* u' \rhd_\beta^* u$. Hence, there is some k greater than or equal to the length of the δ-reduction and some term v, such that $\mathcal{R}^k(S) \rhd_\beta^* v$ and $u^\perp = v^\perp$. Hence, $\mathsf{Tree}_\beta(\mathcal{R}^k(S)) \sqsupseteq \mathsf{Tree}_w(S)$ and since, by Lemma 5.2.11, $\mathsf{Tree}_\beta(\mathcal{R}^k(S)) = \mathsf{Step}^k(\perp)(S)$ and the result follows. □

Interpretation of intersection refinement types

In general, a property of (the interpretation of) a term in a domain D is just a subset of D. However, in this work, we take our cue from higher-order model checking and restrict attention to a mild generalisation of the properties represented by deterministic trivial automata $\mathcal{A}$. Hence, we will define our space of properties out of an observation about the important structural features of $\mathcal{L}(\mathcal{A}^\perp)$ for any given $\mathcal{A}$, of which there are three:

(i) Every such property holds of the completely undefined tree.

(ii) Whenever such a property holds of a tree then the property holds of any less well defined tree.

(iii) Such a property can always be refuted by a finite counter-example.

In other words, such properties are safety properties specialised to the domain of finite and infinite trees. Now, in the context of a general subset of any domain, we formalise the requirements by mandating the following closure conditions.

5.2.12 Definition (Scott closed sets)**.** Given a domain D, the *Scott closure* of a subset $E \subseteq D$, denoted E^*, is the smallest set X which is closed under the following conditions:

(CL1) if $e \in E$ then $e \in X$

(CL2) $\perp \in X$

(CL3) if $e \in X$ and $d \leqslant e$ then $d \in X$

(CL4) if $Z \subseteq X$ is directed, then $\bigsqcup Z \in X$

The correspondence between (CL2), (CL3) and (i), (ii) from the foregoing discussion should be clear. To see the relationship of (CL4) to (iii), note that if Z is a set of ("finite"[1]) approximations to some limit $\bigsqcup Z$, then this condition requires that if $\bigsqcup Z$ belongs to the complement, $D \backslash X$, then some element in Z belongs to $D \backslash X$ too.

For us then, properties are Scott closed sets. Intersection types are a syntactical representation of properties with an associated proof theory, so we next show how to interpret such types as Scott closed sets in a way which validates the theory.

[1]This correspondence might be tighter if we assumed algebraicity of the underlying domains, but we have no need for it in the rest of the work.

5.2.13 Definition (Refinement type interpretation)**.** Given an interpretation D_o of the simple types, an *interpretation* of intersection refinement types over Q is a choice of, for each $q \in Q$, a Scott closed subset P_q of D_o in such a way that the subtype theory is respected, i.e. for all $q_1, q_2 \in Q$, if $(q_1, q_2) \in \Theta$ then $P_{q_1} \subseteq P_{q_2}$. The meaning function maps kinds κ to Scott closed subsets $[\![\kappa]\!]$ and intersection refinement types $\theta :: \kappa$ to inclusions $[\![\theta :: \kappa]\!] \subseteq [\![\kappa]\!]$ and is defined on intersection refinement types (whose interpretations inherit the order of the interpretations of their kinds) by:

$$
\begin{array}{lcl}
[\![q :: o]\!] & = & P_q \\
[\![\sigma \to \tau :: \kappa_1 \to \kappa_2]\!] & = & [\![\sigma :: \kappa_1]\!] \Rightarrow [\![\tau :: \kappa_2]\!] \\
[\![\bigwedge_{i=1}^{n} \tau_i :: \kappa]\!] & = & \bigcap_{i=1}^{n} [\![\tau_i :: \kappa]\!]
\end{array}
$$

Where the set of continuous functions between two Scott closed subsets $P_1 \subseteq D_1$ and $P_2 \subseteq D_2$ is defined as all those functions $f \in D_1 \Rightarrow D_2$ such that $\forall x \in P_1.\, f(x) \in P_2$.

So arrow types are interpreted as function spaces over properties and intersection types as intersections of properties. The fact that the interpretation models the associated subtype theory is a kind of statement of monotonicity.

5.2.14 Lemma (Model of subtype theory)**.** *The following is true in any interpretation of a system of intersection types in which $\theta_1 :: \kappa$ and $\theta_2 :: \kappa$ are refinements:*

$$\theta_1 \leqslant \theta_2 \quad \textit{implies} \quad [\![\theta_1 :: \kappa]\!] \subseteq [\![\theta_2 :: \kappa]\!]$$

Proof. By induction on proof of the subtype inequality:

- When concluded by (Q-Bas), necessarily θ_1 and θ_2 are base types q_1 and q_2 and $(q_1, q_2) \in \Theta$. It follows that $\kappa_1 = \kappa_2 = o$ and, from the definition of interpretation, that $(q_1, q_2) \in \Theta$ implies $P_{q_1} \subseteq P_{q_2}$ and hence $[\![q_1 :: \kappa]\!] \subseteq [\![q_2 :: \kappa]\!]$.

- When concluded by (Q-Trs), there is some θ_3 such that $\theta_1 \leqslant \theta_3$ and $\theta_3 \leqslant \theta_2$ and the result follows from the induction hypothesis and transitivity of subsetting.

- When concluded by (Q-Fun), necessarily θ_1 is of the form $\bigwedge_{i=1}^{n}(\sigma \to \tau_i)$, θ_2 is of the form $\sigma \to \tau$ and κ is of the form $\kappa_1 \to \kappa_2$. Furthermore, there is an immediate premise with $\bigwedge_{i=1}^{n} \tau_i \leqslant \tau$. Hence, it follows from the induction hypothesis that $\bigcap_{i \in [1..n]} [\![\tau_i :: \kappa_2]\!] \subseteq [\![\tau :: \kappa_1]\!]$ (*). So let $f \in [\![\bigwedge_{i=1}^{n}(\sigma \to \tau_i) :: \kappa_1 \to \kappa_2]\!]$ and let $d \in [\![\sigma :: \kappa_1]\!]$. It follows that, for all $i \in [1..n]$, $f(d) \in [\![\tau_i :: \kappa_2]\!]$. Hence, $f(d) \in \bigcap_{i \in [1..n]} [\![\tau_i :: \kappa_2]\!]$ and hence the result follows from (*).

- When concluded by (Q-Arr), necessarily θ_1 is of the form $\sigma_1' \to \tau_1$ and θ_2 is of the form $\sigma_2' \to \tau_2$ and, by definition, necessarily there is some κ_1 and κ_2 such that $\kappa = \kappa_1 \to \kappa_2$ and $\sigma_1', \sigma_2' :: \kappa_1$, $\tau_1, \tau_2 :: \kappa_2$. It follows from the induction hypothesis that $[\![\sigma_2' :: \kappa_1]\!] \subseteq [\![\sigma_1' :: \kappa_1]\!]$ and also that $[\![\tau_1 :: \kappa_2]\!] \subseteq [\![\tau_2 :: \kappa_2]\!]$. To see the result, let $d \in [\![\sigma_1' \to \tau_1]\!]$ and let $e \in [\![\sigma_2' :: \kappa_1]\!]$, then $e \in [\![\sigma_1' :: \kappa_1]\!]$ and, by definition, $d(e) \in [\![\tau_1 :: \kappa_2]\!]$. It follows that $d(e) \in [\![\tau_2 :: \kappa_2]\!]$ as required.

- When concluded by (Q-Prj), result follows from the interpretation of intersection.
- When concluded by (Q-Glb), it follows from the induction hypothesis that, for all $i \in [1..n]$, $[\![\sigma :: \kappa]\!] \subseteq [\![\tau_i :: \kappa]\!]$ and so the result follows from the interpretation of intersection.

□

The property checking problem

We are now able to specify the general setting in which we will prove our result. We first introduce a *term language*, which is the packaging together of a syntax and semantics of terms, described by a choice of signature and interpretation.

5.2.15 Definition (Term language)**.** Let D be a domain. A *term language* for D is described by a signature Σ and a D-interpretation $\{d_c\}_{c \in \Sigma}$ for Σ.

A term language contains everything necessary to build recursion schemes and interpret them. To specify properties that terms should satisfy, we use the syntax and semantics of intersection refinement types, described by a type theory and its interpretation.

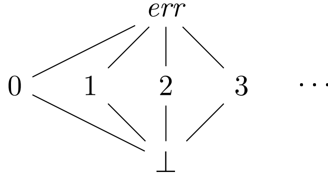
5.2.16 Definition (Property language)**.** Let D be a domain. A *property language for D* is described by an intersection type theory Q and a D-interpretation $\{P_q\}_{q \in Q}$ for Q. We say that a property language is *decidable*, just if the subtype theory Θ is decidable.

After fixing a term language and a property language one can formulate *queries*, which ask whether the interpretation of some intersection refinement type is a property that holds of the interpretation of some term.

5.2.17 Definition (Property checking problem)**.** Fix a term and property language for some domain. The associated *property checking problem* is, given a recursion scheme over the term signature, a typed term $\mathcal{N} \vdash t : \kappa$ and an intersection refinement type $\tau :: \kappa$, to decide the *query*, which is the assertion $[\![t]\!] \in [\![\tau :: \kappa]\!]$. The *ground property checking problem* is the restriction of the general setting to those refinement types with $\kappa = o$.

As we have defined it here, the property checking problem is a generalisation of the trivial automaton model checking problem. To see this, fix a recursion scheme and a trivial automaton. By interpreting the signature of the scheme by the tree-constructor semantics of Example 5.2.7 we obtain a term language. The property language is given by considering the intersection type theory over the set of states of the automaton from Definition 3.5.1 and interpreting each state q by the language $\mathcal{L}(\mathcal{A}^{\perp}, q)$. Then, by Theorem 5.2.8, it follows that $[\![S]\!] \in [\![q_0 :: o]\!]$ iff $\mathsf{Tree}_w(S) \in \mathcal{L}(\mathcal{A}^{\perp})$. However, one can formulate many other instances which are quite different from higher-order model checking in their flavour; the following concern languages with arithmetic and the throwing of errors.

5.2.18 Example (Arithmetical recursion schemes parity checking problem). *Fix a term language over a signature of arithmetic consisting of: a nullary constant* `zero`*, unary constants* `succ` *and* `isEven`*, and binary function symbols* `add` *and* `mult`*. Interpret the signature over the domain of natural numbers with non-termination and errors:*



Here we set err at the top of the domain to model the fact it has, in a sense, too much information and is thus inconsistent. Interpret concretely `zero` *by* 0*,* `succ` *by the strict successor function and, interpret* `isEven` *by the strict function that returns 0 when its argument is even or returns an error otherwise. Interpret the binary constants* `add` *and* `mult` *by the strict natural number addition* $_+_$ *and strict natural number multiplication* $_*_$ *(respectively) whose behaviour on errors is to propagate them. Fix a property language over the atoms* zero*,* even *and* odd*, with preorder generated by* $\mathsf{zero} \leqslant \mathsf{even}$*; and interpreted by* $\{0, \bot\}$*,* $\{n \in \mathbb{N} \mid n \textit{ is even}\} \cup \{\bot\}$ *and* $\{n \in \mathbb{N} \mid n \textit{ is odd}\} \cup \{\bot\}$ *respectively. Then the arithmetical recursion schemes parity checking problem is, given a recursion scheme over the signature, a closed, well kinded term* t *and an intersection refinement type* $\tau :: \kappa$*, to determine the truth of the assertion* $[\![t]\!] \in [\![\tau :: \kappa]\!]$*. For example, an instance of the problem might fix some scheme* $\mathcal{G}$ *and ask whether a given function symbol* $F \in \mathsf{dom}(\mathcal{N})$ *is oddness preserving, given that it is supplied with a first-order argument that flips parities:*

$$[\![F]\!] \in [\![(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{odd} \to \mathsf{even}) \to \mathsf{odd} \to \mathsf{odd} :: (o \to o) \to o \to o]\!]$$

5.2.19 Example (Arithmetical recursion schemes strictness checking problem). *Fix a term language as before but now consider a property language described by the single atom* $*$*, with* $[\![*]\!] = \{\bot\}$*. The strictness checking problem for arithmetical recursion schemes is, given a recursion scheme over the signature, a closed well kinded term* t *and an intersection refinement type* $\tau :: \kappa$*, to determine the truth of the assertion* $[\![t]\!] \in [\![\tau :: \kappa]\!]$*. For example, an instance of the problem might fix some scheme* $\mathcal{G}$ *over tree constructors and ask whether a third order function symbol* F *is strict in its third argument, given that its first argument is a second-order function which maps strict functions to strict functions:*

$$[\![F]\!] \in [\![((* \to *) \to * \to *) \to \top \to * \to * :: ((o \to o) \to o) \to o \to o \to o]\!]$$

The decidability of both of these problems is a straightforward consequence of the main theorem in Section 6.

5.3 Concrete and abstract properties

In this section we will place our work within the framework of abstract interpretation. Abstract interpretation is a theory of abstraction for the mathematical structures used in

the formal description of programming languages. In the classical approach to abstract interpretation, pioneered by Cousot and Cousot in the late seventies [Cousot and Cousot, 1977, 1979], one is interested in solving program analysis problems by means of approximation. The setting is a pair of complete lattices, the *concrete property domain* $\langle C, \subseteq\rangle$ and the *abstract property domain* $\langle A, \leqslant\rangle$, which are related by means of a Galois connection $\langle \alpha : C \to A, \gamma : A \to C\rangle$. The concrete property domain contains the class of properties of interest, and the abstract property domain is some approximation of it. The idea is that each concrete property $Q \in C$ has a *best* abstract description $\alpha(Q)$. Given such a setting, a program p and an abstract property $\tilde{Q}$, one can phrase a property checking assertion: $p \in \gamma(\tilde{Q})$ as a concrete inclusion $(\!|p|\!) \subseteq \gamma(\tilde{Q})$ in which $(\!|p|\!)$, the *collecting semantics* of p, is the strongest concrete property that holds of p. By virtue of the Galois connection, such a concrete inclusion can be equivalently solved by determining the (hopefully more obvious) abstract inclusion $\alpha((\!|p|\!)) \leqslant \tilde{Q}$.

Concrete properties

For us, the properties of interest are safety properties, in the sense of Scott closed sets. The Scott closed sets form a complete lattice, which is sometimes called the Hoare powerdomain. The Hoare powerdomain has many useful features which stem from the fact that it can be viewed as the free completion of the underlying domain by non-empty joins. However, the only fact that we will need is that it is equipped with a unit, whose role is to embed any element of the underlying domain continuously into the powerdomain.

5.3.1 Definition (Concrete properties)**.** In any interpreted kinded term signature, define the space of *concrete properties* of kind κ to be the Hoare powerdomain $\mathcal{P}_{\mathsf{H}}([\![\kappa]\!])$ of $[\![\kappa]\!]$. The strongest concrete property that holds of a domain element $d \in D$ is the *collecting semantics* of d, denoted $\eta_\kappa(d)$ and defined as the downward closure $\{e \in D \mid e \sqsubseteq d\}$.

5.3.2 Lemma. *Fix an interpretation of simple types. Then $\mathcal{P}_{\mathsf{H}}([\![\kappa]\!])$ is a complete lattice under inclusion with, for all $A \subseteq \mathcal{P}_{\mathsf{H}}([\![\kappa]\!])$:*

$$\bigsqcup A = (\bigcup A)^* \quad \textit{and} \quad \bigsqcap A = \bigcap A$$

Proof. A standard result on complete lattices arising from closure operators, see e.g. [Davey and Priestley, 2002]. □

Abstract properties

For us, the abstract properties are (equivalence classes of) intersection refinement types. We mirror the description of the concrete property space by segregating with respect to kind. Since we view intersection types now as abstractions, there is no gain to be made from distinguishing between two types that are subtype (i.e. precision) equivalent.

5.3.3 Definition (Abstract properties)**.** In any intersection type theory, define the space of *abstract properties*, $\mathbb{I}(\kappa)$, of kind κ to be those types that refine κ, modulo subtype equivalence. This set is equipped with a precision order, given by $[\theta_1] \leqslant [\theta_2]$ iff $\theta_1 \leqslant \theta_2$.

5.3.4 Lemma. *In any intersection type theory, $\mathbb{I}(\kappa)$ is a finite lattice under the subtype order with, for all $X \subseteq \mathbb{I}(\kappa)$, $\bigsqcap X = [\bigwedge\{\theta \mid [\theta] \in X\}]$.*

Proof. To see that this lattice is finite, simply observe that since the set of base types Q is finite and the shape of all the intersection types of kind κ is restricted by the shape of κ, there are only finitely many such. That the above defines the meet is standard for intersection type theories validating (Q-Prj) and (Q-Glb) see e.g. [Barendregt et al., 2013]. □

It follows that the join is necessarily determined as $\bigsqcup X = \bigsqcap\{[\theta] \mid \forall[\theta'] \in X. [\theta'] \leqslant [\theta]\}$. Note that the arrow constructor can similarly be lifted to operate on equivalence classes by $[\sigma] \Rightarrow [\tau] = [\sigma \to \tau]$ since this construction is well defined in any intersection type theory validating (Q-Arr).

Notation. In what follows we will drop the explicit notation for equivalence classes of types, simply denoting some class $[\theta]$ by its representative θ. Similarly, we shall use $\bigwedge$ for the meet and $\to$ for the arrow construction in preference of the "official" versions defined over the equivalence classes.

Finally, we consider a family of type environments (but now we view the types that are mapped to as equivalence classes), which are those consistent with the rules of a given recursion scheme. The definition should be compared with that in Section 2.3. The only difference is that, because we order with respect to subtyping which is dual to subset, the greatest fixed point construction becomes a least fixed point construction.

5.3.5 Definition. Fix a recursion scheme $\mathcal{G}$. Define the function Shrink which maps type environments to type environments as follows:

$$\mathsf{Shrink}(\Gamma)(F) = \bigwedge \mathbb{T}(\Gamma)(\mathcal{R}(F))$$

Note that, due to Lemma 3.2.2, Shrink is certain to map an environment that refines Δ to another environment that refines Δ. So, now fix such a kind environment Δ and define Γ_{max} as the *strongest* type environment that is a refinement of Δ, i.e. satisfying the equation $\Gamma_{max}(F) = \bigwedge\{\tau \mid \tau :: \Delta(F)\}$. Then the *strongest $\mathcal{G}$-consistent type environment* is given by $\mu\mathsf{Shrink}$ and can be computed iteratively by $\mu\mathsf{Shrink} = \bigsqcup_{i \in \omega} \mathsf{Shrink}^i(\Gamma_{max})$.

Relationship between concrete and abstract properties

For each intersection refinement type $\theta :: \kappa$, we have in mind some property $[\![\theta :: \kappa]\!] \in \mathcal{P}_{\mathsf{H}}(\kappa)$ that it *represents*. This then is the basis for the relationship between the concrete and abstract properties.

5.3.6 Definition (Concretisation)**.** In any property language define, for each kind κ, a *concretisation* map γ_κ on $\mathbb{I}(\kappa)$ by $\gamma_\kappa(\theta) = [\![\theta :: \kappa]\!]$.

Since, in any property language, the associated interpretation function on intersection refinement types is meet preserving, it follows that concretisation is a right adjoint. The corresponding left adjoint maps every concrete property to its *best* abstraction, which is the strongest intersection type which, when interpreted, represents a concrete property weaker than the original. Despite the fact that it is completely determined by its role as an adjoint, we give the explicit definition since it will be used frequently.

5.3.7 Lemma. *In any property language, at each kind κ there exists a Galois connection:*

$$\mathcal{P}_{\mathsf{H}}([\![\kappa]\!]) \underset{\alpha_\kappa}{\overset{\gamma_\kappa}{\leftrightarrows}} \mathbb{I}(\kappa)$$

in which α_κ is given by, for all Scott closed subsets P, $\alpha_\kappa(P) = \bigwedge\{\theta \in \mathbb{I}(\kappa) \mid P \subseteq \gamma_\kappa(\theta)\}$.

Proof. Follows from the fact that $\mathbb{I}(\kappa)$ has all meets (Lemma 5.3.4) and γ_κ preserves them by definition. □

With the foregoing lemma we complete the description of the setting in which to talk about abstraction. What is left is to specify the way in which concrete properties $\eta([\![t]\!])$, determined as the collecting semantics of terms t, are interpreted in the abstract domain. In order to specify this neatly, we first introduce an abstract version of application, denoted simply by $\cdot$, which given a type of kind $\kappa_1 \to \kappa_2$ and a type of kind κ_1 returns a type of kind κ_2.

5.3.8 Definition (Refinement type application)**.** We define a family of application operators $\cdot_{\kappa_1 \to \kappa_2} : \mathbb{I}(\kappa_1 \to \kappa_2) \times \mathbb{I}(\kappa_1) \to \mathbb{I}(\kappa_2)$ as follows (we will usually omit the subscript):

$$\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i) \cdot \sigma = \bigwedge\{\tau_i \mid i \in [1..n] \wedge \sigma \leqslant \sigma_i\}$$

Recall that any intersection refinement type $\theta :: \kappa_1 \to \kappa_2$ can be viewed as an intersection of arrow types $\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i)$. The operator can be seen as a finite version of the type application defined when constructing filter models using intersection types. Its well definedness over the quotient follows from its monotonicity on the syntax.

5.3.9 Lemma. *Type application is monotone.*

Proof. It is sufficient to see that it is monotone in both arguments separately. Monotonicity in the second argument is obvious. To see that it is monotone in the first argument, let $\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i) \leqslant \bigwedge_{j=1}^{m}(\sigma'_j \to \tau'_j)$. We aim to show, for all σ, $\bigwedge\{\tau_i \mid i \in [1..n] \wedge \sigma \leqslant \sigma_i\} \leqslant \bigwedge\{\tau'_j \mid j \in [1..m] \wedge \sigma \leqslant \sigma'_j\}$ by (Q-Glb), so let $j \in [1..m]$ and let $\sigma \leqslant \sigma'_j$. It follows by inversion (Lemma 3.1.4) that necessarily there is some $N \subseteq [1..n]$ such that $\forall i \in N.\, \sigma'_j \leqslant \sigma_i$ and $\bigwedge_{i \in N} \tau_i \leqslant \tau'_j$. Hence, for each $i \in N$, $\sigma \leqslant \sigma_i$ and hence $\bigwedge\{\tau_i \mid i \in [1..n] \wedge \sigma \leqslant \sigma_i\} \leqslant \bigwedge\{\tau_i \mid i \in N\} \leqslant \tau'_j$. □

To specify the abstract interpretation of (the collecting semantics of) a term, we follow the intuitions that guided us in the motivating remarks in the introduction and formalise the situation as a certain assignment of intersection types to just the constants over which the term is built.

5.3.10 Definition (Abstract interpretation)**.** Fix a term and a property language over some domain D. An *abstract interpretation* of the term language into the property language is a choice of intersection type σ_c for each constant $c \in \Sigma$ such that the kinding is respected, i.e. $\sigma_c :: \mathsf{kind}(c)$. We say that an abstract interpretation is *sound* whenever, for all constants c of rank n:

$$\forall d_1 \in [\![o]\!], \dots, d_n \in [\![o]\!].\ \alpha(\eta([\![c]\!](d_1)\cdots(d_n)) \leqslant \sigma_c \cdot \alpha(\eta(d_1)) \cdot \cdots \cdot \alpha(\eta(d_n))$$

We say that an abstract interpretation is *complete* whenever, for all constants c of rank n:

$$\forall d_1 \in [\![o]\!], \dots, d_n \in [\![o]\!].\ \sigma_c \cdot \alpha(\eta(d_1)) \cdot \cdots \cdot \alpha(\eta(d_n)) \leqslant \alpha(\eta([\![c]\!](d_1)\cdots(d_n))$$

We say that an abstract interpretation is *exact* just if it is both sound and complete. It is exactness of abstract interpretations that will be important in what follows.

5.4 Exact abstraction at ground types

In this section we shall state and prove the ground kind restriction of the main result, discuss its consequences and show that, at ground kind, every problem characterised by a type system can be reduced to higher-order model checking.

The theorem and its consequences

We first introduce the analogue of the type system induced by a (co-)trivial automaton described in Definition 3.5.1, which is the type system induced by an abstract interpretation. Given that the intersection type theory and the assignment of types to constants is part of the specification of an abstract interpretation, the definition of the induced type system is trivial.

5.4.1 Definition (Type system induced by an abstract interpretation)**.** Fix a term language over a signature Σ, a property language over type theory Q and an abstract interpretation $\{\sigma_c\}_{c\in\Sigma}$. The *intersection refinement type system induced by the abstract interpretation* is the system $\langle \Sigma, Q, \mathsf{type} \rangle$ with, for all $c \in \Sigma$, $\mathsf{type}(c) = \sigma_c$.

In what follows, we will not explicitly introduce the type system induced by an abstract interpretation, but assume it can be inferred from the introduction of the abstract interpretation. The content of the main theorem is that, whenever the abstract interpretation is exact, the best abstraction of a ground type term is *exactly* the strongest type assignable to that term in the induced type system. The situation is depicted in Figure 5.1.

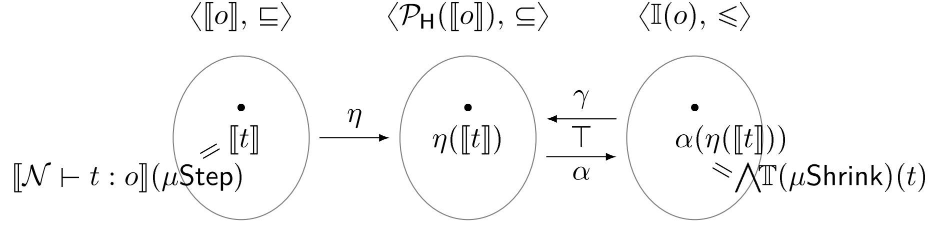


Figure 5.1: Exact abstraction at ground kind.

5.4.2 Theorem (Exact abstraction at ground type)**.** *Fix a term language, a property language and an exact abstract interpretation. Then, for all closed terms t of ground type:*

$$\alpha(\eta(\llbracket t \rrbracket)) = \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t)$$

in any recursion scheme over the signature of the term language.

The first consequence is that, since in any decidable intersection type theory $\mu\mathsf{Shrink}$ is computable, it follows that therefore the best abstraction of the collecting semantics of a given term t is computable. Hence, it is possible to compute all the abstract properties satisfied by a given term. Since the concrete and abstract properties are in Galois correspondence, the ground property checking problem is decidable.

5.4.3 Corollary. *Fix a term language and a decidable property language. If there exists an exact abstract interpretation of the former into the latter, then the ground property checking problem is decidable.*

Proof. We reason equationally:

$$\begin{aligned}
\llbracket t \rrbracket \in \llbracket \tau :: o \rrbracket \quad &\equiv \quad \llbracket t \rrbracket \in \gamma_o(\tau) \\
&\equiv \quad \eta_o(\llbracket t \rrbracket) \subseteq \gamma_o(\tau) \\
&\equiv \quad \alpha_o(\eta_o(\llbracket t \rrbracket)) \leqslant \tau \\
&\equiv \quad \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t) \leqslant \tau
\end{aligned}$$

With the final equivalence following from Theorem 5.4.2. Since the strongest type assignable to t in $\mu\mathsf{Shrink}$ is computable, and the subtype relation is assumed decidable, so the inequality is decidable. □

In the recursion scheme variants introduced in the literature, each is presented alongside an intersection type system characterising its model checking problem. In each case a proof is developed, from scratch, to justify the characterisation. Although the proofs were specific to each variant, they more or less all follow a recipe, and important ingredients are the sub-proofs of subject reduction and subject expansion for the type system. Since subject reduction and subject expansion are often of interest independently of any

attempt to characterise a model checking problem, we next point out that they are a simple consequence of having an exact abstract interpretation. As a related aside, we first build a finite intersection type model of a term language (and the schemes that live in it), defined in terms of the property language and the abstract interpretation.

5.4.4 Definition (Refinement type interpretation of recursion schemes)**.** Fix a term language, a property language and an exact abstract interpretation. We build an intersection refinement type model as follows:

(i) Interpret kind κ by $\mathbb{I}(\kappa)$.

(ii) Interpret kinded term $\Delta \vdash t : \kappa$ by monotone function:

$$\bigwedge \mathbb{T}(-)(t) \in \Pi_{\xi \in \mathsf{dom}(\Delta)}.\mathbb{I}(\Delta(\xi)) \to \mathbb{I}(\kappa)$$

Extend this to recursion schemes by interpreting any given scheme as the least fixed point of the induced Shrink function, $\mu\mathsf{Shrink}$.

To say that the above interpretation is a sound model of weak reduction is exactly to say that subject reduction and subject expansion hold with respect to weak reduction. That this is true of the intersection type interpretation arising from an exact abstract interpretation is the content of the following:

5.4.5 Corollary. *Fix a term language, a property language, an exact abstract interpretation and a recursion scheme. For all closed, ground kind terms s and t:*

$$\text{if} \quad s \rhd_w t \quad \text{then} \quad \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(s) = \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t)$$

Proof. Assume $s \rhd_w t$. Since the kind interpretation models reduction (Lemma 5.2.6), also $[\![s]\!] = [\![t]\!]$ and hence $\alpha(\eta([\![s]\!])) = \alpha(\eta([\![t]\!]))$. The result follows from Theorem 5.4.2. □

Proof of the theorem

We now prove the theorem. Recall that the statement is:

> *Fix a term language, a property language and an exact abstract interpretation. For all closed terms t of ground kind:*
>
> $$\alpha_o(\eta_o([\![t]\!])) = \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t)$$
>
> *in any recursion scheme over the signature of the term language.*

The main tool that we use in proving the theorem is a pair of logical relations $R^\kappa_\leqslant$ and $R^\kappa_\geqslant$ between elements of the domain model and intersection types. The idea is that, if $d \; R^\kappa_\leqslant \; \theta$, then the collecting semantics of d is soundly approximated by θ and, if $d \; R^\kappa_\geqslant \; \theta$, then the collecting semantics of d is completely approximated by θ. However, this description is not precise: although we will see later that $d \; R^\kappa_\leqslant \; \theta$ is, indeed, equivalent to $\alpha(\eta(d)) \leqslant \theta$,

we will also see that, beyond ground kind, $d\ R^{\kappa}_{\geqslant}\ \theta$ is not generally equivalent to $\alpha(\eta(d)) \geqslant \theta$. Nevertheless, we are here interested in ground kind only, so let us proceed with the definition.

5.4.6 Definition. Fix a term language and a property language. We construct two binary logical relations $R^{\kappa}_{\leqslant}$ and $R^{\kappa}_{\geqslant}$, indexed by kinds κ, between $[\![\kappa]\!]$ and $\mathbb{I}(\kappa)$. The definitions are inductive on κ, let $\diamond$ stand for either $\leqslant$ or $\geqslant$, then:

(R1) When $\kappa = o$, $d\ R^{o}_{\diamond}\ \theta$ just if $\alpha(\eta(d)) \diamond \theta$.

(R2) When κ is an arrow kind $\kappa_1 \to \kappa_2$, $f\ R^{\kappa_1 \to \kappa_2}_{\diamond}\ \theta$ just if, for all $d \in [\![\kappa_1]\!]$ and $\theta' \in \mathbb{I}(\kappa_1)$ such that $d\ R^{\kappa_1}_{\diamond}\ \theta'$, $f(d)\ R^{\kappa_2}_{\diamond}\ \theta \cdot \theta'$.

and extended to kind environments Δ by:

(R3) $\Xi\ R^{\Delta}_{\diamond}\ \Gamma$ just if, for all $\xi \in \mathsf{dom}(\Delta)$, $\Xi(\xi)\ R^{\Delta(\xi)}_{\diamond}\ \Gamma(\xi)$

As standard, we make use of the basic lemma. We might have obtained this for free by construing our type assignment as a model, but we prefer to do things the other way around, by means of Corollary 5.4.5.

5.4.7 Lemma. *Fix a term language, a property language and an exact abstract interpretation. For all kinded terms $\Delta \vdash t : \kappa$, environments $\Xi \in \Pi_{\xi \in \Delta}.[\![\Delta(\xi)]\!]$ and $\Gamma :: \Delta$ such that $\Xi\ R^{\Delta}_{\diamond}\ \Gamma$:*

$$[\![\Delta \vdash t : \kappa]\!](\Xi)\ R^{\kappa}_{\diamond}\ \bigwedge(\mathbb{T}(\Gamma)(t))$$

Proof. by induction on the derivation of $\Delta \vdash t : \kappa$.

- If t is a variable or function symbol then $[\![\Delta \vdash t : \kappa]\!](\Xi) = \Xi(t)$ and also $\bigwedge\mathbb{T}(\Gamma)(t) = \Gamma(t)$ and, since $\Xi\ R^{\Delta}_{\diamond}\ \Gamma$, so $\Xi(t)\ R^{\Delta(t)}_{\diamond}\ \Gamma(t)$.

- If t is a constant, then it is has some arity, say n. For each $i \in [1..n]$, let $e_i\ R^{o}_{\diamond}\ \theta_i$, by definition $\alpha(\eta(e_i)) \diamond \theta_i$. Then, since the abstract interpretation is exact and type application is monotonic, $\alpha(\eta([\![c]\!](e_1) \cdots (e_n))) \diamond \sigma_c \cdot \theta_1 \cdots \theta_n$.

- If t is an application $u\,v$ then there is some κ' such that $[\![\Delta \vdash t : \kappa]\!] = [\![\Delta \vdash u : \kappa' \to \kappa]\!](\Xi)([\![\Delta \vdash v : \kappa']\!](\Xi))$ and it follows from the induction hypothesis that

$$[\![\Delta \vdash u : \kappa' \to \kappa]\!](\Xi)\ R^{\kappa' \to \kappa}_{\diamond}\ \bigwedge\mathbb{T}(\Gamma)(u)$$

  and that $[\![\Delta \vdash v : \kappa']\!](\Xi)\ R^{\kappa'}_{\diamond}\ \bigwedge\mathbb{T}(\Gamma)(v)$. The result then follows from (R2).

- If t is an abstraction $\lambda x_1 \cdots x_n.\,s$ then κ is of shape $\kappa_1 \to \cdots \to \kappa_n \to o$, so let $d_i\ R^{\kappa_i}_{\diamond}\ \theta_i$ for each such i. Then we observe that $[\![\Delta \vdash \lambda x_1 \cdots x_n.\,t : \kappa]\!](d_1) \cdots (d_n)$ is the same element of $[\![o]\!]$ as $[\![\Delta, \cup\{x_i : \kappa_i \mid i \in [1..n]\} \vdash t : o]\!](\Xi \cup \{x_i \mapsto d_i \mid i \in [1..n]\})$ which, by the induction hypothesis, is related by $R^{o}_{\diamond}$ to $\bigwedge\mathbb{T}(\Gamma \cup \{x_i : \theta_i \mid i \in [1..n]\})(t)$. Finally, since $\bigwedge\mathbb{T}(\Gamma \cup \{x_i : \theta_i\}_{i \in [1..n]})(t)$ is subtype equivalent to $\bigwedge\mathbb{T}(\Gamma)(\lambda x_1 \cdots x_n.\,t) \cdot \theta_1 \cdots \theta_n$, they are identified in the quotient.

□

The important observation in our situation is that the semantic environment and the type environment are stepwise related in their construction. It follows that they themselves are related.

5.4.8 Lemma. *Fix a term language, a property language and an exact abstract interpretation. For any recursion scheme:*

$$\mu\mathsf{Step}\ R_\diamond^{\mathcal{N}}\ \mu\mathsf{Shrink}$$

Proof. We first demonstrate, by induction on n, that for all $n \in \mathbb{N}$, $\mathsf{Step}^n(\bot)\ R_\diamond^{\mathcal{N}}\ \mathsf{Shrink}^n(\bot)$.

- Suppose $n = 0$, and let $F : \kappa \in \mathcal{N}$ with κ of the form $\kappa_1 \to \cdots \to \kappa_m \to o$. Then $\mathsf{Step}^n(\bot)(F) = \bot(F) = \bot_{[\![\kappa]\!]}$. On the other hand, $\mathsf{Shrink}^n(\bot)(F) = \bot(F) = \bot_{\mathbb{I}(\kappa)}$. Let $d_i\ R_\diamond^{R^{\kappa_i}}\ \theta_i$ for $i \in [1..m]$ then, by definition, $\bot(d_1)\cdots(d_n) = \bot_{[\![o]\!]}$ and $\bot \cdot \theta_1 \cdots \theta_n = \bot_{\mathbb{I}(o)}$. Since $\alpha \circ \eta$ preserves bottom, the result follows.

- Suppose $n = k+1$, and let $F : \kappa \in \mathcal{N}$. Then

$$\mathsf{Step}^n(\bot)(F) = \mathsf{Step}(\mathsf{Step}^k(\bot))(F) = [\![\mathcal{R}(F)]\!](\mathsf{Step}^k(\bot))$$

and, on the right hand side:

$$\mathsf{Shrink}^n(\bot)(F) = \mathsf{Shrink}(\mathsf{Shrink}^k(\bot))(F) = \bigwedge \mathbb{T}(\mathsf{Shrink}^k(\bot))(\mathcal{R}(F))$$

then the result follows from the induction hypothesis and (1).

Now, to see the result, let $F : \kappa_1 \to \cdots \to \kappa_n \to o$ be in $\mathcal{N}$ and let $d_i\ R_\diamond^{\kappa_i}\ \theta_i$ $(i \in [1..n])$. We reason transitively:

$$\begin{array}{rcl}
\alpha(\eta(\mu\mathsf{Step}(F)(d_1)\cdots(d_n))) & = & \alpha(\eta(\bigsqcup\{\mathsf{Step}^m(\bot) \mid m \in \mathbb{N}\}(F)(d_1)\cdots(d_n))) \\
& = & \alpha(\eta(\bigsqcup\{\mathsf{Step}^m(\bot)(F)(d_1)\cdots(d_n) \mid m \in \mathbb{N}\})) \\
& = & \bigsqcup\{\alpha(\eta(\mathsf{Step}^m(\bot)(F)(d_1)\cdots(d_n))) \mid m \in \mathbb{N}\} \\
& \diamond & \bigsqcup\{\mathsf{Shrink}^m(\bot)(F)\cdot\theta_1\cdots\theta_n \mid m \in \mathbb{N}\} \\
& = & \mu\mathsf{Shrink}(F)\cdot\theta_1\cdots\theta_n
\end{array}$$

where the third equality follows by continuity of $\alpha \circ \eta$ and the inequality follows by the previous result and a standard property of join. □

It now remains only to put the two results together in order to prove the theorem.

Proof. By Lemma 5.4.8, the two environments are related and hence the result follows from Lemma 5.4.7. □

Reduction to trivial automaton model checking

We now show that all ground type property checking problems can be reduced to trivial automaton model checking problems. The key point to observe is that the process of type assignment does not depend in any way upon the particular (concrete) interpretation of the constants, and hence they may as well just be tree constructors. More specifically, the type systems that characterise given model checking problems and the type systems that characterise the more general property checking problems are the same, in the sense of both being instances of Definition 3.2.1. The only differences are that one is induced by an automaton and one is induced by an abstract interpretation. However, given any abstract interpretation $\{\sigma_c\}_{c\in\Sigma}$, one can construct a property automaton $\mathcal{A}$ such that the type system induced by the abstract interpretation and the type system induced by the automaton are the same, modulo the subtype relation. The subtype relation induced by the automaton is always generated by a discrete ordering of the states, but the subtype relation associated with an abstract interpretation can be arbitrary (within the constraints set down by the definition). However, we will now show that, given any instance of a property checking problem with an arbitrary subtype relation, we can construct an instance of a related property checking problem in which the subtype relation plays no part.

5.4.9 Definition. Fix a signature Σ, a type system $\mathcal{T} = \langle\Sigma, Q, \mathsf{type}\rangle$. The signature with casting $\Sigma^\dagger$ is the signature Σ augmented by a new unary constant cast. The type system with casting $\mathcal{T}^\dagger$ is the system $\langle\Sigma^\dagger, Q^\dagger, \mathsf{type}^\dagger\rangle$ in which $Q^\dagger$ is the *discrete* type theory over Q and $\mathsf{type}^\dagger$ maps each constant $c \in \Sigma$ to $\mathsf{type}(c)$ and $\mathsf{type}(\mathsf{cast}) = \bigwedge\{\sigma \to q \mid \sigma \leqslant q \text{ and } \sigma :: o\}$. We define a new family of rules $\mathcal{R}_{Cast}$, inductively by kind, in order to lift casting to higher-orders:

$$
\begin{aligned}
Cast^{o} &= \lambda x.\, \mathsf{cast}\, x \\
Cast^{\kappa_1\to\cdots\to\kappa_n\to o} &= \lambda x\, y_1\, \cdots\, y_n.\, \mathsf{cast}\,(x\,(Cast^{\kappa_1}\, y_1)\, \cdots\, (Cast^{\kappa_n}\, y_n))
\end{aligned}
$$

Given a term $\Delta \vdash t : \kappa'$, we define the term with casting $t^\dagger$ obtained by replacing each occurrence of an application $u\,v$ in which v has kind κ by an application $u\,(Cast^{\kappa} v)$. Given a scheme $\mathcal{G}$, we construct a scheme with casting $\mathcal{G}^\dagger$ by replacing the right hand side of each rule t by $t^\dagger$ and adding the minimal subset of the family of rules defined above in order to be well kinded.

We first note that the family of functions $Cast^{\kappa}$ really represent a lifting of the cast constant to higher-orders, in the sense that it is always consistent to type them as such.

5.4.10 Proposition (Casts perform casts)**.** *Fix a type system in which cast is typed by* $\mathsf{type}(\mathsf{cast}) = \sigma_{\mathsf{cast}}$. *Fix a kind κ and let Γ be the environment*

$$\{Cast^{\kappa'} : \sigma \to \tau \mid \kappa' \text{ a sub-kind of } \kappa \text{ and } \sigma \to \tau :: \kappa' \to \kappa' \text{ and } \sigma \leqslant \tau\}$$

Then, for all κ' a sub kind of κ and for all $\tau \in \Gamma(Cast^{\kappa'})$, $\Gamma \vdash \mathcal{R}(Cast^{\kappa'}) : \tau$.

Proof. Straightforward induction on κ. □

Then we show that inserting casts into every application faithfully simulates subtyping.

5.4.11 Lemma (Casts simulate subtyping.)**.** *Fix a scheme $\mathcal{G}$ and a type system $\mathcal{T}$ over the same signature. Let Γ be a type environment and let $\Gamma^\dagger$ be the environment:*

$$\Gamma \cup \{\mathit{Cast}^\kappa : \sigma \to \tau \mid \sigma \leqslant \tau \textit{ and } \sigma \to \tau :: \kappa \to \kappa \textit{ and } \mathit{Cast}^\kappa \in \mathsf{dom}(\mathcal{N}^\dagger)\}$$

There is a derivation of $\Gamma \vdash t : \tau$ in $\mathcal{T}$ iff there is a derivation of $\Gamma^\dagger \vdash t^\dagger : \tau$ in $\mathcal{T}^\dagger$.

Proof. In the forward direction, observe that given an instance of the (T-App) rule:

$$\frac{\Gamma \vdash u : \sigma \to \tau \qquad \Gamma \vdash v : \tau_i (i \in [1..n]) \qquad \bigwedge_{i=1}^n \tau_i \leqslant \sigma}{\Gamma \vdash u\, v : \tau}$$

then one can replace it in the system with explicit casts by an instance:

$$\frac{\Gamma^\dagger \vdash u : \sigma \to \tau \qquad \dfrac{\overline{\Gamma^\dagger \vdash \mathit{Cast}^\kappa : \bigwedge_{i=1}^n \tau_i \to \tau'} \qquad \Gamma^\dagger \vdash v : \tau_i (i \in [1..n])}{\Gamma^\dagger \vdash \mathit{Cast}^\kappa v : \tau'} \text{ (for each } \tau' \in \sigma)}{\Gamma^\dagger \vdash u\, (\mathit{Cast}^\kappa v) : \tau}$$

whilst the rest of the derivation remains the same. The other direction is symmetric. □

Since the consistency of an environment just depends upon typing the right-hand sides of the rules appropriately, and all the right-hand sides t of rules in $\mathcal{G}$ have been replaced by right-hand sides $t^\dagger$ in $\mathcal{G}^\dagger$ (and the self-consistent *Cast* rules added) it follows that there is a consistent environment for $\mathcal{G}$ in $\mathcal{T}$ iff there is one for $\mathcal{G}^\dagger$ in $\mathcal{T}^\dagger$.

5.4.12 Corollary. *Fix a scheme $\mathcal{G}$ and a type system $\mathcal{T}$ over the same signature. There exists a $\mathcal{G}$-consistent environment in $\mathcal{T}$ that types $t : \tau$ iff there exists a $\mathcal{G}$-consistent environment in $\mathcal{T}^\dagger$ that types $t^\dagger : \tau$.*

Proof. In the forward direction, let Γ be the witness and let $\Gamma^\dagger$ be the environment:

$$\Gamma \cup \{\mathit{Cast}^\kappa : \sigma \to \tau \mid \sigma \leqslant \tau \text{ and } \sigma \to \tau :: \kappa \to \kappa \text{ and } \mathit{Cast}^\kappa \in \mathsf{dom}(\mathcal{N}^\dagger)\}$$

Then $\Gamma^\dagger$ is $\mathcal{G}^\dagger$-consistent in $\mathcal{T}^\dagger$ since, let $\tau \in \Gamma^\dagger(F)$, if $F \in \mathsf{dom}(\Gamma)$ then it follows from Lemma 5.4.11 that $\Gamma^\dagger \vdash \mathcal{R}^\dagger(F) : \tau$ and if F is of the form Cast^κ then $\Gamma^\dagger \vdash \mathcal{R}^\dagger(F) : \tau$ follows from Proposition 5.4.10. Furthermore, by Lemma 5.4.11 also $\Gamma^\dagger \vdash t^\dagger : \tau$. The other direction is symmetric. □

Both the trivial automaton model checking problem for higher-order recursion schemes and, given an exact abstract interpretation, the property checking problem associated with any given term and property language come down to a common substrate which is type assignment in the induced type systems. We have now shown that from the system induced by the latter one can construct a system of the former which is, in a sense, equivalent. Hence, we can conclude the following:

5.4.13 Theorem (Problem reduction)**.** *Fix a term language and property language for which there exists an exact abstract interpretation. For every instance of the ground property checking problem there is an equivalent instance of the trivial automaton model checking problem for recursion schemes.*

Proof. Fix a term language over signature, a property language and an exact abstract interpretation and let the induced type system be $\mathcal{T}$. Let $\mathcal{G}$, t and $q :: o$ be an instance of the associated property checking problem. Let trivial automaton $\mathcal{A} = \langle \Sigma^\dagger, Q, \delta, q \rangle$ with, for each constant $c \in \Sigma$ and state $q' \in Q$,

$$\delta(q', c) = \bigvee_{\tau \in \sigma_c} \bigwedge \{(i, q'') \mid \tau = \sigma_1 \to \cdots \to \sigma_n \to q \text{ and } q'' \in \sigma_i\}$$

It follows that the type system induced by $\mathcal{A}$ is exactly $\mathcal{T}^\dagger$ (*). Finally, let $\mathcal{G}'$ be $\mathcal{G}^\dagger$ augmented with new start symbol S' and rule $S' = \mathsf{cast}\ t^\dagger$. It follows from Theorem 5.4.2 that $[\![t]\!] \in [\![q :: o]\!]$ iff there exists some $\mathcal{G}$-consistent environment that types t with some state $q' \leqslant q$ in $\mathcal{T}$, by Corollary 5.4.12 this occurs iff there is some $\mathcal{G}^\dagger$-consistent environment that types $\mathsf{cast}\ t^\dagger : q$, and by Theorem 3.5.6 this occurs iff $\langle \mathcal{G}', \mathcal{A} \rangle$ is a yes instance of the trivial automaton model checking problem for recursion schemes. □

5.5 Applications to higher-order model checking

We now consider some property checking problems arising from particular language features that have been found to be useful in higher-order model checking applications. As a warm up exercise we give (yet another) proof that the trivial automaton model checking problem for recursion schemes is decidable, but thanks to Theorem 5.4.2, the proof is not more than a few lines. We then consider the Boolean recursion schemes property checking problem, which was the situation considered in the introduction. Boolean recursion schemes are interesting because they are, in a precise sense, the analogue of Boolean Programs, an extremely successful class of model for first-order program verification problems. It is not too difficult to generalise from the Boolean case to any finite enumeration, thus capturing the essential feature of RSFD [Kobayashi et al., 2010]. We next take a look at nondeterministic choice, expressed as a binary constant. This allows us to characterise the trivial automaton model checking problem for nondeterministic schemes, which is probably the most popular higher-order model checking situation at present, but one which is not usually given the rigorous treatment that we will give it here: usually the choice constant is considered a tree constructor like any other and some meta-argument is given to justify how the trees generated are related to a particular, nondeterministic, computation. Finally we consider the combination of nondeterministic choice and a case analysis construct over a finite enumeration. This takes us to the realm of HORSC and, following the paper that introduced HORSC model checking [Neatherway et al., 2012], we characterise the situation by introducing unions at base type.

Alternating trivial automaton model checking of recursion schemes. Our first example is just to show that it is easy to recover the type based characterisation of alternating trivial automaton model checking for recursion schemes from the foregoing theorem. So let $\mathcal{G}$ be a recursion scheme and $\mathcal{A}$ a trivial automaton over the same signature Σ. As before, let us interpret the type o by the domain of finite and infinite $\Sigma^{\perp}$-labelled trees and interpret each constant a concretely as the corresponding tree constructor $\underline{a}$. We build our property language over the set of states Q of $\mathcal{A}$ with the discrete subtype theory. Then let us interpret each state q as the set of all trees accepted from state q, i.e. $\mathcal{L}(\mathcal{A}, q)$, thus making precise the informal motivational remarks from Section 2.3. We construct an abstract interpretation by assigning to constant a the intersection type:

$$\sigma_a = \bigwedge\{\bigwedge(S|_1) \to \cdots \to \bigwedge(S|_n) \to q \mid S \models \delta(q, a)\}$$

All that remains is to check that this abstract interpretation is exact, so let c be a constant of arity n, we argue in both directions by (Q-Glb).

- To see soundness, let q be such that there is some satisfying assignment S to $\delta(q, c)$ such that, for all $i \in [1..n]$, $\alpha(\eta(T_i)) \leqslant \bigwedge(S|_i)$. Hence, for each such i and for each $q' \in S|_i$, $T_i \in \gamma(q') = [\![q']\!] = \mathcal{L}(\mathcal{A}^{\perp}, q')$. Thus, $\underline{c}(T_1)\cdots(T_n) \in \mathcal{L}(\mathcal{A}^{\perp}, q)$ and so $\alpha(\eta(\underline{c}(T_1)\cdots(T_n))) \leqslant q$.
- To see completeness, let $q \in \gamma(\underline{c}(T_1)\cdots(T_n))$, so $\underline{c}(T_1)\cdots(T_n) \in \mathcal{L}(\mathcal{A}^{\perp}, q)$. Necessarily, there is some satisfying assignment S to $\delta(q, c)$ and, for each $(i, q') \in S$, $T_i \in \mathcal{L}(\mathcal{A}^{\perp}, q')$. Hence, for each i, $\alpha(\eta(T_i)) \leqslant \bigwedge(S|_i)$; thus $\sigma_c \cdot \alpha(\eta(T_1))\cdots\alpha(\eta(T_n)) \leqslant q$.

By Theorem 5.4.2, it follows that this finite intersection type system gives a sound and complete characterisation of alternating trivial model checking for recursion schemes and hence confirms its decidability.

Boolean recursion schemes. Our second example is of *Boolean recursion schemes* which are built over the signature containing `true` and `false` of zero arity and `if` of arity 3. Of course, we interpret o by the flat domain of Booleans $\{true, false, \perp\}$ and we interpret the zero arity constants by *true* and *false* respectively and the if statement by:

$$[\![\texttt{if}]\!](b)(b_1)(b_2) = \begin{cases} b_1 & \text{when } b = true \\ b_2 & \text{when } b = false \\ \perp & \text{otherwise} \end{cases}$$

The property language is built over the base types Q consisting of t and f under the discrete subtype order and interpreted as $\{true, \perp\}$ and $\{false, \perp\}$ respectively. The abstract interpretation of the two constants `true` and `false` is given by t and f respectively and the `if` by:

$$\begin{aligned} \sigma_{\texttt{if}} \;&=\; (\mathsf{t} \to \mathsf{t} \to \top \to \mathsf{t}) \wedge (\mathsf{t} \to \mathsf{f} \to \top \to \mathsf{f}) \\ &\wedge\; (\mathsf{f} \to \top \to \mathsf{t} \to \mathsf{t}) \wedge (\mathsf{f} \to \top \to \mathsf{f} \to \mathsf{f}) \\ &\wedge\; (\mathsf{t} \wedge \mathsf{f} \to \top \to \top \to \mathsf{t}) \wedge (\mathsf{t} \wedge \mathsf{f} \to \top \to \top \to \mathsf{f}) \end{aligned}$$

All that remains is to check that the abstract interpretation is exact. It is obvious that the condition is met by the constants `true` and `false`, so consider `if`. We aim to show:

$$\alpha(\eta([\![\mathtt{if}]\!](b)(b_1)(b_2))) = \sigma_{if} \cdot \alpha(\eta(b)) \cdot \alpha(\eta(b_1)) \cdot \alpha(\eta(b_2))$$

directly, so proceed by cases on b. If b is true then $[\![\mathtt{if}]\!](b)(b_1)(b_2) = b_1$. Since, trivially, the best abstraction of `true` is t, it follows that the RHS equals $\alpha(\eta(b_1))$. We may reason analogously when b is false. When b is bottom, we have $[\![\mathtt{if}]\!](b)(b_1)(b_2) = \bot$ and, since $\alpha \circ \eta$ is strict and $\bot_{\mathbb{I}(o)} = \mathsf{t} \wedge \mathsf{f}$, the result follows.

Hence, it follows from Theorem 5.4.2 that the ground property checking problem for Boolean recursion schemes is characterised by an intersection type system over base types for true and false and is, therefore, decidable. It should be clear that moving from if-statements and the Booleans to case-statements and finite enumerations gives a similarly easy treatment of the case construct found in, e.g. RSFD.

Nondeterministic term languages. In the higher-order model checking literature, it is typical for a recursion scheme to simulate non-determinism by the addition to its signature of a binary choice constant, usually named `br`, for "branch". Although the constant is usually treated as a tree constructor, an argument is often made as to why the tree that is generated by some scheme over a signature containing `br` represents the result of a nondeterministic computation.

5.5.1 Definition (Nondeterministic term language)**.** For us, a *nondeterministic term language* is a term language that is built over a signature containing binary constant `br` and is interpreted with respect to the powerdomain $\mathcal{P}_{\mathsf{H}}(D)$ for some underlying domain D with $[\![\mathtt{br}]\!](d_1)(d_2) = d_1 \sqcup d_2$.

In our setting, the concrete property domain corresponding to a nondeterministic term language is of the form $\mathcal{P}_{\mathsf{H}}(\mathcal{P}_{\mathsf{H}}(D))$. In practice this seems like it may be a little unwieldy. In particular, at base kind, rather than specify a property $P' \in \mathcal{P}_{\mathsf{H}}(\mathcal{P}_{\mathsf{H}}(D))$ and ask $[\![t]\!] \in P$, one would typically want to specify a property $P \in \mathcal{P}_{\mathsf{H}}(D)$ and ask $[\![t]\!] \subseteq \mathcal{P}_{\mathsf{H}}(D)$. In other words, is every outcome of the program (closed, ground kind term) acceptable? However, note that this form of question is subsumed by this setting since $[\![t]\!] \subseteq P$ iff $[\![t]\!] \in \{P\}^*$, since ground kind $[\![t]\!]$ is Scott closed in any nondeterministic term language. By way of another example, given two properties $P \in \mathcal{P}_{\mathsf{H}}(D_1)$ and $Q \in \mathcal{P}_{\mathsf{H}}(D_2)$, we have $[\![t]\!] \in \{P\}^* \Rightarrow \{Q\}^*$ equivalent to the (perhaps) more natural specification: $\forall D \subseteq P.\, [\![t]\!](D) \subseteq Q$.

Deterministic trivial automaton model checking of nondeterministic recursion schemes. Fix a recursion scheme $\mathcal{G}$ and deterministic trivial automaton $\mathcal{A}$ over the same signature. Recall that a deterministic trivial automaton has a transition function which either maps each constant c of arity n to f, or else to a formula of the form $(1, q_1) \wedge \cdots \wedge (n, q_n)$. We interpret with respect to $[\![o]\!] = \mathcal{P}_{\mathsf{H}}(\Sigma^\bot\mathsf{Trees}^\infty)$, the powerdomain of finite and infinite $\Sigma^\bot$-labelled trees. We take for the non-deterministic meaning d_c of each constant c of rank n the set transformer:

$$d_c(D_1) \cdots (D_n) = \bigsqcup \{\eta(\underline{c}(d_1) \cdots (d_n)) \mid d_i \in D_i\}$$

The choice constant `br` as specified by Definition 5.5.1. The property language is lifted to the nondeterministic setting by defining $[\![q]\!] = \{\mathcal{L}(\mathcal{A}^\perp, q)\}^*$, but the abstract interpretation of the tree constructors remains the same as in the deterministic case. We interpret the choice constant abstractly by the type $\sigma_{\mathtt{br}} = \bigwedge\{q \to q \to q \mid q \in Q\}$.

To see that this is an exact abstraction, we first analyse the behaviour of `br`. We show $\alpha(\eta(D_1 \sqcup D_2)) = \sigma_{\mathtt{br}} \cdot \alpha(\eta(D_1)) \cdot \alpha(\eta(D_2))$ in two applications of (Q-Glb). In the $\leqslant$ direction, let q be such that $\alpha(\eta(D_1)) \leqslant q$ and $\alpha(\eta(D_2)) \leqslant q$, then $\alpha(\eta(D_1)) \sqcup \alpha(\eta(D_2)) \leqslant q$ as required. In the opposite direction, let q be such that $D_1 \sqcup D_2 \in \gamma(q)$. Then, necessarily $D_1 \in \gamma(q)$ and $D_2 \in \gamma(q)$ and hence $\alpha(\eta(D_1)) \leqslant q$ and $\alpha(\eta(D_2)) \leqslant q$. It follows that $\alpha(\eta(D_1 \sqcup D_2)) \leqslant q$ as required. To analyse the behaviour of the tree constructors we first observe that, for each c of arity n:

$$
\begin{aligned}
& \alpha(\eta([\![c]\!](D_1)\cdots(D_n))) \\
= \;& \bigsqcup\{\alpha(\eta(\eta(\underline{c}(d_1)\cdots(d_n)))) \mid d_i \in D_i\} \\
= \;& \bigwedge\{q \mid \forall d_i \in D_i.\, \alpha(\eta(\eta([\![c]\!](d_1)\cdots(d_n)))) \leqslant q\}
\end{aligned}
$$

We aim to show $\alpha(\eta([\![c]\!](D_1)\cdots(D_n))) = \sigma_c \cdot \alpha(\eta(D_1))\cdots\alpha(\eta(D_n))$. Now, to show the $\leqslant$ direction, let q be an element of the meet on the RHS and assume $\delta(q, c) = (1, q_1) \wedge \cdots \wedge (n, q_n)$. Necessarily $\alpha(\eta(D_i)) \leqslant q_i$ for each $i \in [1..n]$ and hence $D_i \subseteq \mathcal{L}(\mathcal{A}^\perp, q_i)$. Consequently, for all $d_i \in D_i$, $[\![c]\!](d_1)\cdots(d_n) \in \mathcal{L}(\mathcal{A}^\perp, q)$ and hence $\eta([\![c]\!](d_1)\cdots(d_n)) \in \mathcal{L}(\mathcal{A}^\perp, q)^*$ so $\alpha(\eta(\eta([\![c]\!](d_1)\cdots(d_n)))) \leqslant q$ whence q is an element of the meet on the LHS. To show the $\geqslant$ direction, assume q is an element of the meet on the LHS. Then, for all $d_i \in D_i$, $\alpha(\eta(\eta([\![c]\!](d_1)\cdots(d_n)))) \leqslant q$ and hence $[\![c]\!](d_1)\cdots(d_n) \in \mathcal{L}(\mathcal{A}^\perp, q)$. Assume $\delta(q, c) = (1, q_1)\cdots(n, q_n)$, since $\mathcal{A}$ is deterministic. It follows that necessarily $d_i \in \mathcal{L}(\mathcal{A}^\perp, q_i)$ for each $d_i \in D_i$. Hence, $\alpha(\eta(D_i)) \leqslant q$ and q is an element of the intersection on the RHS.

Nondeterministic trivial automaton model checking of nondeterministic recursion schemes. In the argument above, we made essential use of the fact that the automaton was deterministic. In fact, the intersection type system induced from the abstract interpretation given above is not complete for non-deterministic trivial automaton model checking in general.

5.5.2 Example. *Consider the following combination of scheme and property automaton:*

$$
\begin{aligned}
S &= \mathtt{a}\, X \\
X &= \mathtt{br}\,\mathtt{b}\,\mathtt{c}
\end{aligned}
\qquad\qquad
\begin{aligned}
\delta(q_0, \mathtt{a}) &= (1, q_1) \vee (1, q_2) \\
\delta(q_1, \mathtt{b}) &= \mathtt{t} \\
\delta(q_2, \mathtt{c}) &= \mathtt{t}
\end{aligned}
$$

over the signature consisting of, in addition to the choice constant, `a` *of arity* 2*,* `b` *and* `c` *of arity 0. Under the concrete interpretation above, this scheme generates two finite trees* `a b` *and* `a c` *which are both accepted by the automaton, yet under the typing induced by the abstract interpretation, S has only an empty intersection of types.*

This example motivates adding extra structure to the type theory in order to be able to better represent non-trivial joins.

Nondeterministic recursion schemes with cases. We take as signature a finite set of zero-arity constants $d_1, \ldots, d_n$, a case analysis constant $\mathtt{case}$ of arity $n+1$ and, as previously, the choice constant $\mathtt{br}$. The term signature is intended as a language for describing non-deterministic computations over some finite enumeration of n values, so let us take $[\![o]\!] = \mathcal{P}_{\mathsf{H}}(\{1, \ldots, n, \bot\})$ to be concrete with $[\![d_i]\!] = \eta(i)$. Let us define a deterministic interpretation of the case construct by:

$$\phi_{\mathtt{case}}(d)(e_1)\cdots(e_n) = \begin{cases} e_i & \text{if } d = i \\ \bot & \text{otherwise} \end{cases}$$

then use this to motivate the definition of the non-deterministic version by :

$$\begin{aligned} [\![\mathtt{case}]\!](D)(E_1)\cdots(E_n) &= \bigsqcup\{\phi_{\mathtt{case}}(d)(e_1)\cdots(e_n) \mid e_i \in E_i \wedge d \in D\} \\ &= (\textstyle\bigcup_{j \in D \cap \mathbb{N}}\{\phi_{\mathtt{case}}(j)(e_1)\cdots(e_n) \mid e_i \in E_i\} \cup \{\bot\})^* \\ &= \bigsqcup\{E_j \mid j \in D \neq \bot\} \end{aligned}$$

Let us now consider the abstract interpretation. We follow Neatherway et al. [2012], and construct union types. We define the base types Q to consist of union types of the form $\bigvee_{i=1}^m t_i$ in which each t_i is a natural number in $\{1, \ldots, n\}$. We order the unions by asserting $\bigvee_{i=1}^m t_i \leqslant \bigvee_{j=1}^n t'_j$ just in case for every $i \in [1..m]$, there is some $j \in [1..n]$ such that $t_i = t'_j$. We interpret $\bigvee_{i=1}^m t_i$ as $\{\bigsqcup_{i=1}^m \eta(t_i)\}^*$. For each data item d_i, we interpret it abstractly by the corresponding type i. We interpret the constant $\mathtt{case}$ by the type:

$$\bigwedge\{\bigvee X \to \sigma_1 \to \cdots \to \sigma_n \to q \mid q \in Q \wedge \sigma_i = \mathcal{F}_i(X, q)\}$$

where $\mathcal{F}_i(X, q) = q$ if $i \in X$ and $\top$ otherwise. Finally we interpret the choice constant by the type $\bigwedge\{q \to q \to q \mid q \in Q\}$.

All that remains is to show exactness. Consider a data item d_i, clearly $\alpha(\eta([\![d_i]\!])) = i = \sigma_{d_i}$. The exactness of the interpretation of the choice constant follows the same argument as in the previous example. Finally consider the case construct. Note that:

$$\begin{aligned} \alpha(\eta([\![\mathtt{case}]\!](D)(E_1)\cdots(E_n))) &= \bigsqcup\{\alpha(\eta(E_j)) \mid j \in D \neq \bot\} \\ &= \bigwedge\{q \mid \forall j \in D.\, \alpha(\eta(E_j)) \leqslant q\} \end{aligned}$$

We aim to show $\alpha(\eta([\![\mathtt{case}]\!](D)(E_1)\cdots(E_n))) = \sigma_{\mathtt{case}} \cdot \alpha(\eta(D)) \cdot \alpha(\eta(E_1))\cdots\alpha(\eta(E_n))$. Then to show that RHS $\leqslant$ LHS, we argue that RHS is a lower bound. So let q be such that, for all $j \in E$, $\alpha(\eta(E_j)) \leqslant q$ (*). Now $\alpha(\eta(E))$ is just $\bigvee E$ so, by the definition of σ_{case}, it remains to show only that $\alpha(\eta(E_j)) \leqslant q$ for each $j \in E$, but this is exactly (*). To show that LHS is a subtype of the RHS, we show that, for all q in the meet on the RHS, $\bigsqcup\{\alpha(\eta(E_j)) \mid j \in D \neq \bot\} \leqslant q$. So let q be in the meet $\sigma_{case} \cdot \alpha(\eta(D)) \cdot \alpha(\eta(E_1))\cdots\alpha(\eta(E_n))$. Then, by definition, it must be the case that, for all $j \in D \neq \bot$, $\alpha(\eta(E_j)) \leqslant q$. Hence, $\bigsqcup\{\alpha(\eta(E_j)) \mid j \in D \neq \bot\} \leqslant q$.

5.6 Exact abstraction at higher types

In this section we are interested in lifting Theorem 5.4.2 to higher-kinds, in order to give locally justified characterisations of all property checking problems for a given term and property language. We first show that the hypotheses required in order to obtain the conclusion of that theorem are, on their own, not enough to obtain exact characterisations at higher-kinds and we contrast this with the case for safety. To rectify the situation we describe an additional requirement, which is that the abstract property space at ground kind contains "no junk", and this leads to exact characterisations at higher-kinds.

We begin by showing that after fixing a term and property language and an exact abstract interpretation, it may still be the case that $\alpha(\eta([\![t]\!])) \neq \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t)$ when closed term t is of kind κ for some κ of arrow kind.

5.6.1 Example. *Consider a term language consisting of any signature with interpretation in the one point domain* $[\![o]\!] = \{\bot\}$*. The corresponding concrete property space is* $\{\{\bot\}\}$*. Consider a property language over lattice of intersection types* $\{\top, q\}$ *with interpretation determined uniquely as* $[\![q]\!] = \{\bot\} = [\![\top]\!]$*. Then* α *is forced to map* $\{\bot\}$ *to* q*. Assume that there is an exact abstraction for the choice of signature, obviously many choice exist. Now consider term* $t = \lambda x.\, x$ *of kind* $o \to o$*. It follows that* $\bigwedge \mathbb{T}(\varnothing)(t) = q \to q$*, but* $\alpha(\eta([\![\vdash t : o \to o]\!])) = \top \to q$ *and* $\top \to q < q \to q$*.*

So, for characterising problems at higher kind, something is missing. However, note that whatever more is needed, it is not likely to be a restriction on the choice of constants or their abstract interpretation since the conclusion of this example does not depend upon this information. The fact that more is needed in order to obtain characterisations of property checking problems at higher-kind is in contrast to the case for merely soundly approximating problems at higher kind which carries over directly thanks to the following result:

5.6.2 Lemma. *For all* $d \in [\![\kappa]\!]$ *and* $\theta :: \kappa$*,* $\alpha_\kappa(\eta_\kappa(d)) \leqslant \theta$ *iff* $d\ R^\kappa_\leqslant\ \theta$*.*

Proof. By induction on κ. The base case is by definition, so consider κ of the form $\kappa_1 \to \kappa_2$.

- In the forward direction, let $e\ R^{\kappa_1}_\leqslant\ \theta'$. It follows from the induction hypothesis that $\alpha(\eta(e)) \leqslant \theta'$. We aim to show $\alpha(\eta(d(e))) \leqslant \theta \cdot \theta'$ from which the result will follow from the induction hypothesis. To do so, observe that since $e \in [\![\theta' :: \kappa_1]\!]$ and, by assumption, $d \in [\![\theta :: \kappa_1 \to \kappa_2]\!]$, then therefore $d(e) \in [\![\theta \cdot \theta' :: \kappa_2]\!]$.

- In the backward direction, observe that θ is necessarily of the form $\bigwedge_{i=1}^n (\sigma_i \to \tau_i)$, so let $i \in [1..n]$ and we aim to show $d \in [\![\sigma_i \to \tau_i :: \kappa_1 \to \kappa_2]\!]$. Let $e \in [\![\sigma_i :: \kappa_1]\!]$, then $\alpha(\eta(e)) \leqslant \sigma_i$ and, by the induction hypothesis, $e\ R^{\kappa_1}_\leqslant\ \sigma_i$. Hence, by assumption, $d(e)\ R^{\kappa_2}_\leqslant\ \theta \cdot \sigma_i$. It follows from the induction hypothesis that therefore $\alpha(\eta(d(e))) \leqslant \theta \cdot \sigma_i$ and hence $d(e) \in [\![\theta \cdot \sigma_i :: \kappa_2]\!] \subseteq [\![\tau_i :: \kappa_2]\!]$.

□

Hence, from the view of the situation given to us by the logical relations, the problem is that whilst $R^{\kappa}_{\leqslant}$ characterises sound abstractions, $R^{\kappa}_{\geqslant}$ does not characterise complete ones. Intuitively, the reason is as follows. Let us think of sound abstractions of some element d, i.e. those θ such that $\alpha(\eta(d)) \leqslant \theta$, as being over-approximations, and let us think of complete abstractions as being under-approximations. Then whilst it is true that to be a sound abstraction at higher-type is exactly to map over-approximations to over-approximations; mapping under-approximations to under-approximations does not entail being an under-approximation.

The problem of Example 5.6.1 is that, as an abstraction, the intersection type $\top$ is essentially redundant. Every semantic entity $d \in D$ is already better described, in the sense of lower in the subtype order, by some other type which in this case is necessarily q. Hence, we take as our requirement that $\alpha \circ \eta$ is a surjection. This ensures that the abstract domain contains "no junk" with respect to those properties that are the collecting semantics of some term. Note that this condition implies the often advocated requirement in abstract interpretation that $\langle \alpha, \gamma \rangle$ is a Galois surjection. Since, in our case, the abstract domain is quotiented with respect to the type theory, the "no junk" assumption can often be ensured by manipulating this ordering. Let us call a property language universal when it satisfies this requirement.

5.6.3 Definition. Fix a term language over a domain D. A property language over D is said to be *universal* just if the induced composite $\alpha_o \circ \eta_o$ is surjective.

We aim to prove the following lemma which states that under the requirements discussed above and for each kind κ, $(R^{\kappa}_{\leqslant} \cap R^{\kappa}_{\geqslant})$ is a sub-graph of $\alpha_\kappa \circ \eta_\kappa$:

5.6.4 Lemma. *Fix a term language and a universal property language and some exact abstract interpretation. Then, for all $\theta :: \kappa$, $d\ (R^{\kappa}_{\leqslant} \cap R^{\kappa}_{\geqslant})\ \theta$ implies $\alpha_\kappa(\eta_\kappa(d)) = \theta$.*

The main theorem is then a straightforward corollary.

5.6.5 Theorem (Exact abstraction at all types)**.** *Fix a term language and a universal property language and an exact abstract interpretation. Then, for all closed terms t of kind κ:*

$$\alpha_\kappa(\eta_\kappa(\llbracket t \rrbracket)) = \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t)$$

Proof. Since the abstract interpretation is exact, it follows from Lemmas 5.4.7 and 5.4.8 that for all closed terms t of kind κ, $\llbracket t \rrbracket\ (R^{\kappa}_{\leqslant} \cap R^{\kappa}_{\geqslant}) \bigwedge \mathbb{T}(\mu(\mathsf{Shrink}))(t)$. The result follows from Lemma 5.6.4. ☐

However, before we prove the key lemma, let us see two applications of the theorem which are showing the characterisation and hence decidability of the problems described in Examples 5.2.18 and 5.2.19.

5.6.6 Example (Arithmetical recursion schemes parity checking problem)**.** *First observe that the property language is universal with respect to the term language, since $\alpha(\eta(0)) = \mathsf{even}$, $\alpha(\eta(1)) = \mathsf{odd}$, $\alpha(\eta(err)) = \top$ and $\alpha(\eta(\bot)) = \mathsf{even} \wedge \mathsf{odd}$. An exact abstract interpretation arises by setting $\sigma_{\mathtt{zero}} = \mathsf{even}$, the unary functions as follows:*

$$\begin{aligned}\sigma_{\texttt{succ}} &= (\mathsf{even} \to \mathsf{odd}) \\ &\wedge\ (\mathsf{odd} \to \mathsf{even}) \\ &\wedge\ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \\ &\wedge\ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{even})\end{aligned}$$

$$\begin{aligned}\sigma_{\texttt{isEven}} &= (\mathsf{even} \to \mathsf{even}) \\ &\wedge\ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \\ &\wedge\ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{even})\end{aligned}$$

We interpret binary addition by:

$$\begin{aligned}\sigma_{\texttt{add}} &= (\mathsf{even} \to \mathsf{odd} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even} \to \mathsf{odd}) \\ &\wedge\ (\mathsf{odd} \to \mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{even} \to \mathsf{even}) \\ &\wedge\ (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{odd}) \wedge (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{even}) \\ &\wedge\ (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \wedge (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{even})\end{aligned}$$

and binary multiplication by:

$$\begin{aligned}\sigma_{\texttt{mult}} &= (\mathsf{even} \to \mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{odd} \to \mathsf{even} \to \mathsf{even}) \\ &\wedge\ (\mathsf{odd} \to \mathsf{odd} \to \mathsf{odd}) \wedge (\mathsf{even} \to \mathsf{even} \to \mathsf{even}) \\ &\wedge\ (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{odd}) \wedge (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{even}) \\ &\wedge\ (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \wedge (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{even})\end{aligned}$$

It is straightforward to verify, via simple case analyses, that the abstract interpretation is exact and hence the parity checking problem is characterised by the induced intersection refinement type system. Consequently, the problem is decidable.

5.6.7 Example (Arithmetical recursion schemes strictness checking problem)**.** *First observe that the property language is universal since $\alpha(\eta(\bot)) = *$ and $\alpha(\eta(0)) = \top$. We interpret abstractly* `zero` *by $\top$ and each first-order constant c of arity n by the type $* \to \cdots \to * \to *$. It is easy to see that this abstract interpretation is exact and hence the strictness checking problem is characterised by the induced intersection refinement type system. Consequently, the problem is decidable.*

We now come back to the proof of Lemma 5.6.4. As a first step, we show that the property of $\alpha_o \circ \eta_o$ being a surjection at ground kind is enough to guarantee surjectivity at all higher-kinds. We construct a family of continuous functions $\delta_\kappa(\theta)$, one for each type θ, such that $\alpha_\kappa o(\eta_\kappa(\delta_\kappa(\theta))) = \theta$.

5.6.8 Definition. Fix a term language and a universal property language. We define a (continuous) mapping from intersection refinement types $\theta :: \kappa$ to elements of the model $\delta_\kappa(\theta)$. Since $\alpha_o \circ \eta_o$ is surjective, there is some choice c in $\Pi_{\theta \in \mathbb{I}(o)}\{d \in [\![o]\!] \mid \alpha_o(\eta_o(d)) = \theta\}$ which will suffice at ground kind, the rest follows by induction on κ.

$$\begin{aligned}\delta_o(\theta) &= c(\theta) \\ \delta_{\kappa_1 \to \kappa_2}(\theta)(x) &= \delta_{\kappa_2}(\theta \cdot \alpha_{\kappa_1}(\eta_{\kappa_1}(x)))\end{aligned}$$

Before we prove that this family has the desired property, let us note a fact about type application which will be useful in what follows.

5.6.9 Lemma. *If, for all $j \in [1..m]$, $\sigma \cdot \sigma_j \leqslant \tau_j$, then $\sigma \leqslant \bigwedge_{j=1}^{m}(\sigma_j \to \tau_j)$.*

Proof. Assume the hypothesis, then let $j \in [1..m]$. Necessarily σ is of the form $\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i)$ and it follows from the assumption that there is some $N \subseteq [1..n]$ such that $\forall i \in N.\, \sigma_j \leqslant \sigma_i$ and $\bigwedge_{i \in N} \tau_i \leqslant \tau_j$. Hence, the result follows from Lemma 3.1.4. □

Now we are in a position to show that, in the setting of a universal property language, higher-kind abstractions that map under-approximations to under-approximations are under-approximations themselves. We do so simultaneously with showing that $\delta_\kappa(\theta)$ is related by both $R^\kappa_\leqslant$ and $R^\kappa_\geqslant$ to θ. It follows from Lemma 5.6.2 that $\delta_\kappa(\theta)$ is therefore an element of the model whose best abstraction $\alpha(\eta(\delta_\kappa(\theta)))$ is exactly θ.

5.6.10 Lemma. *Fix a term language and a universal property language. Then:*

(i) For all $\theta :: \kappa$: $d\ R^\kappa_\geqslant\ \theta$ implies $\theta \leqslant \alpha_\kappa(\eta_\kappa(d))$

(ii) For all $\theta :: \kappa$: $d_\kappa(\theta)\ (R^\kappa_\leqslant \cap R^\kappa_\geqslant)\ \theta$.

Proof. By induction on κ.

- When $\kappa = o$, note that (i) follows by definition. To see (ii) simply observe that $\theta = \alpha_o(\eta_o(\delta_o(\theta)))$ and the result follows by definition.

- When κ is of the form $\kappa_1 \to \kappa_2$, we proceed as follows.

  For (i), assume $d\ R^{\kappa_1 \to \kappa_2}_\geqslant\ \theta$ and let $\alpha(\eta(d))$ be of the form $\bigwedge_{i=1}^{n} \sigma_i \to \tau_i$. So let $i \in [1..n]$ and $d \in \gamma(\sigma_i \to \tau_i)$ (*). We aim to show $\theta \cdot \sigma_i \leqslant \tau_i$ and thus conclude by Lemma 5.6.9. It follows from the induction hypothesis that $\delta_{\kappa_1}(\sigma_i)\ R^{\kappa_1}_\geqslant\ \sigma_i$ and hence $d(\delta_{\kappa_1}(\sigma_i))\ R^{\kappa_2}_\geqslant\ \theta \cdot \sigma_i$. Consequently, and also from the induction hypothesis, $\theta \cdot \sigma_i \leqslant \alpha_{\kappa_1}(\eta_{\kappa_1}(d(\delta_{\kappa_1}(\sigma_i))))$, so it remains to show only that $\alpha_{\kappa_1}(\eta_{\kappa_1}(d(\delta_{\kappa_1}(\sigma_i))) \leqslant \tau_i$; but this follows from (*) since the induction hypothesis gives $\delta_{\kappa_1}(\sigma_i)\ R^{\kappa_1}_\leqslant\ \sigma_i$ and Lemma 5.6.2 thus gives $\delta_{\kappa_1}(\sigma_i) \in \gamma(\sigma_i)$.

  For (ii), note that κ is generally of the form $\kappa_1 \to \cdots \to \kappa_n \to \kappa$. We show membership in both relations separately. To see that $\delta_\kappa(\theta)\ R^\kappa_\leqslant\ \theta$ let $e_i\ R^{\kappa_i}_\leqslant\ \theta_i$ for each $i \in [1..n]$. It follows from Lemma 5.6.2 that therefore $\alpha_{\kappa_i}(\eta_{\kappa_i}(e_i)) \leqslant \theta_i$ for each $i \in [1..n]$. Since, furthermore:

  $$\delta_\kappa(\theta)(e_1) \cdots (e_n) = \delta_o(\theta \cdot \alpha_{\kappa_1}(\eta_{\kappa_1}(e_1)) \cdots \alpha_{\kappa_n}(\eta_{\kappa_n}(e_n)))$$

  it follows from the induction hypothesis part (ii) that:

  $$\alpha_o(\eta_o(\delta_\kappa(\theta)(e_1) \cdots (e_n)))) \leqslant \theta \cdot \alpha_{\kappa_1}(\eta_{\kappa_1}(e_1)) \cdots \alpha_{\kappa_n}(\eta_{\kappa_n}(e_n))$$

  and hence $\alpha_o(\eta_o(\delta_\kappa(\theta)(e) \cdots (e_n)))) \leqslant \theta \cdot \theta_1 \cdots \theta_n$. The argument for showing $\delta^\kappa(\theta)\ R^\kappa_\geqslant\ \theta$ is analogous except we appeal to the induction hypothesis clause (i) rather than Lemma 5.6.2.

□

We can now prove Lemma 5.6.4 quite straightforwardly.

Proof. Assume $d\ (R^{\kappa}_{\leqslant} \cap R^{\kappa}_{\geqslant})\ \theta$, then it follows from Lemma 5.6.2 that $\alpha_\kappa(\eta_\kappa(d)) \leqslant \theta$ and from Lemma 5.6.10 that $\theta \leqslant \alpha_\kappa(\eta_\kappa(d))$, as required. □

5.7 Related work

Our work is a view of the intersection type approach to higher-order model checking as a kind of exact abstract interpretation and, furthermore, a generalisation of that situation to safety checking problems that go beyond the verification of trees that are accepted by trivial tree automata. To the best of our knowledge, this is the first work to study higher-order model checking in an abstract interpretation setting and the first to place higher-order model checking as an instance of a more general framework for property checking problems decidable on classes of recursion scheme. Although we do not know of any work with similar goals, the techniques we have used in order to achieve our results are well understood; we will give a brief overview of the most relevant literature.

Exact abstract interpretations. As already mentioned, abstract interpretation was pioneered by Cousot and Cousot [1977, 1979] and exactness has been studied in the setting of abstract interpretation by many authors, including in these early papers. Of particular note is that of Giacobazzi, Ranzato, and Scozzari [2000], who present a comprehensive analysis of exactness in terms of the completeness of the underlying abstract domains and which also includes many pointers to other literature in the area. Let us only remark that our definition of exactness is a slight variation on the standard one, since we wish to pay explicit attention to the role of the collecting semantics in order to obtain an easier to prove, though less general, condition. This is discussed further in Section 6.2.

Intersection types as abstract interpretations. A classic paper on a view of types as abstract interpretations is Cousot's paper [Cousot, 1997], which presents a picture of type systems of all sorts as instances of abstract interpretation and relates them in a unified framework. This paper does not discuss traditional intersection type systems specifically, but it includes a treatment of a system which has a kind of infinite intersection. Three separate sets of authors have come up with frameworks for program analysis based on intersection types during the early nineties which are very related to our own, namely Burn [1992], Coppo and Ferrari [1993] and Hankin and Métayer [1994]. All are abstract interpretations of a denotational semantics of a lambda calculus and all have, in common with this work, the feature that the correctness of an abstraction of whole programs is obtained as a consequence of showing the correctness of an abstraction of the constants over which the program is built. Our work can also be cast as a kind of framework for building program analyses, and then it is clear that the central difference between it and that of the others is that our work is focused on guaranteeing both soundness and

completeness of the program analysis, whereas all the other work stops at soundness. Of course, this follows from our somewhat unique motivation in trying to generalise the key features of higher-order model checking. However, in certain other respects the work of these three authors goes further than our work.

Burn [1992] explores a more expressive type system which, in addition to arrows and conjunctions, includes type constructors for disjunctions and products. It seems very likely that our result could be extended to such a system. We already introduced disjunctions as a set of atoms in our characterisation of nondeterministic recursion schemes with case; but a more principled incorporation would likely bring additional benefits. Although this was not a topic explored by Burn, one might hope to be able to, for example, lift abstract interpretations from a determinstic setting to the nondeterministic one (together with the accompanying proofs of exactness) in a systematic way.

Coppo and Ferrari [1993] allow for a more liberal subtype theory and their work is also notable for constructing program analyses over untyped lambda calculus. Of course, the decidability guarantees given by our theorem depend upon the fact that programs are restricted by a simple type discipline. On the one hand, it seems desirable for us to allow a greater degree of freedom with the specification of the subtype theory so that, for example, more of the abstract domain can become identified under the quotient. On the other hand, many of the proofs exploit the inversion lemmas which themselves rely on the particular way in which the theory is generated from the specification.

The work of Hankin and Métayer [1994] follows up the work of Burn, by carving out a restriction of his type system for which they give a type assignment algorithm. The restriction is essentially to remove disjunctions, which also helps them to consider a generalised class of properties. It seems quite clear that this work could follow theirs in considering the same class of properties and hence be more broadly applicable. We did not do that here because we wanted to retain some concreteness to the work and because we were primarily interested in capturing the essence of the trivial automaton model checking scenario.

Higher-order abstract interpretation. Much work was done on the abstract interpretation of higher-order programs during the late eighties and early nineties, see for example the volume [Abramsky and Hankin, 1987]. However, the usual approach based on powerdomains was criticised by Cousot and Cousot [1994] since the requirement that properties of functions were necessarily expressed as functions over powerdomains did not allow for a very expressive space of properties. There is no such restriction in this work, which places properties built from powerdomains in a traditional Galois connection based setting, where properties of functions are (closed) sets of functions, as expected.

Our main tool for establishing results has been logical relations, which were first advocated as a general approach to the abstract interpretation of models of simply typed lamdba calculi by Abramsky [1990]. Abramsky's work, in common with much of the early literature on the abstract interpretation of higher-order programming languages used finite domains and continuous functions as an abstraction. Following Abramsky's work on domain theory in logical form [Abramsky, 1991], Jensen [1991] showed, for the particular problem of strictness analysis, that such abstractions are isomorphic to filters of intersec-

tion types. Jensen's result about the equivalence of these two kinds of abstract domain sheds light on the relationship between our Lemma 5.6.2 and the paper of Backhouse and Backhouse [2004]. In that paper it is shown that certain logical relations uniquely determine Galois connections at higher-type. The logical relations considered there include a version of our relation $R^{\kappa}_{\leqslant}$ (modulo our use of η) but the induced Galois connection is between sets of functions, so our lemma can be seen as the analogous result for intersection type abstractions.

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we have presented a subtyping extension of Kobayashi's intersection type system for higher-order model checking. By restricting subsumption so that it can be applied only to arguments during application, we have ensured that the system retains the important theoretical and practical attributes of the original. In particular, we have shown that there is a straightforward and efficient algorithm for type checking, that it preserves intersection type refinement and, despite its additional permissiveness, can still be used to characterise the (co-)trivial automaton model checking problem for recursion schemes.

The greater flexibility afforded by subtyping is essential (see, e.g., the proofs of Lemmas 4.3.18 and 4.3.13) in the justification for our decision procedure for model checking. This procedure is a new, fixed-parameter polynomial time algorithm for the (co-)trivial automaton model checking problem for recursion schemes. By starting from a relatively coarse but cheap abstraction and letting types direct its refinement, detail is added only when needed. Furthermore, at termination, the type environments constructed certify the correctness of the solution. The algorithm breaks new ground amongst intersection-type based algorithms by showing how it is possible to use types to describe both the acceptance and rejection behaviours of the instance, thus minimising the amount of work done by exploring only those parts of the configuration graph that are not known to be one or the other. Furthermore, our implementation, Preface, and empirical evaluation shows the algorithm to be well ahead of the state of the art.

Our decision procedure is essentially a type assignment algorithm, i.e. an automated method for associating certain syntactic expressions, namely intersection types built over some atoms, to certain other syntactic expressions, namely terms built over some constants. By taking the atoms to represent the language of trees accepted from particular states of an automaton and by taking the constants to represent tree constructors, it is the fundamental result of the intersection type approach to higher-order model checking that such type assignment is equivalent to model checking. Our study of intersection

type assignment as exact abstract interpretation gives a much broader perspective on the computational situations for which intersection type assignment is sound and complete. We show that the work is directly relevant to current model checking practice by using our framework in order to recover, in a very concise and straightforward way, the soundness and completeness of certain extended higher-order model checking situations in the literature. Furthermore, we look beyond the typical ground-type, whole-program view of model checking by characterising also those situations in which terms are higher-type and properties are sets of functions.

6.2 Discussion

Model checking via type directed abstraction refinement

Based on evaluation over the benchmarks currently found in the literature, our algorithm scales much more robustly than others and it is the first to scale convincingly beyond a few hundred lines. However, although we used the full corpus of model checking instances that are currently available within the community, this set of examples (and hence our evaluation) is, in two areas, a little weaker than we would like.

The first is that the very large model checking instances, those containing more than 1000 rules, are all of the same kind. These very large instances are all derived from the same family, defined by Kobayashi [2013]. Hence, although it is clear that our algorithm scales significantly more robustly than its contemporaries, it is not clear that it scales *robustly* at these very large instances, since there is no evidence beyond this very uniform kind. However, the weakness of these instances is, to an extent, ameliorated by their construction. This particular family was designed to stress-test higher-order model checkers and there is some strong evidence to support the case that the family succeeds in this, since (as demonstrated by our comparison) all other model checkers find these examples to be particularly difficult.

The second is that all of the very large model checking instances are yes-instances. Hence, our evaluation has not been particularly comprehensive in comparing the performance on no- versus yes-instances. Good performance on no-instances can be very important in practice because CEGAR-style algorithms may need to model check many no-instances arising from approximations before finally checking a single yes-instance that signals termination. Unfortunately, verification tools that build upon higher-order model checking and which might provide a source of such instances are not, at the present time, mature enough to be able to handle large enough inputs in order to generate large, yet non-trivial (in the sense of being too easy), examples.

In all the examples on which the tool performs badly, the unacceptably high time taken to solve the problem instance stems from the algorithm working through a very large number of iterations. As we discussed in the analysis in Section 4.6, this seems to indicate a lack of good type information being extracted from each individual iteration. There is some flexibility to make changes to the algorithm in order to extract more or less type information. By inspecting the proof of Lemma 4.3.14, one can see that progress

is guaranteed so long as rejection type information is extracted from a single rejecting leaf on each round. As specified, the algorithm goes much further than this basic requirement, extracting rejection types from the entire rejecting region and extracting acceptance types from the accepting region. It could potentially go much further, essentially by extending the size of the regions, but this will compromise the time taken to compute the limits of larger regions which, in going beyond the current specification, will no longer be straightforward.

Intersection types as exact abstract interpretations

Our work studies the landscape of ground-type property checking problems which are characterised by intersection type assignment besides the classical higher-order model checking problem. Furthermore, it gives an answer to the question of what kinds of higher-type property checking problems are characterised by intersection type assignment, which has not even been formulated precisely before in the literature. However, the answer we have obtained so far is quite a narrow one. In particular, the condition that $\alpha \circ \eta$ be surjective is strong. Many situations of practical interest may not satisfy this condition, although we have given two possibilities: the parity checking and strictness checking families of Examples 5.2.18 and 5.2.19 respectively.

It is our belief that it is possible to give a more satisfactory answer which requires only the condition that α is surjective. This condition, which is often advocated in abstract interpretation, is very easy to satisfy since one can exploit the quotient structure on the intersection type abstract property space in order to guarantee it “for free”. To obtain this more relaxed condition, one would work not with logical relations between $[\![\kappa]\!]$ and $\mathbb{I}(\kappa)$, but between $\mathcal{P}_{\mathsf{H}}([\![\kappa]\!])$ and $\mathbb{I}(\kappa)$ — in other words, relating concrete properties and abstract ones rather than elements of the model and abstract properties. In doing this, the associated condition required in order for an abstract interpretation to be exact would also become more standard, requiring, for all constants c:

$$\alpha(\eta([\![c]\!]) \bullet (D_1) \cdots (D_n)) = \sigma_c \cdot \alpha(D_1) \cdots \alpha(D_n)$$

in which $\bullet$ is the obvious application operator between properties, defined by $F \bullet X = \bigsqcup\{\eta(f(x)) \mid f \in F \text{ and } x \in X\}$. However, this condition is also a *stronger* requirement and, in particular, is not generally satisfied by the usual formulation of (non-deterministic) higher-order model checking! To see why, recall Example 5.5.2 and let $D = \gamma(q_1) \sqcup \gamma(q_2)$. It follows that $\alpha(D) = \top$ and so $\sigma_c \cdot \alpha(D) = \top$, but $\alpha(\eta([\![c]\!]) \bullet D) = q_0$. Since our primary goal in this work was to capture the essential properties of the intersection type approach to higher-order model checking, we have not pursued this direction here.

6.3 Future directions

We have already suggested some ways in which this work would benefit from further investigation, but there are many other interesting directions.

Inferring types for higher-order terms. The first, and perhaps most obvious, is to bring the two main contributions closer together by extending the algorithm to infer types for higher-order terms. One way of doing this might be to "ground" higher-type configurations by applying them to fresh variables. For example, to check that some term $\mathcal{N} \vdash t : o \to o$ has type $q_1 \to q_2$, one might root the configuration graph at $(t\,y, q_2)$ with $\mathsf{A}(y) = q_1$. It seems likely that, in this way, it would also be possible to deal with higher-order constants.

Inferring types beyond (co-)trivial properties. A different extension of the algorithm considers a greater variety of properties. Theoretically, the obvious extension is to all modal mu-calculus definable properties. The aim would be to maintain the same overall structure of the algorithm but incorporate general parity conditions. To do this, we expect that the language of intersection types would need to be extended to incorportate priorites and employ the more general type assignment system of Kobayashi and Ong [2009]. Then, we expect to be able to define the accepting and rejecting regions in such a way that they correspond to winning regions for Eloïse in parity games induced by $\mathcal{A}$ and $\mathcal{A}^c$. However, parity games are difficult to solve and so, from a practical perspective, the trade off between complexity and expressiveness might be better settled at Büchi properties or properties expressible in the alternation free mu-calculus. In all these cases, the interest will be in adapting the algorithm to work with parity conditions.

Inferring types for functional programs. Our algorithm is almost a general type inference procedure for functional programs. If it could be extended so as to type a realistic functional programming language without losing any of its good features, then we believe it would be a useful tool for programmers. The algorithm is incremental: it infers increasing amounts of type information round by round and, at all times, the type information is "correct", in the sense that the environments that contain it are (co-)consistent. Consequently, although it is slower than a traditional compiler type checker, it could be neatly incorporated into an integrated development environment (IDE), running in the background where it does not occupy the programmer's focus. At the end of each iteration, the current state of the context could be made available to the programmer in the IDE. When the programmer makes changes, the types may no longer be consistent but it is easy to eliminate those that should be invalidated by type-checking with respect to the new definitions.

Their are two main ways in which the algorithm would need to be extended in order to serve as a type inference procedure for functional programs. The first is that it would have to handle higher-order constants so as to model constructs like pattern matching. It seems likely that the same method we proposed for inferring types of higher-order terms could be used for this purpose. The second is that the simple typing (kinding) discipline

would have to be relaxed. A step towards supporting more expressive type systems is to properly support ML-style polymorphism. For this, one could imagine refinements of universal types. For example, a refinements of $\forall\alpha,\ \alpha \to o \to \alpha$ would be types of the form $\forall\alpha,\ \alpha \to \bigwedge_{i=1}^{n} q_i \to \alpha$. By only allowing refinements of the atoms, and not of quantified variables, one could maintain decidability by ensuring a finite space of types.

Improving type rejection extraction. As we pointed out in the previous section, the algorithm requires further evaluation, especially with regard to large no-instances. The results of this evaluation should be used to inform the design of a more sophisticated method of type extraction, which avoids the very high number of iterations that is currently required in order to solve e.g. those examples in Table 4.5. Since this will almost certainly require a more detailed look at the shape of the rejecting region, this should integrate the design of a method of counterexample extraction. To see that these two aspects are closely related, observe that when solving a no-instance, after only one iteration the counter-example trace that witnesses the negative result is already present in the configuration graph. However, it is not "seen" by our method since the algorithm does not look beyond what is contained in the rejecting region and, consequently, it may take hundreds of iterations to find it.

Connecting with our PMRS verification pathway. One possible source of interesting model checking instances with which to further the evaluation is our verification method for functional programs with pattern matching which we reported in [Ong and Ramsay, 2011]. We have, until now, not been able to properly evaluate this verification method for want of a more scalable model checker. However, to do so will require the extension of the algorithm so that it is able to generate counterexample traces, which are essential for guiding the refinement in that work. Furthermore, based on our experience with HORSC in [Neatherway et al., 2012], it seems likely that our model checking algorithm will need to be optimised for working with the weak pattern matching construct, since it inevitably introduces a large number of non-deterministic choices (one for each case).

Weakening the surjectivity requirement. There are two orthogonal avenues to explore concerning the second main contribution. The first is to work out the details of the proposal sketched in the previous section that allows for the weaker hypothesis: α is surjective. This would give a much more satisfactory outcome to the investigation of higher-type properties. However, because this leads to a stronger (but more standard) condition on the exactness of the abstract interpretation we believe that a better setting for the work would be an intersection type system with ground kind unions. In such a type system, it seems likely that one could design type assignment in such a way that the induced type application operator $\cdot$ would be additive (in the sense of preserving all joins, rather than just those which are the limits of directed sets). The hope would be that, in this setting, the stronger condition on exactness would be implied by the weaker one,

since (in case c is a unary constant):

$$\begin{aligned} \alpha(\eta(\llbracket c \rrbracket) \bullet D) &= \alpha(\bigsqcup\{\eta(f(d)) \mid f \in \eta(\llbracket c \rrbracket) \text{ and } d \in D\}) \\ &= \bigsqcup\{\alpha(\eta(f(d))) \mid f \in \eta(\llbracket c \rrbracket) \text{ and } d \in D\} \\ &= \bigsqcup\{\alpha(\eta(\llbracket c \rrbracket(d))) \mid d \in D\} \\ &= \bigsqcup\{\sigma_c \cdot \alpha(\eta(d)) \mid d \in D\} \\ &= \sigma_c \cdot \bigsqcup\{\alpha(\eta(d)) \mid d \in D\} \\ &= \sigma_c \cdot \alpha(D) \end{aligned}$$

As mentioned when discussing the related work, an investigation along these lines might also shed some light on a method for lifting existing abstractions exact for the deterministic setting to ones exact for the non-deterministic setting.

Characterising problems of mu-calculus definable properties. The second avenue to explore would be in generalising the properties from safety properties to also include liveness properties and then, ideally, to a class which includes all mu-calculus definable properties. This would then form a quite comprehensive explanation of what it means to do recursion scheme model checking at higher-type.

Appendix A

Evaluation: Complete Results

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

The tools we compared were Preface 1.2, HorSat and HorSatT 1.0, GTRecS2 3.2, C-SHORe 1.1, TravMC 2.0 and TRecS 1.35. Since GTRecS2 has two modes of which one or the other is better depending on whether the instance is positive or negative, we ran just the right one on each example in order to give the best possible results. We ran Preface and TravMC on Mono 3.0.10, HorSat, HorSatT, GTRecS2 and TRecS were compiled under OCaml 3.11.2 and C-SHORe was run on the Oracle Java SE Runtime 1.7.0_25.

The general form of the examples listed in Table 4.5 is as follows. The scheme tn is defined schematically as follows:

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

```
  Not (
    if g (Bits false false false g) (L2 (L2 g))
  )
L2 g =
  Not (
    if g (Bits false false false g) (L3 (L3 g))
  )

  ...

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

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

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

Benchmark	Rules	Order	Decision	Preface	TRecS
ack	17	2	A	0.38	0.01
copy_intro	33	2	A	0.38	0.02
e-fact	12	2	A	0.49	0.01
e-simple	3	1	A	0.36	0.00
fact_notpos	25	4	A	0.47	0.01
harmonic	60	4	A	0.39	0.03
hors	9	2	A	0.37	0.00
isnil	19	3	A	0.38	0.01
iter	49	4	A	0.38	0.03
l-zipmap	57	2	A	0.40	0.06
l-zipunzip	31	2	A	0.38	0.02
length	29	3	A	0.38	0.01
max	5	1	A	0.37	0.00
mc91	29	2	A	0.51	0.03
mem	52	4	A	0.38	0.03
mult	18	2	A	0.38	0.02
nth	49	3	A	0.48	0.04
nth0	38	3	A	0.48	0.02
r-lock	5	1	A	0.36	0.00
sum	10	2	A	0.37	0.01
sum_intro	10	2	A	0.37	0.01
fact_notpos-e	24	4	R	0.40	0.00
harmonic-e	24	3	R	0.41	0.00
map_filter-e	64	5	R	0.53	0.01
fold_left	65	4	A	0.39	0.03
fold_right	65	4	A	0.39	0.03
forall_eq_pair	66	4	A	0.39	0.03
forall_leq	66	4	A	0.39	0.03
a-cppr	74	3	R	0.38	0.01
search-e	96	5	R	0.90	0.01
search	119	4	A	0.46	1.04
map_filter	143	5	A	0.51	0.13
risers	148	5	A	0.44	0.33
r-file	156	2	A	0.82	1.50
fold_fun_list	197	6	A	0.44	0.89
zip	210	3	A	0.58	15.10

Table A.1: Benchmarks arising from MoCHi.

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

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

Benchmark	Rules	Order	Decision	Preface	TRecS
accfile	2	1	A	0.37	0.00
app__fo	7	2	A	0.38	0.02
appx__fo	6	2	A	0.39	0.02
flatten	3	1	A	0.37	0.01
gapid	11	3	A	0.50	0.02
mult	3	1	A	0.39	0.00
remove__b	4	2	A	0.37	0.00
replace	8	3	A	0.44	0.01
rev	2	1	A	0.37	0.01
split2__coerce1	28	2	A	0.42	0.03
split2__coerce2	28	2	A	0.43	0.03
split2__main	5	2	A	0.39	0.01
split__coerce1	28	2	A	0.43	0.03
split__coerce2	28	2	A	0.43	0.03
split__coerce3	28	2	A	0.42	0.03
split__coerce4	28	2	A	0.42	0.03
split__main	5	2	A	0.39	0.01
xhtmlf__drop-a	15	1	A	1.07	0.33
xhtmlf__id	2	1	A	1.10	0.58
xhtmlf__remove-meta	14	1	A	0.87	0.20
xhtmlm__drop-a	2	1	A	0.56	0.15
xhtmlm__id	2	1	A	0.52	0.07
xhtmlm__remove-meta	2	1	A	0.52	0.06
xhtmls__drop-a	2	1	A	0.42	0.02
xhtmls__id	2	1	A	0.40	0.11
xhtmls__remove-meta	2	1	A	0.40	0.01
xmarkq1__coerce1	100	2	A	0.66	0.20
xmarkq1__main	13	2	A	0.55	0.07
xmarkq2__coerce1	95	1	A	0.47	0.05
xmarkq2__coerce2	95	1	A	0.48	0.05
xmarkq2__main	10	1	A	0.42	0.02
xml__rep1	7	3	A	0.38	0.01
jwig-cal2__coerce1	35	1	R	0.43	0.01
jwig-guess2__coerce1	25	2	R	0.42	0.01
mapfilt__main	6	2	R	0.38	0.00
split2__coerce3	28	2	R	0.44	0.01
split2__coerce4	28	2	R	0.44	0.01
xhtmlf__drop-div	15	1	R	1.43	0.03
xhtmlf__remove-div	14	1	R	0.89	0.18
xhtmlm__drop-div	2	1	R	0.62	0.02
xhtmlm__remove-div	2	1	R	0.52	0.01
xhtmls__drop-div	2	1	R	0.42	0.01
xhtmls__remove-div	2	1	R	0.41	0.01
xml__rep2	7	3	R	0.39	–

Table A.3: Benchmarks arising from HMTT verification (2/2).

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

Table A.4: Benchmarks arising from exact flow analysis.

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

Table A.5: Benchmarks associated with HorSat(T).

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

Table A.6: Benchmarks arising from PMRS verification.

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

Table A.7: Benchmarks associated with TRecS.

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

Table A.8: Benchmarks associated with GTRecS at order 2.

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

Table A.9: Benchmarks associated with GTRecS at order 3.

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

Table A.10: Benchmarks associated with GTRecS at order 4.

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

Table A.11: Benchmarks associated with GTRecS at order 5.

Bibliography

Samson Abramsky. Abstract interpretation, logical relations, and kan extensions. *Journal of Logic and Computation*, 1(1):5–40, 1990.

Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51 (1–2):1–77, 1991.

Samson Abramsky and Chris Hankin, editors. *Abstract interpretation of declarative languages*. Halsted Press, 1987.

Kevin Backhouse and Roland Carl Backhouse. Safety of abstract interpretations for free, via logical relations and galois connections. *Science of Computer Programming*, 51(1–2): 153–196, 2004.

Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Symposium on Model Checking of Software, SPIN'00*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Symposium on the Principles of Programming Languages, POPL'02*, pages 1–3. ACM, 2002.

Henk Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North Holland, revised edition, 1984.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4): 931–940, 1983.

Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types.* Perspectives in Logic. Cambridge University Press, 2013.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency*

Theory, CONCUR'97, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

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

Christopher H. Broadbent and C.-H. Luke Ong. On global model checking trees generated by higher-order recursion schemes. In *Foundations of Software Science and Computational Structures, FOSSACS'09*, volume 5504 of *Lecture Notes in Computer Science*, pages 107–121. Springer Berlin Heidelberg, 2009.

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

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

Geoffrey L. Burn. A logical framework for program analysis. In *Glagow Workshop on Functional Programming*, Workshops in Computing, pages 30–42. Springer, 1992.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, CAV'00*, pages 154–169. Springer-Verlag, 2000.

Mario Coppo and Mariangiola Dezani. A new type-assignment for lambda terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19:139–156, 1978.

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

Mario Coppo and Alberto Ferrari. Type inference, abstract interpretation and strictness analysis. *Theoretical Computer Science*, 121(1–2):113–143, 1993.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Furio Honsell, and Giuseppe Longo. Extended type structures and filter lambda models. In *Logic Colloquium '82*, volume 112 of *Studies in Logic and the Foundations of Mathematics*, pages 241–262. Elsevier, 1984.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Maddalena Zacchi. Type theories, normal forms, and d-lambda-models. *Information and Computation*, 72(2):85–116, 1987.

Bruno Courcelle. Recursive applicative program schemes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 459–492. 1990.

Bruno Courcelle and Maurice Nivat. The algebraic semantics of recursive program schemes. In *Mathematical Foundations of Computer Science, MFCS'78*, volume 64 of *Lecture Notes in Computer Science*, pages 16–30, 1978.

Patrick Cousot. Types as abstract interpretations (invited paper). In *Symposium on Principles of Programming Languages, POPL'97*, pages 316–331. ACM, 1997.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages, POPL'77*, pages 238–252. ACM, 1977.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages, POPL'79*, pages 269–282. ACM, 1979.

Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *International Conference on Computer Languages, ICCL'94*, pages 95–112. IEEE Computer Society Press, 1994.

Werner Damm. Higher type program schemes and their tree languages. In *Theoretical Computer Science*, volume 48 of *Lecture Notes in Computer Science*, pages 51–72. Springer, 1977.

Werner Damm. The IO- and OI-Hierarchies. *Theoretical Computer Science*, 20:95–207, 1982.

Werner Damm, Elfriede Fehr, and Klaus Indermark. Higher-type recursion and self-application as control structures. In *Formal Descriptions of Programming Concepts, FDPC'78*, pages 187–461. North-Holland, 1978.

Brian A. Davey and Hilary A. Priestley. *Lattices and Order.* Cambridge University Press, second edition, 2002.

Jolie G. de Miranda. *Structures Generated by Higher-Order Grammars and the Safety Constraint.* PhD thesis, University of Oxford, 2006.

Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

Irène Guessarian. *Algebraic Semantics*, volume 99 of *Lecture Notes in Computer Science.* Springer, 1981.

Axel Haddad. Io vs oi in higher-order recursion schemes. In *Fixed Points in Computer Science, FICS'12*, volume 77 of *EPTCS*, pages 23–30, 2012.

Chris Hankin and Daniel Le Métayer. A type-based framework for program analysis. In *Static Analysis, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 1994.

J.Roger Hindley. Types with intersection: an introduction. *Formal Aspects of Computing*, 4(5):470–486, 1992.

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

Thomas P. Jensen. Strictness analysis in logical form. In *Functional Programming Languages and Computer Architecture, FPCA'91*, volume 523 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 1991.

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

Neil D. Jones and Nils Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1–3):120–136, 2007.

Marcin Jurdziński. Small progress measures for solving parity games. In *Symposium on Theoretical Aspects of Computer Science, STACS'00*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2000.

Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. Higher-order pushdown trees are easy. In *Foundations of Software Science and Computational Structures, FOSSACS'02*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002.

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

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

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

Naoki Kobayashi. Model checking higher-order programs. *Journal of the ACM*, 60(3): 20:1–20:62, 2013.

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

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

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

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

Robert P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 12: 38–43, 1976.

E. Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *International Conference on Software Engineering, ICSE '03*, pages 564–569. IEEE Computer Society, 2003.

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

Christian Mossin. Exact flow analysis. *Mathematical Structures in Computer Science*, 13 (1):125–156, 2003.

David E. Muller and Paul E. Schupp. Alternating automata on infinite trees. *Theoretical Compututer Science*, 54(2–3):267–276, 1987.

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

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

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

Pawel Parys. On the significance of the collapse operation. In *Logic in Computer Science, LICS'12*, pages 521–530. IEEE, 2012.

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

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. An abstraction refinement approach to higher-order model checking. In *Principles of Programming Languages, POPL'14*. ACM, 2014. To appear.

RTI. The economic impact of inadequate infrastructure for software testing. Technical Report (Planning) 02–3, National Institute for Standards and Technology, 2002.

Patrick Sallé. Une extension de la theorie des types en lambda-calcul. In *International Colloquium on Automata, Logic and Programming, ICALP'78*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410, 1978.

Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. In *International Colloquium on Automata, Languages and Programming, ICALP'11*, volume 6756 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 2011.

Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a scalable software model checker for higher-order programs. In *Partial Evaluation and Semantic-Based Program Manipulation, PEPM'13*, pages 53–62. ACM, 2013.

Yoshihiro Tobita, Takeshi Tsukada, and Naoki Kobayashi. Exact flow analysis by higher-order model checking. In *Functional and Logic Programming, FLOPS'12*, volume 7294 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2012.

Takeshi Tsukada and Naoki Kobayashi. Untyped recursion schemes and infinite intersection types. In *Foundations of Software Science and Computational Structures, FOSSACS'10*, volume 6014 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2010.

Hiroshi Unno, Naoshi Tabuchi, and Naoki Kobayashi. Verification of tree-processing programs via higher-order model checking. In *Asian Symposium on Programming Languages and Systems, APLAS'10*, volume 6461 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

US National Oceanic and Atmospheric Administration. Billion dollar weather/climate disasters. http://www.ncdc.noaa.gov/billions/events. Last accessed 23/07/2013.

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

Index of Definitions

(co-)trivial automaton, 24
 acceptance, 24
 alternating, 24
 complement of, 26
 deterministic, 25
 dual of, 26
 language of, 24

abstract configuration graph, *see* configuration graph
abstract interpretation, 122
 complete, 122
 exact, 122
 sound, 122
abstract property, 120
acceptance type
 see type, 65

concrete property, 119
concretisation function, 121
configuration, 72
 graph, *see* configuration graph
 prefix, 72
configuration graph, 72
 acceptance type extraction, 78
 accepting leaf, 74
 accepting region, 77
 rejecting leaf, 74
 rejecting region, 75
 rejection type extraction, 76
context
 sequence, 79
 typability, 72
continuous, 110
 function space, 110

domain, 110
 of trees, 19

fold, 16
function symbol, 9

higher-order recursion scheme, *see* recursion scheme

intersection type, *see* type

kind, 10
 argument, 71
 arity of, 10
 environment, 10
  refinement of, 49
 ground, 10
 interpretation, 110
 of a constant, 11
 order of, 10
 refinement of, 49
 term judgement, 11
 type judgement, 49

logical relation
 A, 113
 $R_{\leqslant}$, 125
 $R_{\geqslant}$, 125

model checking problem
    alternating (co-)trivial automaton, 27
move function, 58

play function, 59
positive Boolean formula, 24
    satisfying assignment, 24
property checking problem, 117
property language, 117
    universal, 135

rank, 11
recursion scheme, 12
    interpretation, 111
    non-terminal, 12
    order of, 13
    reduction relation, *see* reduction
    terminal, 12
    value tree, 20
redex, 14
reduction
    delta, 15
    scheme, 14
    sequence, 14
    weak beta, 16
refinement type, 49
    application, 121
    interpretation, 116
    interpretation of recursion schemes, 124
rejecting region, *see* configuration graph
rejection type, *see* type

Scott closed set, 115
shrink function, 120
signature, 11
    with casting, 127
substitution, 11
subtype, *see* type

term, 10
    applicative, 11
    arity of, 11
    closed, 11
    interpretation, 110
    order of, 11
    well-kinded, 11
    with casting, 127
term language, 117
    nondeterministic, 131
tree
    constructor, 19
    definedness order, 19
    domain, 19
    embedding, 20
    labelled, 18
    ranked, 18
type, 46
    acceptance, 65
    base, 46
    context, 71
    environment, 46
        (co-)consistency, 56
        multiplication, 46
        refinement, 49
        subsetting, 46
        subtype order, 47
        union, *see* multiplication
    equivalence, 47
    intersection, 46
    judgement, 50
    refinement, *see* refinement type
    rejection, 65
    strict, 46
    sub-, 47
    system, *see* type system
    theory, 47
type system, 50
    of abstract interpretation, 122
    of automaton, 60
    with casting, 127

variable, 9
    bound, 11
    free, 11
    typed, for abstraction, 71

Index of Notations

A^κ, 113
$D_1 \Rightarrow D_2$, 110
$F : \kappa$, 12
$P_1 \Rightarrow P_2$, 116
$S \models \phi$, 24
$T_1 \sqsubseteq T_2$, 19
$[\theta_1] \leqslant [\theta_2]$, 120
$\Delta, x : \kappa$, 11
$\Delta \vdash t : \kappa$, 11
$\Gamma, x : \sigma$, 50
$\Gamma :: \Delta$, 49
$\Gamma \ggcurly F : \sigma$, 56
$\Gamma \ggcurly F : \tau$, 56
$\Gamma \vdash t : \tau$, 50
$\Gamma_1 \leqslant \Gamma_2$, 47
$\Gamma_1 \mathrel{\underline{\Subset}} \Gamma_2$, 46
$\Gamma_1 \uplus \Gamma_2$, 46
$\Sigma^\perp$, 19
$\Sigma^\dagger$, 127
$\lambda x_1^{\kappa_1} \cdots x_n^{\kappa_n}.t$, 11
$\alpha_\kappa(P)$, 121
arity(κ), 10
$\bigsqcup A$, 119
$\bigsqcup E$, 110
$\bigwedge X$, 46
$\bigwedge \mathbb{T}(\Gamma)(t)$, 50
$\bigwedge_{i=1}^{n} \tau_i$, 46
$\perp$, 110
$\mathcal{A}^\perp$, 26
$\mathcal{A}_\perp$, 26
$\mathcal{Y}$, 71
$\mathsf{env}_\mathsf{R}(C)$, 77
$\eta_\kappa(d)$, 119
f, 24
FV, 11
$\gamma_\kappa(\theta)$, 121
$\ggcurly \Gamma$, 56
$\mathcal{P}_\mathsf{H}(D)$, 119
$\mathsf{exp}_n(x)$, 27
$\bigsqcup X$, 120
$\kappa_1 \to \kappa_2$, 10
kind(c), 11
$\mathbb{I}(\kappa)$, 120
$\mathbb{S}$, 10
$\mathcal{A}$, 61
$\mathcal{A}^c$, 26
$\mathcal{F}$, 9
$\mathcal{L}(\mathcal{A})$, 24
$\mathcal{M}(T)$, 76
$\mathcal{T}^\dagger$, 127
$\mathcal{V}$, 9
$R^\kappa_\geqslant$, 124
$R^\kappa_\leqslant$, 124
ACG(C), 72
A(y), 71
$\mathsf{B}^+(X)$, 24
K(y), 71
R(y), 71
Tree(s), 20
$\mathsf{Tree}_\beta(s)$, 20
cast, 127
$\mathsf{env}_\mathsf{A}(C)$, 78
extr(c, s), 78
RA(C), 77
RR(C), 75
$\bigsqcap X$, 120
$[\![\Delta \vdash t : \kappa]\!]$, 111
$[\![\mathcal{G}]\!]$, 111
$[\![\kappa]\!]$, 110
$[\![\theta :: \kappa]\!]$, 116
order(κ), 10
ϕ^c, 26
$\phi_1 \vee \phi_2$, 24
$\phi_1 \wedge \phi_2$, 24
rank(c), 11
Shrink(Γ), 120
$\sigma \to \tau$, 46
Step(Ξ), 111
t, 24
$\tau \in \sigma$, 46
$\tau_1 \wedge \tau_2$, 46
$s\,t$, 11
$\theta :: \kappa$, 49
$\theta_1 \equiv \theta_2$, 47
$\theta_1 \cdot \theta_2$, 121
$\theta_1 \leqslant \theta_2$, 47
$\top$, 46
var($\sigma_\mathsf{A}, \sigma_\mathsf{R}, \kappa$), 71
$d_1 \sqsubseteq d_2$, 110
o, 10
$s \rhd t$, 14
$s \rhd_\beta t$, 16
$s \rhd_\delta t$, 15
$s \rhd_w t$, 14
$s \rhd_{\delta\beta} t$, 16
s^+, 16
$t[\rho]$, 11
$t[s_1/x_1, ..., s_n/x_n]$, 11
$t^\perp$, 20
$F = \lambda x_1 \cdots x_n.t$, 12