Exact Intersection Type Abstractions for Safety Checking of Recursion Schemes

Steven J. Ramsay*
University of Warwick
s.ramsay@warwick.ac.uk

ABSTRACT

Higher-order recursion schemes are a class of higher-order programs built from a collection of first-order constants using higher-order functions and general recursion. Since the trees that they define have many decidable properties, they have become widely studied in the context of the verification through higher-order model checking.

We present a new semantic framework, in which recursion schemes can be extended by interpreted constants that are modelled concretely as a continuous function over domains and abstractly as an intersection type. When the intersection type of each constant is shown to be an exact abstract interpretation of the concrete semantics of that constant then intersection type assignment will characterise safety property checking for arbitrary terms. Since we focus on finite intersection type systems, decidability of the property checking problem is an immediate corollary.

Keywords

intersection types, higher-order model checking, abstract interpretation

1. INTRODUCTION

Higher-order recursion schemes are a class of higher-order programs built from a collection of first-order constants using higher-order functions and general recursion. When the first-order constants are all taken to be tree constructors, a given recursion scheme evaluates to a (possibly infinite) tree and various model checking problems can be defined which question whether or not this tree satisfies a given property. In a landmark result, Ong has shown that the class of trees so defined has a decidable monadic second-order theory [17].

Kobayashi [13] and later Kobayashi and Ong [14] have shown that such model checking problems are exactly captured by a family of type assignment problems in an intersection type system. The finite nature of this system ensures that type assignment is computable and hence it can be concluded that the original problems are decidable.

Let us consider a safety property defined by a tree automaton, as in [13]. Kobayashi obtains his correspondence by defining an intersection type system that is tailored to the property automaton in a very strong way. The types themselves are built from the states of the automaton, viewed as type atoms, and then basic typings for the tree constructors are axiomatised to the effect that state q, when viewed as a type, is inhabited by exactly those trees that are accepted by the automaton from state q. The quite standard rules for intersection, abstraction and application take care of extending type assignment to arbitrary terms in a way which is consistent with this view of the atoms, that is, a term has type q iff it evaluates to a tree accepted from q.

A natural question is whether there are other decision problems over recursion schemes that can be exactly captured by intersection type assignment, given a choice of type atoms and an axiomatisation of the scheme's constants.

On one hand, answering this question allows us to consider properties beyond languages of trees defined by automata. In particular, the language of an automaton can only specify properties of ground type objects, in this case trees, but in the syntax of intersection types it is entirely natural to write expressions that classify terms of higher-order. We wish to know when such expressions exactly characterise properties of objects of higher-type, i.e. properties of functions.

On the other hand, answering this question allows us to generalise beyond recursion schemes whose only constants are tree constructors. This has useful applications in verification, where the restriction has been found to be an obstacle to clean and efficient algorithms. In particular, the methods of [15, 18, 16] introduce extensions of tree constructing recursion schemes by limited forms of case analysis and nondeterministic branching. In each, the extension necessitated a change to the type system and hence a rewrite of the proof of correctness. We would like to minimise such duplication and smooth the way to problems considered over new recursion scheme variants.

We therefore present general conditions ensuring that safety property checking problems over recursion schemes can be completely characterised by intersection type assignment. Our conditions arise naturally as a requirement of *exact* abstract interpretation [7]. More specifically, we present a type based framework in the sense of those of Burn [4], Coppo and Ferrari [6], and Hankin and Le Métayer [11] which is parametrised by the constants over which recursion schemes

*The author carried out this work whilst a doctoral student at the University of Oxford.

are built and the basic atoms out of which intersection types are formed. When the chosen basic atoms are sufficiently expressive to capture the behaviour of the each chosen constant by an intersection type, the framework guarantees that type assignment *characterises* the associated safety checking problems for schemes built over those constants using higher-order functions and general recursion.

Let us give an example. Suppose we are interested in recursion schemes that do arithmetic and so consider basic constants for the integers, errors, addition, subtraction, multiplication and assertions. Say that we wish to understand how functions behave with respect to the parity of their arguments. Hence, we choose basic type atoms even and odd with the following interpretation: a given expression has type even (respectively odd) just if it is either non-terminating or evaluates to an even (respectively odd) integer. Then, given a function F and an intersection type τ, we can define queries by typing statements $F : \tau$. For example, the statement $F : ((\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even})) \to \mathsf{odd} \to \mathsf{odd})$ asks if F preserves oddness whenever it is given a function that flips parities. This is a question about the semantics of F phrased in the syntax of intersection types.

It follows from our framework that all such parity checking queries are decidable. More precisely, there is a finitary intersection type assignment problem that is a *sound and complete* characterisation of such problems. In other words, F is typed by τ iff the function defined by F satisfies the property. The reason is that the behaviour of the arithmetic constants with respect to parity can be completely captured by intersection types. For example, the addition and subtraction constants can be given the following type:

$$\begin{array}{ll} & (\mathsf{even} \to \mathsf{odd} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even} \to \mathsf{odd}) \\ \wedge & (\mathsf{odd} \to \mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{even} \to \mathsf{even}) \\ \wedge & (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{odd}) \wedge (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{even}) \\ \wedge & (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \wedge (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{even}) \end{array}$$

The first four conjuncts express the usual rules governing the parity of an addition or subtraction given the parity of its arguments. The latter four express the fact that when an argument is non-terminating (the only way a term may be said to be *both* odd *and* even) then anything may be said about the result[1]. Our framework ensures that, since this type exactly captures the behaviour of addition and subtraction with respect to parity (and the other constants can be typed similarly), it follows that type assignment exactly captures the behaviour with respect to parity of all terms built over these constants.

Related work.

The key requirement imposed by our framework is to show that an assignment of types to the constants of the scheme is an exact abstract interpretation of those constants. We take the term "exact" from Cousot [7], but such lossless abstractions have also been called "faithful" or "complete", see e.g. [10]. Let us only remark that our definition of exactness is a slight variation on the standard one, since we make explicit the role of the collecting semantics resulting in (for us) an easier to prove, though less general, condition.

The view of types as abstract interpretations has been investigated many times before, such as in Cousot's paper of the same title [7]. His work derives sound type systems of all kinds and relates them, but it is not interested in sound type systems that are also complete for a given class of program properties as we are here. Three separate sets of authors have come up with frameworks for program analysis based on intersection types during the early nineties which are very closely related to our own, namely [4], [6] and [11]. All are based on a denotational semantics of a lambda calculus and an interpretation of intersection types as certain subsets of domains. The central difference between our work and these is that our work is focused on guaranteeing both soundness and completeness of type assignment, 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. Where soundness is relatively straightforward to show, completeness throws up many issues (particularly at higher-types) and has precipitated our importing of the traditional Galois connection based approach to abstract interpretation, which is absent from these other works.

However, in certain other respects the work of these three authors goes further than ours. Burn [4] explores a more expressive type system which, in addition to arrows and conjunctions, includes type constructors for disjunctions and products. The work of Hankin and Le Métayer [11] follows that of Burn, on the one hand generalising the kinds of properties considered, but on the other restricting the language of types in order to give an efficient type assignment algorithm. Coppo and Ferrari [6] 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.

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 [1]. 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 but Jensen [12] showed, for the particular problem of strictness analysis, that such abstractions are isomorphic to filters of intersection types. Jensen's result about the equivalence of these two kinds of abstract domain sheds light on the relationship between our Lemma 8 and the paper of [2]. 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 $R^{\kappa}_{\leq}$ (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.

Independently, Tsukada and Ong [21] extended higher-order model checking to higher-types, but treat both ground type objects and higher-type objects as trees (the latter including binders). For us, higher-type objects are higher-order functions. Their work only considers recursion schemes that define trees using uninterpreted constants; by contrast, we develop a more general framework where constants can be given meaning and yet decidability retained through abstract interpretation. However, their work goes beyond ours in considering all properties definable by ω-regular winning conditions.

[1]Recall that we are interested in safety properties, which will not be violated by a term looping silently forever.

Structure of the paper.

The rest of the paper is as follows. In Section 2 we describe the preliminaries to the work and in Section 3 we specify property checking problems, with which we are ultimately concerned. In Section 4 we describe the central requirement of the framework, which is a statement of exact abstract interpretation. In Section 5 we prove that the ground-type restriction of our framework is correct and in Section 6 we give applications of how the framework can be used to simplify results in higher-order model checking. Finally, in Section 7 we investigate the conditions under which our framework is correct at all types. An extended version of the paper appears as Chapter 5 of the author's thesis [19].

2. PRELIMINARIES

In this section we aim to describe the setting for our framework, namely the syntax and semantics of recursion schemes and intersection types.

Higher-order recursion schemes.

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

Definition 1. The *kinds*[2] 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.

Definition 2. Let $(a, b, c \in) \Sigma$ be a set of atomic constants. The set of *terms* over Σ, is defined by the grammar:

$$s, t, u, v \quad ::= \quad x \mid F \mid c \mid s\, t \mid \lambda x^{\kappa}.\, t$$

We will feel free to omit the kind annotations on abstractions whenever they are unimportant or clear from the context. A term not containing any occurrence of a lambda abstraction is called *applicative* and a term not containing any occurrence of a free variable is called *closed*.

A *signature* declares a number of first-order term constants along with their arities.

Definition 3. 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 its *kind* defined by $\mathsf{kind}(c) = \underbrace{o \to \cdots \to o}_{\mathsf{rank}(c)\text{-times}} \to o$.

[2]We shall prefer *kind* to the more usual *simple type* so that *type* can be used to refer to intersection types exclusively.

$$\frac{\Delta \vdash s : \kappa_1 \to \kappa_2 \qquad \Delta \vdash t : \kappa_1}{\Delta \vdash s\, t : \kappa_2}$$

$$\frac{}{\Delta,\, x : \kappa \vdash x : \kappa} \qquad \frac{\mathsf{kind}(c) = \kappa}{\Delta \vdash c : \kappa} \qquad \frac{}{\Delta,\, F : \kappa \vdash F : \kappa}$$

$$\frac{\Delta,\, x : \kappa_1 \vdash t : \kappa_2}{\Delta \vdash \lambda x^{\kappa_1}.\, t : \kappa_1 \to \kappa_2}$$

Figure 1: Kind assignment for terms.

Definition 4. Fix a signature. A *kinded term judgement* is an expression of the form $\Delta \vdash t : \kappa$ which is provable in the system in Figure 1. A *well-kinded term* 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 constant $\Delta \vdash c : \kappa$ is identified with $\mathsf{rank}(c)$.

Higher-order recursion schemes [9] are a model of simply typed (kinded), higher-order functional programs with general recursion. They define the behaviour of a finite number of function symbols through equations between terms.

Definition 5. A *higher-order recursion scheme* $\mathcal{G}$ is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ consisting of: a signature Σ whose constants are called *terminals*, a finite set $\mathcal{N}$ of kinded *non-terminals* — a map from a finite subset of $\mathcal{F}$ to kinds in $\mathbb{S}$, a finite set of rules $\mathcal{R}$ and a distinguished non-terminal symbol of ground kind called the *start* symbol. We require $\mathcal{R}$ to be a function mapping each non-terminal $F : \kappa_1 \to \cdots \to \kappa_n \to o$ to its definition: a term of the form $\lambda x_1 \cdots x_n.\, t$ with t applicative. We will typically view this mapping as a finite set of equations and write $F = \lambda x_1 \cdots x_n.\, t$ whenever $\mathcal{R}(F) = \lambda x_1 \cdots x_n.\, t$.

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. We shall exclusively consider well-kinded higher-order recursion schemes in what follows.

Definition 6. Given a recursion scheme, the associated notion of *reduction* is the binary relation $\triangleright$ between applicative terms, defined as the compatible closure of the rule:

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

The reflexive, transitive closure of this relation is denoted by $\triangleright^*$ and finite reduction sequences are defined as usual.

For the purposes of higher-order model checking, an important feature of recursion schemes is that they can be used to define certain finite and infinite trees.

Definition 7. 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 $c_1 = \bot$.

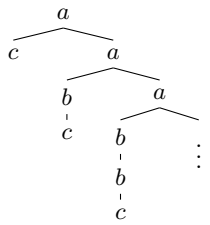
The order can be naturally extended to the set of all non-empty, $\Sigma^\perp$-ranked and labelled trees[3], $\Sigma\mathsf{Tree}^\infty$, by $T_1 \sqsubseteq T_2$ iff for all $w \in \mathsf{dom}(T_1)$, $T_1(w) \sqsubseteq T_2(w)$. Under this order, $\Sigma\mathsf{Tree}^\infty$ is a directed complete, partially ordered set with a least element (the least element is the tree consisting of only a root labelled by $\perp$).

Definition 8. Fix a recursion scheme. For any closed, ground kind term t, define the corresponding Σ-labelled tree $t^\perp$ as $\perp$ in case t has shape $F\ s_1\ \cdots\ s_n$ and as the tree with a labelling the root and subtree $s_i^\perp$ as i^{th} child in case t has shape $a\ s_1\ \cdots\ s_n$. Then given any closed, ground kind term s, the *tree generated by* s, $\mathsf{Tree}(s)$, is defined as $\mathsf{Tree}(s) = \bigsqcup\{t^\perp \mid s \triangleright^* t\}$

Example 1. The following is a scheme built over the constants a of arity 2, b of arity 1 and c of arity 0. The non-terminals $S : o$ and $F : o \to o$ are defined by the rules:

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

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.



Higher-order model checking is interested in verifying properties of the trees generated by recursion schemes. For this purpose, properties are defined by classes of tree automata.

Definition 9. A *trivial tree automaton*[4] $\mathcal{A}$ consists of a finite set of ranked constants (a, b, $c \in$) Σ, a finite set of states ($q \in$) Q and a partial mapping, the transition function, from pairs $(q,\ a) \in (Q \times \Sigma)$ to sets in $\mathcal{P}(Q^{\mathsf{rank}(a)})$.

Given a Σ-labelled tree T, a *q-rooted run tree on T* is a Q-labelled, unranked tree R satisfying the following conditions: (i) $\mathsf{dom}(R) = \mathsf{dom}(T)$ (ii) $R(\epsilon) = q$ (iii) For all $w \in \mathsf{dom}(R)$, $R(w{\cdot}1) \cdots R(w{\cdot}k) \in \delta(R(w), T(w))$ (where $k = \mathsf{rank}(T(w))$). The *language* of an automaton $\mathcal{A}$ at state q, $\mathcal{L}(\mathcal{A},\ q)$, is the set of Σ-ranked and labelled trees T for which there exists a q-rooted run-tree on T.

For the purposes of safety property checking, it is usual to assume that Σ does not include any symbol $\perp$ and define $\mathcal{A}^\perp = \langle \Sigma^\perp,\ Q,\ \delta \cup \{(q,\ \perp) \mapsto \{\epsilon\} \mid q \in Q\},\ q_0\rangle$. In this way, unproductive branches are silently accepted.

Intersection types.

Intersection types were first used to investigate the properties of lambda terms by Coppo and Dezani [5] and Sallé [20]. We follow the practice from higher-order model checking in stratifying the shape of types for convenience.

Definition 10. 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:

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

[3]As usual, we view a Σ-labelled tree as a map from a prefix closed subset of $\mathbb{N}^*$ to Σ.

[4]We omit initial states as they are not needed in this work.

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

Figure 2: Kind assignment for types.

in which $q \in Q$ and $n \geq 0$. 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 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.

We will typically write $\top$ for the empty intersection $\bigwedge \emptyset$, an intersection containing two elements infix as $\tau_1 \wedge \tau_2$ and an intersection of the singleton set containing τ simply as τ. Whenever convenient, we will identify $\sigma = \bigwedge_{i=1}^n \tau_i$ 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 $\bigwedge\{\tau \mid \tau \in X$ or there is some $\sigma \in X$ and $\tau \in \sigma\}$.

Definition 11. 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 $\leq$, is the least relation on types that validates the following rules:
(Q-Bas) if $(q_1,\ q_2) \in \Theta$ then $q_1 \leq q_2$
(Q-Arr) if $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$ then $\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$
(Q-Fun) if $\bigwedge_{i=1}^n \tau_i \leq \tau$ then $\bigwedge_{i=1}^n (\sigma \to \tau_i) \leq \sigma \to \tau$
(Q-Prj) for all $i \in [1..n]$, $\bigwedge_{j=1}^n \tau_j \leq \tau_i$
(Q-Glb) if, for all $i \in [1..n]$, $\sigma \leq \tau_i$, then $\sigma \leq \bigwedge_{j=1}^n \tau_j$
(Q-Trs) if $\theta_1 \leq \theta_2$ and $\theta_2 \leq \theta_3$ then $\theta_1 \leq \theta_3$
When two types θ_1 and θ_2 are mutual subtypes, i.e. $\theta_1 \leq \theta_2$ and $\theta_2 \leq \theta_1$, we say they are *subtype equivalent* and write $\theta_1 \equiv \theta_2$. We extend the ordering to type environments, writing $\Gamma_1 \leq \Gamma_2$ just if, for all $\xi \in \mathsf{dom}(\Gamma_2)$, $\Gamma_1(\xi) \leq \Gamma_2(\xi)$. When $\theta_1 \leq \theta_2$ we will often say that θ_1 is *stronger* than θ_2.

We shall not be interested in all the intersection types, but only those that have a shape that matches a kind. This restriction has the important consequence that the number of intersection types of a given kind is finite, and ultimately ensures the decidability of all that we shall consider.

Definition 12. 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 2. Kinding can be lifted to type environments by writing $\Gamma :: \Delta$ just if $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Delta)$ and, for all $F \in \mathsf{dom}(\Gamma)$, $\Gamma(F) :: \Delta(F)$.

Definition 13. An *intersection refinement type system*, is a triple $\langle \Sigma,\ Q,\ \mathsf{type}\rangle$ in which Σ is a signature, 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

$$\frac{\mathsf{type}(c) = \bigwedge_{i=1}^{n} \tau_i}{\Gamma \vdash c : \tau_i} \qquad \frac{\Gamma,\, x : \sigma \vdash t : \tau \qquad \sigma :: \kappa}{\Gamma \vdash \lambda x^{\kappa}.\, t : \sigma \to \tau}$$

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

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

Figure 3: Type assignment for terms.

assignment of types to the basic constants is *kind respecting*, in the sense that, for all constants c, $\mathsf{type}(c) :: \mathsf{kind}(c)$.

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. 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.

Only allowing subtyping to occur at applications is one way in which the system ensures that type assignment preserves kinding, in the following sense:

LEMMA 1. *In any intersection refinement type system:* $\Gamma \vdash t : \tau$ *and* $\Gamma :: \Delta$ *and* $\Delta \vdash t : \kappa$ *implies* $\tau :: \kappa$.

Kobayashi's type system [13] can be recovered by setting the type of each constant to be given by the property automaton and the associated subtype theory to be discrete.

Definition 14. Let $\mathcal{A} = \langle \Sigma, Q, \delta \rangle$. The type system induced by $\mathcal{A}$ is the system $\langle \Sigma, Q, \mathsf{type} \rangle$ in which:

$$\mathsf{type}(c) = \bigwedge \{q_1 \to \cdots \to q_n \to q \mid q_1 \cdots q_n \in \delta(q, c)\}$$

for all $c \in \Sigma$, and $(q_1, q_2) \in \Theta$ iff $q_1 = q_2$.

The type system obtained in this way characterises acceptance by the tree automaton.

THEOREM 1 ([13]). *Fix a recursion scheme* $\mathcal{G}$ *and trivial automaton* $\mathcal{A}$ *over the same signature. Then the following are equivalent for all closed terms* t *of ground kind and states* $q \in Q$*: (i)* $\mathsf{Tree}(t) \in \mathcal{L}(\mathcal{A}^{\perp}, q)$ *and (ii) there exists some* $\mathcal{G}$*-consistent*[5] Γ *such that* $\Gamma :: \mathcal{N}$ *and* $\Gamma \vdash t : q$

Further, since type checking is decidable and because, for a given non-terminal environment, there are only finitely many intersection type environments that refine it, it follows that (i) can be decided via (ii).

Interpretation of recursion schemes.

One view of the tree generating schemes in higher-order model checking is that each constant c behaves as a *tree constructor* — a function that simply glues together the trees given to it as input with c at the root. This view allows us to see the tree generated by a scheme as one instance of an interpretation of the constants.

[5]A type environment Γ is said to be $\mathcal{G}$-consistent just if, for every F and τ such that $\tau \in \Gamma(F)$, $\Gamma \vdash \mathcal{R}(F) : \tau$.

Definition 15. For us a *domain*, D, is a directed-complete partially ordered set with a least element $\perp$. That is to say $\perp \in D$ 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.

Definition 16. 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:

$$\begin{aligned} [\![o]\!] &= D_o \\ [\![\kappa_1 \to \kappa_2]\!] &= [\![\kappa_1]\!] \Rightarrow [\![\kappa_2]\!] \end{aligned}$$

An interpretation for a signature Σ is an interpretation for the kinds D_o and a choice of continuous function $d_c \in [\![\mathsf{kind}(c)]\!]$ for each constant $c \in \Sigma$. Then the choice is extended to each term $\Delta \vdash t : \kappa$ by assigning a continuous function:

$$[\![\Delta \vdash t : \kappa]\!] \in (\Pi_{\xi \in \mathsf{dom}(\Delta)} [\![\Delta(\xi)]\!]) \Rightarrow [\![\kappa]\!]$$

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, as specified by the clauses in Figure 4.

Since recursion schemes give definitions to function symbols, the interpretation of a given recursion scheme determines a semantic environment.

Definition 17. Given an interpretation for the signature, we interpret $\mathcal{G}$ as a semantic environment $[\![\mathcal{G}]\!]$, an element of $\Pi_{F \in \mathsf{dom}(\mathcal{N})} [\![\mathcal{N}(F)]\!]$ 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 $\mathsf{Step}(\Xi)(F) = [\![\mathcal{N} \vdash \mathcal{R}(F) : \mathcal{N}(F)]\!](\Xi)$.

Within 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. A routine argument shows that any such interpretation is a sound model of reduction.

LEMMA 2 (MODEL OF REDUCTION). *Fix a term signature and its interpretation. In any scheme over the signature, for all closed terms* s *and* t, $s \triangleright t$ *implies* $[\![s]\!] = [\![t]\!]$.

We now show 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.

Example 2. Consider the recursion scheme $\mathcal{G}$ from Example 1. The terminal symbols are interpreted as tree constructors $\underline{a}$, $\underline{b}$ and $\underline{c}$ over the domain of finite and infinite

$$
\begin{array}{rcl}
[\![\Delta \vdash c : \kappa]\!](\Xi) & = & d_c \\
[\![\Delta \vdash x : \kappa]\!](\Xi) & = & \Xi(x) \\
[\![\Delta \vdash F : \kappa]\!](\Xi) & = & \Xi(F) \\
[\![\Delta \vdash t_0\, t_1 : \kappa_2]\!](\Xi) & = & [\![\Delta \vdash t_0 : \kappa_1 \to \kappa_2]\!](\Xi)([\![\Delta \vdash t_1 : \kappa_1]\!](\Xi)) \\
[\![\Delta \vdash \lambda x^{\kappa_1}.\, s : \kappa_2]\!](\Xi)(d) & = & [\![\Delta, x : \kappa_1 \vdash s : \kappa_2]\!](\Xi \cup \{x \mapsto d\})
\end{array}
$$

Figure 4: Interpretation of terms.

$\Sigma^\perp$-ranked and labelled trees. We define, for each terminal symbol f of arity n, the tree constructor $\underline{f}$ by:

$$
\begin{array}{rcl}
\underline{f}(T_1)\cdots(T_n)(\epsilon) & = & f \\
\underline{f}(T_1)\cdots(T_n)(i \cdot w) & = & T_i(w)
\end{array}
$$

The scheme $\mathcal{G}$ determines a tree $[\![S]\!]$ and a function on trees $[\![F]\!]$ which together solve the following equations:

$$
\begin{array}{rcl}
[\![S]\!] & = & [\![F]\!](\underline{b}) \\
[\![F]\!](d) & = & \underline{a}(d)([\![F]\!](\underline{b}(d)))
\end{array}
$$

This ensures that the interpretation $[\![S]\!]$ of S is exactly the infinite, $\Sigma^\perp$-ranked and labelled tree, $\mathsf{Tree}(S)$.

It has long been known that such an interpretation yields a characterisation of the trees generated by safe schemes [9]. It is entirely straightforward to show that this extends also to arbitrary ones.

THEOREM 2 (ADEQUACY). *Fix a recursion scheme and consider the interpretation of the signature by the tree constructor semantics of Example 2. For all closed applicative terms t of ground kind, $[\![t]\!] = \mathsf{Tree}(t)$.*

Interpretation of intersection 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 trivial automaton model checking and define our space of properties out of an observation about the important structural features of $\mathcal{L}(\mathcal{A}^\perp, q)$ for any given $\mathcal{A}$ and q, namely: (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 and (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. In the context of a general subset of any domain, we formalise the requirements as follows.

Definition 18. Given a domain D, the *Scott closure* of a subset $E \subseteq D$, denoted E^*, is the smallest set X containing E and closed under the following conditions: (CL1) $\perp \in X$, (CL2) if $e \in X$ and $d \sqsubseteq e$ then $d \in X$ and (CL3) if $Z \subseteq X$ is directed, then $\bigsqcup Z \in X$.

To see the relationship of (CL3) to (iii), note that if Z is a set of finite[6] approximations to some limit $\bigsqcup Z$, then this condition requires that if $\bigsqcup Z$ belongs to the complement, $D \setminus X$, then some element in Z belongs to $D \setminus X$ too.

For us then, properties are Scott closed sets. Intersection types are a syntactical representation of properties with an associated proof theory, the following describes the standard way to interpret such types as Scott closed sets whilst validating the theory.

Definition 19. 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 (with ordering inherited) by:

$$
\begin{array}{rcl}
[\![q :: o]\!] & = & P_q \\
[\![\sigma \to \tau :: \kappa_1 \to \kappa_2]\!] & = & [\![\sigma :: \kappa_1]\!] \Rightarrow [\![\tau :: \kappa_2]\!] \\
[\![\bigwedge_{i=1}^n \tau_i :: \kappa]\!] & = & \bigcap_{i=1}^n [\![\tau_i :: \kappa]\!]
\end{array}
$$

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 $\forall x \in P_1.\, f(x) \in P_2$.

Thus arrow types are interpreted as properties of functions and intersection types as intersections of properties.

3. PROPERTY CHECKING PROBLEMS

Our framework ultimately concerns a certain class of property checking problems, which are described using recursion schemes and intersection types. We now make this precise.

Definition 20. A *term language* for domain D is described by a signature Σ and its interpretation $\{d_c\}_{c \in \Sigma}$ in D.

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.

Definition 21. A *property language* for domain D is described by an intersection type theory Q and its interpretation $\{P_q\}_{q \in Q}$ in D. We say that a property language is *decidable*, just if the subtype theory Θ is decidable.

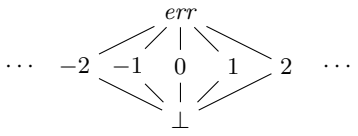
After fixing a term language and a property language one can formulate queries, which ask whether the interpretation of some intersection refinement type is a property that holds of the interpretation of some term.

Definition 22. 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 $\kappa = o$.

[6]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.

As defined here, the property checking problem is a generalisation of the trivial automaton model checking problem. To see this, fix a recursion scheme and an automaton. By interpreting the signature of the scheme by the tree-constructor semantics of Example 2 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 14 and interpreting each state q by the language $\mathcal{L}(\mathcal{A}^{\perp}, q)$. Then, by Theorem 2, it follows that $[\![S]\!] \in [\![q :: o]\!]$ iff $\mathsf{Tree}(S) \in \mathcal{L}(\mathcal{A}^{\perp}, q)$. 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.

Example 3. (Arithmetical recursion schemes parity checking problem) Fix a term language over a signature of arithmetic consisting of: a nullary constant `zero`, unary constants `succ`, `pred` and `assertEven`, and binary function symbols `add`, `sub` and `mul`. Interpret the signature over the domain of integers with non-termination and errors:



Interpret concretely `zero` by 0, `succ` by the strict successor function, `pred` by strict predecessor and interpret `assertEven` by the strict function that returns 0 when its argument is even or returns an error otherwise. Interpret the binary constants `add`, `sub` and `mul` by the strict natural number addition, subtraction and multiplication (respectively) whose behaviour on errors is to propagate them. Fix a property language over the atoms `even` and `odd`, interpreted by $\{0, \perp\}$, $\{n \in \mathbb{Z} \mid n \text{ is even}\} \cup \{\perp\}$ and $\{n \in \mathbb{Z} \mid n \text{ is odd}\} \cup \{\perp\}$ respectively. Then an instance of the parity checking 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:

$$[\![F]\!] \in [\![(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{odd} \to \mathsf{even}) \to \mathsf{odd} \to \mathsf{odd}]\!]$$

Example 4. (Arithmetical recursion schemes strictness checking problem) Fix a term language as before but now consider a property language described by the single atom $*$, with $[\![*]\!] = \{\perp\}$. Then an instance of the strictness checking 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:

$$[\![F]\!] \in [\![((* \to *) \to * \to *) \to \top \to * \to *]\!]$$

The decidability of both of these problems is a straightforward consequence of the main theorem in Section 7.

4. PROPERTIES AND ABSTRACTION

In this section we will place our work within the sphere of abstract interpretation. This is the correct language in which to state, in Definition 28, the proof obligation associated with our framework: exact abstraction of the constants.

Abstract interpretation is a theory of abstraction for the mathematical structures used in the formal description of programming languages. In the classical approach, pioneered by Cousot and Cousot [8], 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* $\langle A, \leq \rangle$, which are related by means of a Galois connection.

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, sometimes called the Hoare powerdomain.

Definition 23. 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 $\{e \in D \mid e \sqsubseteq d\}$.

LEMMA 3. *Fix an interpretation of simple types. Then $\mathcal{P}_{\mathsf{H}}([\![\kappa]\!])$ is a complete lattice under inclusion with:*

$$\bigsqcup A = (\bigcup A)^* \quad \textit{and} \quad \bigsqcap A = \bigcap A$$

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.

Definition 24. 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 $[\theta_1] \leq [\theta_2]$ iff $\theta_1 \leq \theta_2$.

LEMMA 4. *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. [3].* □

It follows that the join is necessarily determined as $\bigsqcup X = \bigcap\{[\theta] \mid \forall [\theta'] \in X.\, [\theta'] \leq [\theta]\}$. 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).

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.

Definition 25. 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 1, 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 $\mu\mathsf{Shrink} = \bigsqcup_{i\in\omega} \mathsf{Shrink}^i(\Gamma_{max})$.

We shall see in the following section that $\mu\mathsf{Shrink}$ is *precisely* the intersection type analogue of $\mu\mathsf{Step}$ for schemes.

Relationship between 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.

Definition 26. 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. We give the explicit definition since it will be used frequently.

Lemma 5. *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, 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 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 $\cdot$ and 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.

Definition 27. 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):

$$\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i) \cdot \sigma = \bigwedge\{\tau_i \mid i \in [1..n] \wedge \sigma \leq \sigma_i\}$$

Note that any intersection refinement type $\theta :: \kappa_1 \to \kappa_2$ can be viewed as an intersection of arrow types $\bigwedge_{i=1}^{n}(\sigma_i \to \tau_i)$.

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.

Definition 28. 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 and $d_1, \ldots, d_n \in [\![o]\!]$:

$$\alpha(\eta([\![c]\!](d_1)\cdots(d_n)) \leq \sigma_c \cdot \alpha(\eta(d_1)) \cdot \cdots \cdot \alpha(\eta(d_n))$$

We say that an abstract interpretation is *complete* whenever, for all constants c of rank n and $d_1, \ldots, d_n \in [\![o]\!]$:

$$\sigma_c \cdot \alpha(\eta(d_1)) \cdot \cdots \cdot \alpha(\eta(d_n)) \leq \alpha(\eta([\![c]\!](d_1)\cdots(d_n))$$

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.

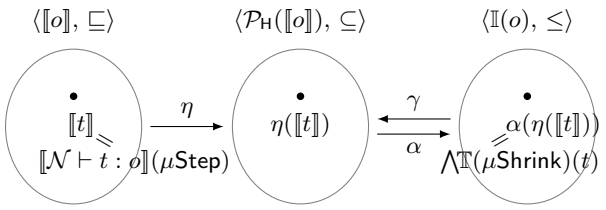


Figure 5: Exact abstraction at ground kind.

5. ABSTRACTION AT GROUND TYPES

The main result of our framework is that it guarantees that any property checking problem for which there is an exact abstract interpretation is completely characterised by intersection type assignment. In this section we prove the ground kind restriction of this result, Theorem 3. We first introduce the analogue of the type system induced by a trivial automaton described in Definition 14, which is the type system induced by an abstract interpretation.

Definition 29. 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 such systems but assume they 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.

Theorem 3 (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 $\alpha(\eta([\![t]\!])) = \bigwedge\mathbb{T}(\mu\mathsf{Shrink})(t)$ holds 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.

COROLLARY 1. *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:

$$\begin{array}{rcl} [\![t]\!] \in [\![\tau :: o]\!] & \equiv & [\![t]\!] \in \gamma_o(\tau) \\ & \equiv & \eta_o([\![t]\!]) \subseteq \gamma_o(\tau) \\ & \equiv & \alpha_o(\eta_o([\![t]\!])) \leq \tau \\ & \equiv & \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t) \leq \tau \end{array}$$

With the final equivalence following from Theorem 3. 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. □

Since subject reduction and subject expansion are often of independent interest, we point out that they are a simple consequence of having an exact abstract interpretation.

COROLLARY 2. *Fix a term language, a property language, an exact abstract interpretation and a recursion scheme. For all closed, ground kind terms s and t:*

$$\textit{if} \quad s \triangleright t \quad \textit{then} \quad \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(s) = \bigwedge \mathbb{T}(\mu\mathsf{Shrink})(t)$$

PROOF. Assume $s \triangleright t$. Since the kind interpretation models reduction (Lemma 2), also $[\![s]\!] = [\![t]\!]$ and hence $\alpha(\eta([\![s]\!])) = \alpha(\eta([\![t]\!]))$. The result follows from Theorem 3. □

We now prove the theorem. The main tool that we use is a pair of logical relations $R^\kappa_\leq$ and $R^\kappa_\geq$ between elements of the domain model and intersection types. The idea is that, if $d\ R^\kappa_\leq\ \theta$, then the collecting semantics of d is soundly approximated by θ and, if $d\ R^\kappa_\geq\ \theta$, then the collecting semantics of d is completely approximated by θ. However, this description is not precise: although we will see later that $d\ R^\kappa_\leq\ \theta$ is, indeed, equivalent to $\alpha(\eta(d)) \leq \theta$, we will also see that, beyond ground kind, $d\ R^\kappa_\geq\ \theta$ is not generally equivalent to $\alpha(\eta(d)) \geq \theta$. Nevertheless, we are here interested in ground kind only, so let us proceed with the definition.

Definition 30. Fix a term language and a property language. We construct two binary logical relations $R^\kappa_\leq$ and $R^\kappa_\geq$, indexed by kinds κ, between $[\![\kappa]\!]$ and $\mathbb{I}(\kappa)$. The definitions are inductive on κ, let $\diamond$ stand for either $\leq$ or $\geq$, 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 2.

LEMMA 6. *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 $\Xi\ R^\Delta_\diamond\ \Gamma$, we have $[\![\Delta \vdash t : \kappa]\!](\Xi)\ R^\kappa_\diamond\ \bigwedge(\mathbb{T}(\Gamma)(t))$.*

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 c, 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 $\lambda x.\, s$ then κ is of shape $\kappa_1 \to \kappa_2$, so let $d\ R^{\kappa_1}_\diamond\ \theta$. Then we observe that $[\![\Delta \vdash \lambda x.\, s : \kappa_1 \to \kappa_2]\!](d)$ is the same as $[\![\Delta, x : \kappa_1 \vdash s : o]\!](\Xi \cup \{x \mapsto d\})$ which, by the induction hypothesis, is related by $R^{\kappa_2}_\diamond$ to $\bigwedge \mathbb{T}(\Gamma \cup \{x : \theta\})(s)$. Finally, since $\bigwedge \mathbb{T}(\Gamma \cup \{x : \theta\})(s)$ is subtype equivalent to $\bigwedge \mathbb{T}(\Gamma)(\lambda x.\, s) \cdot \theta$, 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.

LEMMA 7. *Fix a term and a property language and an exact abstract interpretation, then: $\mu\mathsf{Step}\ R^\mathcal{N}_\diamond\ \mu\mathsf{Shrink}$ in any recursion scheme.*

PROOF. We first demonstrate, by induction on n, that for all $n \in \mathbb{N}$, $\mathsf{Step}^n(\bot)\ R^\mathcal{N}_\diamond\ \mathsf{Shrink}^n(\bot)$.

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 $d_i\ R^{R^{\kappa_i}}_\diamond\ \theta_i$ 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, $\mathsf{Shrink}^n(\bot)(F)$ is the same as $\mathsf{Shrink}(\mathsf{Shrink}^k(\bot))(F)$ which is just $\bigwedge \mathbb{T}(\mathsf{Shrink}^k(\bot))(\mathcal{R}(F))$. The result follows by the induction hypothesis and Lemma 6.

Now, to see the overall result, let $F : \kappa_1 \to \cdots \to \kappa_n \to o$ be in $\mathcal{N}$ and let $d_i\ R^{\kappa_i}_\diamond\ \theta_i$ $(i \in [1..n])$. We reason transitively:

$$\begin{array}{rl} & \alpha(\eta(\mu\mathsf{Step}(F)(d_1) \cdots (d_n))) \\ = & \alpha(\eta(\bigsqcup\{\mathsf{Step}^m(\bot) \mid m \in \mathbb{N}\}(F)(d_1) \cdots (d_n))) \\ = & \alpha(\eta(\bigsqcup\{\mathsf{Step}^m(\bot)(F)(d_1) \cdots (d_n) \mid m \in \mathbb{N}\})) \\ = & \bigsqcup\{\alpha(\eta(\mathsf{Step}^m(\bot)(F)(d_1) \cdots (d_n))) \mid m \in \mathbb{N}\} \\ \diamond & \bigsqcup\{\mathsf{Shrink}^m(\bot)(F) \cdot \theta_1 \cdots \theta_n \mid m \in \mathbb{N}\} \\ = & \mu\mathsf{Shrink}(F) \cdot \theta_1 \cdots \theta_n \end{array}$$

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.

6. APPLICATIONS IN VERIFICATION

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 and see how they can be characterised easily in our framework.

Trivial automaton model checking.

As a warm up we show that it is easy to recover the type based characterisation of 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, interpret kind o by the domain of $\Sigma^{\perp}$-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 interpret each state q as $\mathcal{L}(\mathcal{A}, q)$. We construct an abstract interpretation by assigning to constant a the intersection type:

$$\sigma_a = \bigwedge\{q_1 \to \cdots \to q_n \to q \,|\, q_1 \cdots q_n \in \delta(q, a)\}$$

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 $q_1 \cdots q_n \in \delta(q, c)$ and, for all $i \in [1..n]$, $\alpha(\eta(T_i)) \leq q_i$. Hence, for each such i, $T_i \in \gamma(q_i) = [\![q_i]\!] = \mathcal{L}(\mathcal{A}^{\perp}, q_i)$. Thus, $\underline{c}(T_1) \cdots (T_n) \in \mathcal{L}(\mathcal{A}^{\perp}, q)$ and so $\alpha(\eta(\underline{c}(T_1) \cdots (T_n))) \leq q$. 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 $q_1 \cdots q_n \in \delta(q, c)$ and, for each i, $T_i \in \mathcal{L}(\mathcal{A}^{\perp}, q_i)$. Hence, for each i, $\alpha(\eta(T_i)) \leq q_i$; thus $\sigma_c \cdot \alpha(\eta(T_1)) \cdots \alpha(\eta(T_n)) \leq q$.

By Theorem 3, it follows that this intersection type system gives a sound and complete characterisation of the trivial automaton model checking problem for recursion schemes.

Boolean recursion schemes.

Our second example is of *Boolean recursion schemes*[7] which are built over the signature containing `true` and `false` of zero arity and `if` of arity 3. We interpret o by the flat domain of Booleans $\{\mathit{true}, \mathit{false}, \perp\}$, the zero arity constants by *true* and *false* respectively and the if statement by setting $[\![\mathtt{if}]\!](b)(b_1)(b_2)$ equal to b_1 when $b = \mathit{true}$, b_2 when $b = \mathit{false}$ and $\perp$ otherwise. The property language is built over the base types Q consisting of `t` and `f` under the discrete subtype order and interpreted as $\{\mathit{true}, \perp\}$ and $\{\mathit{false}, \perp\}$ respectively. The abstract interpretation of the two constants `true` and `false` is given by `t` and `f` respectively and the `if` by:

$$\begin{array}{rcl} \sigma_{\mathtt{if}} & = & (\mathtt{t} \to \mathtt{t} \to \top \to \mathtt{t}) \wedge (\mathtt{t} \to \mathtt{f} \to \top \to \mathtt{f}) \\ & \wedge & (\mathtt{f} \to \top \to \mathtt{t} \to \mathtt{t}) \wedge (\mathtt{f} \to \top \to \mathtt{f} \to \mathtt{f}) \\ & \wedge & (\mathtt{t} \wedge \mathtt{f} \to \top \to \top \to \mathtt{t}) \wedge (\mathtt{t} \wedge \mathtt{f} \to \top \to \top \to \mathtt{f}) \end{array}$$

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:

$$\alpha(\eta([\![\mathtt{if}]\!](b)(b_1)(b_2))) = \sigma_{if} \cdot \alpha(\eta(b)) \cdot \alpha(\eta(b_1)) \cdot \alpha(\eta(b_2))$$

by distinguishing cases on b. If b is true then $[\![\mathtt{if}]\!](b)(b_1)(b_2)$ is just b_1. Since 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 $[\![\mathtt{if}]\!](b)(b_1)(b_2) = \perp$ and, since $\alpha \circ \eta$ is strict and $\perp_{\mathbb{I}(o)} = \mathtt{t} \wedge \mathtt{f}$, the result follows.

Hence, characterisation and decidability follow from Theorem 3. It should be clear that it is straightforward to move from if-statements and the Booleans to case-statements and finite enumerations e.g. RSFD [15].

[7] Of course, Booleans can be encoded as higher-order functions, but to do so is undesirable as the order of a scheme dominates the worst-case complexity of model checking.

Nondeterministic term languages.

In higher-order model checking literature, non-determinism is often simulated in recursion schemes by a binary choice constant, usually named `br`, for "branch".

Definition 31. 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 $\mathcal{P}_{\mathsf{H}}(\mathcal{P}_{\mathsf{H}}(D))$, which may be 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 typically wants 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.

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 `case` of arity $n+1$ and, as previously, the choice constant `br`. The term signature is intended as a language for describing non-deterministic computations over some finite enumeration of n values so, to be concrete, take $[\![o]\!] = \mathcal{P}_{\mathsf{H}}(\{1, \ldots, n, \perp\})$ with $[\![d_i]\!] = \eta(i)$ and $[\![\mathtt{case}]\!](D)(E_1) \cdots (E_n) = \bigsqcup\{E_j \,|\, j \in D \neq \perp\}$.

Let us now consider the abstract interpretation. We follow [16], 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 $\bigvee_{i=1}^{m} t_i \leq \bigvee_{j=1}^{n} t'_j$ 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 `case` by the type:

$$\bigwedge\{\bigvee X \to \sigma_1 \to \cdots \to \sigma_n \to q \,|\, q \in Q \wedge \sigma_i = \mathcal{F}_i(X, q)\}$$

where $\mathcal{F}_i(X, q) = q$ if $i \in X$ and $\top$ otherwise. Finally we interpret `br` by the type $\bigwedge\{q \to q \to q \,|\, q \in Q\}$.

All that remains is to show exactness. Consider a data item d_i, clearly $\alpha(\eta([\![d_i]\!])) = i = \sigma_{d_i}$. We next show that $\alpha(\eta(D_1 \sqcup D_2)) = \sigma_{\mathtt{br}} \cdot \alpha(\eta(D_1)) \cdot \alpha(\eta(D_2))$ in two applications of (Q-Glb). In the $\leq$ direction, let q be such that $\alpha(\eta(D_1)) \leq q$ and $\alpha(\eta(D_2)) \leq q$, then $\alpha(\eta(D_1)) \sqcup \alpha(\eta(D_2)) \leq q$ 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 $\alpha(\eta(D_1)) \leq q$ and $\alpha(\eta(D_2)) \leq q$ whence also $\sigma_{\mathtt{br}} \cdot \alpha(\eta(D_1)) \cdot \alpha(\eta(D_2)) \leq q$ as required. Next, note that:

$$\begin{array}{l} \alpha(\eta([\![\mathtt{case}]\!](D)(E_1) \cdots (E_n))) \\ \quad = \quad \bigsqcup\{\alpha(\eta(E_j)) \,|\, j \in D \neq \perp\} \\ \quad = \quad \bigwedge\{q \,|\, \forall j \in D.\, \alpha(\eta(E_j)) \leq q\} \end{array}$$

We aim to show that $\alpha(\eta([\![\mathsf{case}]\!](D)(E_1)\cdots(E_n)))$ is equal to $\sigma_{\mathtt{case}}\cdot\alpha(\eta(D))\cdot\alpha(\eta(E_1))\cdots\alpha(\eta(E_n))$. So to see that RHS $\leq$ LHS, we argue that RHS is a lower bound. So let q be such that, for all $j \in E$, $\alpha(\eta(E_j)) \leq q$ (*). Now $\alpha(\eta(E))$ is just $\bigvee E$ so, by the definition of σ_{case}, it remains to show only that $\alpha(\eta(E_j)) \leq q$ 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, $\bigsqcup\{\alpha(\eta(E_j)) \mid j \in D \neq \bot\} \leq q$. 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 $j \in D \neq \bot$, $\alpha(\eta(E_j)) \leq q$. Hence, $\bigsqcup\{\alpha(\eta(E_j)) \mid j \in D \neq \bot\} \leq q$.

To move from nondeterministic schemes with cases to weak pattern matching schemes [18] increases the complexity of the proof, since the corresponding type assignment system is more complicated. However, it is not any more conceptually difficult.

7. ABSTRACTION AT HIGHER-TYPES

In this section we lift Theorem 3 to higher kinds. An additional requirement is necessary, which is that the abstract property space at ground kind contains "no junk".

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.

Example 5. 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 choices exist. Now consider term $t = \lambda x.\, x$ of kind $o \to o$. It follows that $\bigwedge\mathbb{T}(\emptyset)(t) = q \to q$, 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 due to the fact:

LEMMA 8. *$\forall d \in [\![\kappa]\!]$, $\theta :: \kappa$: $\alpha_\kappa(\eta_\kappa(d)) \leq \theta$ iff $d\ R^\kappa_\leq\ \theta$.*

The problem of Example 5 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 (in this case 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.

Definition 32. 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, $(R^\kappa_\leq \cap R^\kappa_\geq)$ is a sub-graph of $\alpha_\kappa \circ \eta_\kappa$:

LEMMA 9. *Fix a term language and a universal property language and some exact abstract interpretation. Then, for all $\theta :: \kappa$, d $(R^\kappa_\leq \cap R^\kappa_\geq)$ θ implies $\alpha_\kappa(\eta_\kappa(d)) = \theta$.*

The main theorem is then a straightforward corollary.

THEOREM 4 (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, $\alpha_\kappa(\eta_\kappa([\![t]\!])) = \bigwedge\mathbb{T}(\mu\mathsf{Shrink})(t)$.*

PROOF. Since the abstraction is exact and by Lemmas 6 and 7, for all closed terms t of kind κ, $[\![t]\!]$ $(R^\kappa_\leq \cap R^\kappa_\geq)$ $\bigwedge\mathbb{T}(\mu(\mathsf{Shrink}))(t)$. The result follows from Lemma 9. □

Before proving the key lemma, let us see two applications of the theorem which show the characterisation and hence decidability of the problems described in Examples 3 and 4.

Example 6. (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:

$$\begin{array}{ll} \sigma_{\mathtt{succ}},\ \sigma_{\mathtt{pred}} & \sigma_{\mathtt{assertEven}} \\ \quad = \ (\mathsf{even} \to \mathsf{odd}) & \quad = \ (\mathsf{even} \to \mathsf{even}) \\ \quad \wedge \ (\mathsf{odd} \to \mathsf{even}) & \quad \wedge \ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \\ \quad \wedge \ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) & \quad \wedge \ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{even}) \\ \quad \wedge \ (\mathsf{odd} \wedge \mathsf{even} \to \mathsf{even}) & \end{array}$$

We interpret binary addition and subtraction by $\sigma_{\mathtt{add}}$, $\sigma_{\mathtt{sub}}$ which were given in the introduction and binary multiplication by $\sigma_{\mathtt{mul}}$:

$$\begin{array}{l} (\mathsf{even} \to \mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{odd} \to \mathsf{even} \to \mathsf{even}) \\ \quad \wedge \ (\mathsf{odd} \to \mathsf{odd} \to \mathsf{odd}) \wedge (\mathsf{even} \to \mathsf{even} \to \mathsf{even}) \\ \quad \wedge \ (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{odd}) \wedge (\mathsf{odd} \wedge \mathsf{even} \to \top \to \mathsf{even}) \\ \quad \wedge \ (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd}) \wedge (\top \to \mathsf{odd} \wedge \mathsf{even} \to \mathsf{even}) \end{array}$$

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.

Example 7. (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, both problems are decidable. We now come back to the proof of Lemma 9. 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 $\alpha_\kappa(\eta_\kappa(\delta_\kappa(\theta))) = \theta$.

Definition 33. 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.

$$\begin{array}{lcl} \delta_o(\theta) & = & c(\theta) \\ \delta_{\kappa_1 \to \kappa_2}(\theta)(x) & = & \delta_{\kappa_2}(\theta \cdot \alpha_{\kappa_1}(\eta_{\kappa_1}(x))) \end{array}$$

Before proving that this family has the desired property, let us note a fact about type application which will be useful.

LEMMA 10. *If* $\forall j \in [1..m].\, \sigma \cdot \sigma_j \leq \tau_j$, *it follows that* $\sigma \leq \bigwedge_{j=1}^{m}(\sigma_j \to \tau_j)$.

We next show that, for universal property languages, higher-kind abstractions that map under-approximations to under-approximations are under-approximations. At the same time, we show that $\delta_\kappa(\theta)$ is related by both $R^\kappa_\leq$ and $R^\kappa_\geq$ to θ. It follows from Lemma 8 that $\delta_\kappa(\theta)$ is therefore an element of the model whose best abstraction $\alpha(\eta(\delta_\kappa(\theta)))$ is exactly θ.

LEMMA 11. *Fix a term language and a universal property language. Then:*

(i) For all $\theta :: \kappa$: $d\ R^\kappa_\geq\ \theta$ *implies* $\theta \leq \alpha_\kappa(\eta_\kappa(d))$

(ii) For all $\theta :: \kappa$: $d_\kappa(\theta)\ (R^\kappa_\leq \cap R^\kappa_\geq)\ \theta$.

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 $d\ R^{\kappa_1 \to \kappa_2}_\geq\ \theta$ 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 $\theta \cdot \sigma_i \leq \tau_i$ and thus conclude by Lemma 10. It follows from the induction hypothesis that $\delta_{\kappa_1}(\sigma_i)\ R^{\kappa_1}_\geq\ \sigma_i$ and hence $d(\delta_{\kappa_1}(\sigma_i))\ R^{\kappa_2}_\geq\ \theta \cdot \sigma_i$. Consequently, and also from the induction hypothesis, $\theta \cdot \sigma_i \leq \alpha_{\kappa_1}(\eta_{\kappa_1}(d(\delta_{\kappa_1}(\sigma_i))))$, so it remains to show only that $\alpha_{\kappa_1}(\eta_{\kappa_1}(d(\delta_{\kappa_1}(\sigma_i)))) \leq \tau_i$; but this follows from (*) since the induction hypothesis gives $\delta_{\kappa_1}(\sigma_i)\ R^{\kappa_1}_\leq\ \sigma_i$ and Lemma 8 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 $\delta_\kappa(\theta)\ R^\kappa_\leq\ \theta$ let $e_i\ R^{\kappa_i}_\leq\ \theta_i$ for each $i \in [1..n]$. It follows from Lemma 8 that therefore $\alpha_{\kappa_i}(\eta_{\kappa_i}(e_i)) \leq \theta_i$ 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:

$$\alpha_o(\eta_o(\delta_\kappa(\theta)(e_1) \cdots (e_n)))) \leq \theta \cdot \alpha_{\kappa_1}(\eta_{\kappa_1}(e_1)) \cdots \alpha_{\kappa_n}(\eta_{\kappa_n}(e_n))$$

and hence $\alpha_o(\eta_o(\delta_\kappa(\theta)(e) \cdots (e_n)))) \leq \theta \cdot \theta_1 \cdots \theta_n$. The argument for showing $\delta^\kappa(\theta)\ R^\kappa_\geq\ \theta$ is analogous except we appeal to induction hypothesis clause (i) rather than Lemma 8. □

We can now prove Lemma 9 quite straightforwardly.

PROOF. Assume $d\ (R^\kappa_\leq \cap R^\kappa_\geq)\ \theta$, then it follows from Lemma 8 that $\alpha_\kappa(\eta_\kappa(d)) \leq \theta$ and from Lemma 11 that $\theta \leq \alpha_\kappa(\eta_\kappa(d))$, as required. □

8. REFERENCES

[1] S. Abramsky. Abstract interpretation, logical relations, and kan extensions. *Journal of Logic and Computation*, 1(1):5–40, 1990.

[2] K. Backhouse and R. C. Backhouse. Safety of abstract interpretations for free, via logical relations and galois connections. *Sci. Comp. Prog.*, 51(1–2):153–196, 2004.

[3] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. CUP, 2013.

[4] G. L. Burn. A logical framework for program analysis. In *GWFP'92*, Workshops in Computing, pages 30–42. Springer, 1992.

[5] M. Coppo and M. Dezani. A new type-assignment for lambda terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19:139–156, 1978.

[6] M. Coppo and A. Ferrari. Type inference, abstract interpretation and strictness analysis. *Theoretical Computer Science*, 121(1–2):113–143, 1993.

[7] P. Cousot. Types as abstract interpretations (invited paper). In *POPL'97*, pages 316–331. ACM, 1997.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

[9] W. Damm. The IO- and OI-Hierarchies. *Theoretical Computer Science*, 20:95–207, 1982.

[10] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[11] C. Hankin and D. L. Métayer. A type-based framework for program analysis. In *SAS'94*, volume 864 of *LNCS*, pages 380–394. Springer, 1994.

[12] T. P. Jensen. Strictness analysis in logical form. In *Functional Programming Languages and Computer Architecture, FPCA'91*, volume 523 of *LNCS*, pages 352–366. Springer, 1991.

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

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

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

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

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

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

[19] S. J. Ramsay. *Intersection Types and Higher-Order Model Checking*. PhD thesis, University of Oxford, 2014.

[20] P. Sallé. Une extension de la theorie des types en lambda-calcul. In *ICALP'78*, volume 62 of *LNCS*, pages 398–410, 1978.

[21] T. Tsukada and C.-H. L. Ong. Compositional higher-order model checking via ω-regular games over böhm trees. In *CSL-LICS'14*, 2014.