A Verified Cyclicity Checker

For Theories with Overloaded Constants

Arve Gengelbach
KTH Royal Institute of Technology, Stockholm, Sweden

Johannes Åman Pohjola
University of New South Wales, Sydney, Australia

Abstract

Non-terminating (dependencies of) definitions can lead to logical contradictions, for example when defining a boolean constant as its own negation. Some proof assistants thus detect and disallow non-terminating definitions. Termination is generally undecidable when constants may have different definitions at different type instances, which is called *(ad-hoc) overloading*. The Isabelle/HOL proof assistant supports overloading of constant definitions, but relies on an unclear foundation for this critical termination check. With this paper we aim to close this gap: we present a mechanised proof that, for restricted overloading, non-terminating definitions are of a detectable cyclic shape, and we describe a mechanised algorithm with its correctness proof. In addition we demonstrate this cyclicity checker on parts of the Isabelle/HOL main library. Furthermore, we introduce the first-ever formally verified kernel of a proof assistant for higher-order logic with overloaded definitions. All our results are formalised in the HOL4 theorem prover.

2012 ACM Subject Classification Software and its engineering → Correctness; Theory of computation → Program verification; Theory of computation → Higher order logic; Theory of computation → Graph algorithms analysis

Keywords and phrases cyclicity, non-termination, ad-hoc overloading, definitions, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.9

Related Version *Previous Version*: `Towards Correctly Checking for Cycles in Overloaded Definitions` [7]

Supplementary Material https://code.cakeml.org/tree/master/candle/overloading

Acknowledgements We are grateful to Andrei Popescu and Tjark Weber for technical discussion. We thank the anonymous reviewers for their constructive feedback. This work was begun while the first author was affiliated with Uppsala University, Sweden.

1 Introduction

For a consistent logical foundation, a theorem prover should only accept contradiction-free definitions. Although the logical foundations of many theorem provers are well studied, e. g. [21, 10, 13, 20, 5], still unverified implementations may allow proof of contradiction, e. g. by contradictory definitions [18, 14, 15].

Contradictory definitions can be avoided if each defined symbols and its dependants span a graph with only finite chains, i. e. if the so-called *dependency* graph is *terminating*. We showed this in earlier work for a variant of higher-order logic (HOL) [1]. For more expressive definitions, termination of the dependency graph is generally undecidable, which Obua [18] showed if a symbol (like a constant or a type) may be defined at different type instances, so-called *(ad-hoc) overloading*. Overloaded definitions enable recursion through types and their dependency graphs are generally infinite.

In-logic overloading of constant definitions distinguishes the Isabelle/HOL theorem prover from others, and permits Haskell-style type classes [26], which enable types to carry structure

with operations, e. g. a monoid type class with composition and a neutral element [24].

We illustrate non-terminating definitions by an example,[1] that enabled proof of contradiction in an earlier version of Isabelle/HOL. Assume a theory with the three polymorphic constants $c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}$, $d_{(\alpha\times\beta)\to\mathsf{Bool}}$ and $\mathsf{undefined}_\alpha$, and the following two definitions.

$$c\,(x_{\alpha\ \mathsf{list}}) \equiv d(\mathsf{undefined}_{\alpha\times\alpha}) \qquad\qquad d\,(x_{\alpha\times\mathsf{nat}}) \equiv \neg c([\mathsf{undefined}_\alpha])$$

Here, c with the argument $x_{\alpha\ \mathsf{list}}$ is defined in terms of d, and d with the argument $x_{\alpha\times\mathsf{nat}}$ is defined in terms of c. For the type instances of c at type $\mathsf{nat\ list} \to \mathsf{Bool}$ and of d at type $(\mathsf{nat} \times \mathsf{nat}) \to \mathsf{Bool}$, we obtain a non-terminating sequence and also a contradiction.

$$c\,([\mathsf{undefined}_{\mathsf{nat}}]) = d\,(\mathsf{undefined}_{\mathsf{nat}\times\mathsf{nat}}) = \neg c\,([\mathsf{undefined}_{\mathsf{nat}}])$$

Non-termination and the contradiction only surface after type variable instantiation. (To make this an overloaded definition, we could instead declare c of type $\alpha \to \mathsf{Bool}$ and in the definitions replace each name d by c.)

In order to facilitate the check of termination of overloaded definitions in Isabelle/HOL, Kunčar's work suggests that so-called *composable* non-terminating dependency graphs have a structure [14]. Detecting this structure in dependency relations that additionally are *orthogonal* is decidable. Kunčar defines a dependency relation orthogonal if it is functional, i. e. any symbol only depends on one other symbol. Despite these findings, we are unaware of any formalisation of the theorems.

In this paper, we present a complete formalisation of the theory that any non-terminating dependency relation contains cycles. We innovate to resolve a problem in Kunčar's argument for the main theorem that stems from an incorrect size comparison of a type prior to and after type instantiation. Further, we discover that Kunčar's restriction to orthogonal dependency relations is not satisfiable by dependencies stemming from definitions, and invent a cyclicity checker algorithm of our own. By formal proof our cyclicity checker can correctly calculate non-termination for composable dependency relations. We profit from the rich infrastructure around the CakeML language [23, 11] to synthesise a correct binary cyclicity checker. This checker shows that extracts of dependencies from Isabelle/HOL theories are composable and acyclic. If the checker detects acyclic dependencies of a theory of definitions, then by its correctness guarantees, we can discharge the assumption of non-terminating definitions, and obtain a consistent theory of definitions [1] with model-theoretic conservativity guaranties [8]. Altogether, we compose our verified cyclicity checker with the infrastructure from prior work [1, 8], to obtain a formally verified theorem prover kernel for HOL with overloading, that has the mentioned mechanised foundational properties.

All our definitions and theorems are formalised in the HOL4 theorem prover [19], and available online.[2]

The remainder of this paper is structured as follows. In Section 2 we discuss the syntax, to give an insight into the theory of non-terminating dependencies in Section 3. The full technical account of the proof are documented in a technical report [7]. Our algorithm is presented in Section 4 and illustrated with examples. We discuss potential future optimisations in Section 6 and conclude with Section 7.

---

[1] A variant of this example is attributed to Popescu by Kunčar [14].

[2] Our mechanisation of Sections 3 and 4 are part of the CakeML development repository at https://code.cakeml.org/tree/master/candle/overloading.

2 Syntax

In this section we define the syntax that we use throughout the paper. Type substitutions are one essential component that we define in Section 2.2. We define symbols in Section 2.3 to be types and typed constants (i. e. tuples of names and types). We extend all the notions to symbols (Section 2.3) and introduce dependency relations on symbols (Section 2.4). In Section 2.5 we define composability of dependency relations.

2.1 Notation

The definitions and theorems are mostly generated from our HOL4 formalisation, with theorems prefixed by $\vdash$. Constants in HOL4 are printed in sans-serif font and variables in *italic face*, i. e. SUC n for the successor of a natural number n. We freely move between lists and sets, and equate $X\ x$ with $x \in X$. The list functions last, front, $(++)$ and null denote the last element, all elements but the last, append and emptiness, respectively. A colon denotes the type of a term, as in $(\mathsf{last} : \alpha\ \mathsf{list} \Rightarrow \alpha)$.

The sum type is written $\alpha + \beta$, with disjoint branches $(\mathsf{INL} : \alpha \Rightarrow \alpha + \beta)$ and $(\mathsf{INR} : \beta \Rightarrow \alpha + \beta)$.

2.2 Types and Type Substitutions

Types are rank 1 polymorphic, and follow the grammar:

$$\mathsf{type} = \mathsf{Tyvar\ string} \mid \mathsf{Tyapp\ string\ (type\ list)}$$

The set of all type variables of a type ty is $\mathsf{FV}(ty)$. The size of a type is defined as

$$\mathsf{size}\ (\mathsf{Tyvar}\ m) \stackrel{\mathsf{def}}{=} 1 \qquad \mathsf{size}\ (\mathsf{Tyapp}\ m\ tys) \stackrel{\mathsf{def}}{=} 1 + |tys| + \mathsf{sum}\ (\mathsf{map\ size}\ tys).$$

We identify Tyvar «a», Tyvar «b», Tyvar «c» with α, β, γ respectively, and abbreviate common types like Tyapp «list» $[\alpha]$ with α list, and Tyapp «bool» [] with Bool, and function types Tyapp «fun» $[\alpha, \beta]$ with $\alpha \to \beta$. By the definition of size nullary types and type variables have the same size, e. g. size Bool = size $\alpha = 1$.

2.2.1 Type Substitutions

A *type substitution* ρ is a list of pairs of types such that $(y, \mathsf{Tyvar}\ x) \in \rho$ whenever $\rho\ (\mathsf{Tyvar}\ x) = y$. Duplicates w. r. t. the second component within the list ρ are ignored.

Type substitutions extend to types homomorphically, that is type substitutions instantiate type variables in a type. A type ty' is an *instance* of a more general type ty, written $ty \geq ty'$, if there exists a type substitution ρ such that $ty' = \rho\ ty$. We have implemented and verified an algorithm that computes whether or not a type is an instance of another type.

In addition, we implement and verify a first-order unification algorithm (from [3, § 2.3.2]) that produces an idempotent, most general unifier of two types (if one exists). A type unification algorithm can be used to calculate if two types have no common type instance, i. e. are *orthogonal* (written with infix #):

$$ty_1 \# ty_2 \stackrel{\mathsf{def}}{=} \neg\exists\, ty.\ ty_1 \geq ty \wedge ty_2 \geq ty$$

The types $\alpha \times \alpha$ and $\alpha \times \mathsf{nat}$ have the common instance $\mathsf{nat} \times \mathsf{nat}$, and are not orthogonal.

2.2.2 Variable Renamings

A special kind of type substitution η is a *renaming* of type variables, written $\mathsf{var\_renaming}\ \eta$. It is bijective and acts as the identity everywhere except on the subset of its domain $\mathsf{dom}\ \eta$, and only renames type variables.

$$\mathsf{var\_renaming}\ \eta \stackrel{\text{def}}{=} (\mathsf{img}\ \eta) = (\mathsf{dom}\ \eta) \wedge (\forall x.\ x \in \mathsf{img}\ \eta \Rightarrow \exists a.\ x = \mathsf{Tyvar}\ a) \wedge \mathsf{all\_distinct}\ (\mathsf{dom}\ \eta)$$

For instance, the type substitution $\eta = \{\alpha \mapsto \beta, \beta \mapsto \gamma, \gamma \mapsto \alpha\}$ is a renaming.

Two types x and y are *equivalent* if they differ by a renaming.

$$x \approx y \stackrel{\text{def}}{=} \exists \eta.\ \mathsf{var\_renaming}\ \eta \wedge x = \eta\ y$$

For instance, the types $\alpha\ \mathsf{list}$ and $\beta\ \mathsf{list}$ are equivalent by the renaming $\eta = \{\alpha \mapsto \beta, \beta \mapsto \alpha\}$. The relation $\approx$ is an equivalence (reflexive, symmetric and transitive).

2.3 Typed Constants and Symbols

Constants consist of a name and a type. *Symbols* are the sum type whose left leaves are types, $\mathsf{INL}\ (ty : \mathsf{type})$, and whose right leaves are typed constants, $\mathsf{INR}\ (\mathsf{Const}\ (c : \mathsf{string})\ (ty : \mathsf{type}))$. For a constant we sometimes write the type as an index, like c_{Bool} for a constant c of type Bool.

We lift all notions (like size, FV, type substitutions, $\leq$, $\#$, $\approx$) from types to constants in the obvious manner, like $\mathsf{size}(c_\tau) = \mathsf{size}(\tau)$ for a constant c_τ of type τ.

2.4 Dependency Relations

A dependency relation $\rightsquigarrow$ is a binary relation on symbols, i. e. types and typed constants. For any relation $\mathcal{R}$ we interchangeably use the infix notation $x\,\mathcal{R}\,y$, and $(x, y) \in \mathcal{R}$, even when $\mathcal{R}$ is internally represented as a list. Definitions imply dependencies, as is described elsewhere [15, 1]. For example, the two definitions

$$c\,(x_{\alpha\ \mathsf{list}}) \equiv d(\mathsf{undefined}_{\alpha\times\alpha}) \qquad \text{and} \qquad d\,(x_{\alpha\times\mathsf{nat}}) \equiv \neg c([\mathsf{undefined}_\alpha])$$

from Section 1 entail a dependency relation $\curvearrowright$ that contains the two elements

$$c_{\alpha\ \mathsf{list}\to\mathsf{Bool}} \curvearrowright d_{(\alpha\times\alpha)\to\mathsf{Bool}} \qquad \text{and} \qquad d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}\,.$$

We refer to this particular dependency relation as the bold-face green-coloured arrow $\curvearrowright$ throughout this Section 2.

We write $\mathcal{R}^+$ for transitive closure, and $\mathcal{R}^*$ for reflexive-transitive closure. With $\mathcal{R}^n$ we denote the n-times iterated composition relation $\underbrace{\mathcal{R} \cdots \mathcal{R}}_{n\text{-times}}$.

2.4.1 Monotone Relations

A relation is *monotone* if each type variable of the second argument is contained in the type variables of the first argument.

$$\mathsf{monotone}\ \mathcal{R} \stackrel{\text{def}}{=} \forall x\ y.\ (x,y) \in \mathcal{R} \Rightarrow \mathsf{FV}\ y \subseteq \mathsf{FV}\ x$$

For dependencies arising from definitions, monotonicity is a natural assumption as it means that each type variable of the right-hand side occurs in the defined symbol's type. For

example, attempting to define a constant e_{nat} as the cardinality of the universe of type α, i. e. $\mathsf{CARD}\,\mathcal{U}(:\alpha)$, entails a non-monotone dependency $e_{\mathsf{nat}} \rightsquigarrow \alpha$. This attempted definition is unsound as $\mathsf{CARD}\,\mathcal{U}(:\mathsf{bool}) = 2$ and there certainly are types of different cardinality. From this point onwards all dependency relations under consideration are monotone, unless otherwise stated.

2.4.2 Type-substitutive Closure

With overloading, non-termination may stem from recursion through the types of constants. Thus the analysis needs to consider type instances of dependencies. For a binary relation $\mathcal{R}$ on symbols, two symbols x and y are in the type-substitutive closure relation $x\,\mathcal{R}^{\downarrow}\,y$ if there exists a type substitution ρ such that $(\rho\ x)\,\mathcal{R}\,(\rho\ y)$. For example, $c_{\alpha\ \mathsf{list}\to\mathsf{Bool}} \curvearrowright d_{(\alpha\times\alpha)\to\mathsf{Bool}}$ implies $c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} d_{(\mathsf{nat}\times\mathsf{nat})\to\mathsf{Bool}}$. The type-substitutive closure of a dependency relation is infinite if the dependency relation contains a type variable.

2.4.3 Paths of Dependencies

In this section we define *solutions*, that we later use to witness elements in $\rightsquigarrow^{\downarrow *}$, *paths* as witnesses for elements in $\rightsquigarrow\rightsquigarrow^{\downarrow *}$ modulo renaming, and introduce paths of fixed length.

A *solution* to a list of pairs of symbols pq is a list of type substitutions ρ, with the constraint that applying the respective type substitution the i-th component $(\rho_i)\ (\mathsf{snd}\ (pq_i))$ equals the next $(\rho_{i+1})\ (\mathsf{fst}\ (pq_{i+1}))$.

$$\mathsf{sol\_seq}\ \rho\ pq \stackrel{\mathsf{def}}{=} \mathsf{wellformed}\ pq \wedge |\rho| = |pq| \wedge \forall\, i.\ i+1 < |\rho| \Rightarrow (\rho_i)\ (\mathsf{snd}\ (pq_i)) = (\rho_{i+1})\ (\mathsf{fst}\ (pq_{i+1}))$$

The wellformed predicate restricts the sequences to symbols (and could instead have been realised by the type system). For example, the sequence of length two of the pairs $(c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}, d_{(\alpha\times\alpha)\to\mathsf{Bool}})$ and $(d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}}, c_{\alpha\ \mathsf{list}\to\mathsf{Bool}})$ has a solution ρ with the compontents $\rho_0 = \rho_1 = \alpha \mapsto \mathsf{nat}$. This solution witnesses $c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} d_{(\mathsf{nat}\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright^{\downarrow} c_{\mathsf{nat\ list}\to\mathsf{Bool}}$, cf. Section 2.4.

We are mainly interested in solutions to lists of dependencies, i. e. where $pq \subseteq \rightsquigarrow$ holds.

A *path* through a dependency relation $\rightsquigarrow$ is a list of pairs of symbols pq in $\rightsquigarrow$, that have a solution ρ, such that ρ_0 is invertible on the type variables of $\mathsf{fst}\ (pq_0)$.

$$\mathsf{path} \rightsquigarrow \rho\ pq \stackrel{\mathsf{def}}{=} 0 < |\rho| \wedge pq \subseteq \rightsquigarrow \wedge \mathsf{invertible\_on}\ (\rho_0)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_0))) \wedge \mathsf{sol\_seq}\ \rho\ pq$$

Hence, modulo renaming, any path $\mathsf{path} \rightsquigarrow \rho\ pq$, corresponds to an element of $\rightsquigarrow\rightsquigarrow^{\downarrow *}$, namely

$$(\rho_0)\ (\mathsf{fst}\ (pq_0)) \ \rightsquigarrow\rightsquigarrow^{\downarrow *}\ (\mathsf{last}\ \rho)\ (\mathsf{snd}\ (\mathsf{last}\ pq)).$$

A cyclicity checker that calculates paths will fix the first element of a path and try to extend it. For the verification, we write $\mathsf{has\_path\_to} \rightsquigarrow n\ x\ y$ when there exists a path of length $n > 0$ from x to y, namely $x \rightsquigarrow (\rightsquigarrow^{\downarrow})^{n-1}\ y$.

$$\mathsf{has\_path\_to} \rightsquigarrow n\ x\ y \stackrel{\mathsf{def}}{=} \exists\, \rho\ pq.\ \mathsf{path} \rightsquigarrow \rho\ pq \wedge n = |pq| \wedge x = \mathsf{fst}\ (pq_0) \wedge y \approx (\mathsf{last}\ \rho)\ (\mathsf{snd}\ (\mathsf{last}\ pq))$$

We define $x = \mathsf{fst}\ (pq_0)$ instead of $x = (\rho_0)\ (\mathsf{fst}\ (pq_0))$, because due to monotonicity the sequence $({\rho_0}^{-1} \circ \rho_i)$ is a solution with $({\rho_0}^{-1} \circ \rho_0) = \mathsf{id}$ on $\mathsf{FV}\ (\mathsf{fst}\ (pq_0))$ and ${\rho_0}^{-1}$ only renames type variables of $(\mathsf{last}\ \rho)\ (\mathsf{snd}\ (\mathsf{last}\ pq))$.

For a fixed x and length n we can freely rename variables of y:

$$\vdash \mathsf{var\_renaming}\ \eta \Rightarrow (\mathsf{has\_path\_to} \rightsquigarrow n\ x\ y \iff \mathsf{has\_path\_to} \rightsquigarrow n\ x\ (\eta\ y))$$

2.4.4 Terminating and Cyclic Relations

A relation $\mathcal{R}$ is *terminating* if its converse relation has no infinite chains, i. e. is well-founded.

$$\mathsf{terminating}\ \mathcal{R} \stackrel{\text{def}}{=} \mathsf{WF}\ (\lambda\, x\ y.\ (y,x) \in \mathcal{R})$$

For a dependency relation $\rightsquigarrow$, termination of its type-substitutive transitive closure $\rightsquigarrow^{\downarrow+}$ is difficult to characterise, but for certain relations we prove "not cyclic implies terminating" in Section 3.6.

A path of length n is *cyclic*, written $\mathsf{cyclic\_len}\ \rightsquigarrow\ n$, if its last element is an instance of the first, namely $x\ \rightsquigarrow\ (\rightsquigarrow^{\downarrow})^{n-1}\ y$ with $x \geq y$, modulo renaming. A dependency relation is *cyclic*, written $\mathsf{cyclic\_dep}\ \rightsquigarrow$, if it has a cyclic path.

$$\mathsf{cyclic\_len}\ \rightsquigarrow\ n \stackrel{\text{def}}{=} \exists\, x\ y.\ \mathsf{has\_path\_to}\ \rightsquigarrow\ n\ x\ y \wedge x \geq y$$
$$\mathsf{cyclic\_dep}\ \rightsquigarrow \stackrel{\text{def}}{=} \exists\, n.\ \mathsf{cyclic\_len}\ \rightsquigarrow\ n$$

A cyclic relation is not terminating, because a cycle, i. e. $x\ \rightsquigarrow\rightsquigarrow^{\downarrow *}\ y$ with $y = \rho\ x$, entails non-terminating dependencies $x\ \rightsquigarrow\rightsquigarrow^{\downarrow *}\ (\rho^i\ x)$ for any $i > 0$.

The converse is not true. As an example, the closure $\curvearrowright^{\downarrow+}$ (cf. Section 2.4) is non-terminating, as witnessed by

$$c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} d_{(\mathsf{nat}\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright^{\downarrow} c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} \cdots,$$

however all paths in $\curvearrowright\curvearrowright^{\downarrow *}$ are of length one and not cyclic.

2.5 Composability

Composability is a central concept that makes checking for termination of dependency relations more feasible. We first give its definition and then exemplify the intuition.

A path $x\ \rightsquigarrow\ (\rightsquigarrow^{\downarrow})^n\ y$ from x to y is *composable* if for all $p \rightsquigarrow q$, either $y \leq p$ or $y \geq p$, or otherwise y and p are orthogonal $y \# p$. We quantify over x and y, and formally write that all paths of length n within $\rightsquigarrow$ are composable as $\mathsf{composable\_len}\ \rightsquigarrow\ n$:

$$\mathsf{composable\_len}\ \rightsquigarrow\ n \stackrel{\text{def}}{=}$$
$$\forall\, x\ y\ p\ q.\ \mathsf{has\_path\_to}\ \rightsquigarrow\ n\ x\ y \wedge (p,q) \in \rightsquigarrow \Rightarrow y \geq p \vee p \geq y \vee y \,\#\, p$$

A dependency relation $\rightsquigarrow$ is *composable*, denoted by $\mathsf{composable\_dep}\ \rightsquigarrow$, if all paths are. The relation $\curvearrowright$ (cf. Section 2.4) is not composable, because attempting to extend the dependency $c_{\alpha\ \mathsf{list}\to\mathsf{Bool}} \curvearrowright d_{(\alpha\times\alpha)\to\mathsf{Bool}}$ by $d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}$ contradicts composability. The constants $d_{(\alpha\times\alpha)\to\mathsf{Bool}}$ and $d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}}$ are not orthogonal, because their types are not. If d is instead defined at a more general type, like $d_{\alpha\times\alpha\to\mathsf{Bool}} \equiv \ldots$ then the dependency relation becomes composable, but remains non-terminating.

The implications of composability for a user are discussed in the Isabelle/Isar reference manual [25, § 5.9]. Composability requires all instances of overloaded constants to occur either at their most general type, or with all type variables instantiated.

3 Theory

Our main theoretic contribution is a formal proof that any monotone, composable, and finite dependency relation $\rightsquigarrow$ has a not terminating type-substitutive closure if, and only if the relation is cyclic.

$$\vdash \mathsf{monotone}\ \rightsquigarrow \wedge\ \mathsf{composable\_dep}\ \rightsquigarrow \wedge\ \mathsf{wellformed}\ \rightsquigarrow \wedge\ \mathsf{finite}\ (\rightsquigarrow) \Rightarrow$$
$$(\neg\mathsf{terminating}\ \rightsquigarrow^{\downarrow+} \iff \mathsf{cyclic\_dep}\ \rightsquigarrow)$$

The direction "cyclic implies not terminating" follows without composable_dep $\rightsquigarrow$, as sketched in Section 2.4.4. In the following we discuss the interesting converse direction, which in other words says, that for a finite, composable dependency relation $\rightsquigarrow$ an infinite sequence in $\rightsquigarrow^{\downarrow+}$ gives rise to a cycle in $\rightsquigarrow\rightsquigarrow^{\downarrow*}$ of particular shape. (Generally, the type-substitutive closure $\rightsquigarrow^{\downarrow}$ is already infinite if any type variables occur in $\rightsquigarrow$.)

Our proof arguments deviate from Kunčar's due to a false lemma [14, Lemma 3.1b], whose proof argues, that a type substitution that is not the identity should instantiate at least one type variable by a type. This argument misses the effect of renamings, and requires overall changes to the proof, although following Kunčar's general proof idea. We discuss how we circumvent this problem in Section 3.1.

The remaining section is structured as follows. We introduce some background concepts, like the effect of type substitution on the type variables FV and type size size in Section 3.1, and solutions that are *most general* in Section 3.2. Thereafter we continue the two main proof arguments. First, in Section 3.3, with composability every sequence in $\rightsquigarrow^{\downarrow+}$ implies that there is some corresponding sequence in $\rightsquigarrow\rightsquigarrow^{\downarrow*}$ [14, Lemma 5.11]. As a consequence (in Section 3.4) a solution can only be prolonged in two ways, either through a $\leq$-*extension* or a *strict* $\geq$-*extension*. Second, in Section 3.5, by composability the shape of any infinite sequence in $\rightsquigarrow\rightsquigarrow^{\downarrow*}$ can be narrowed down further. We combine these two main arguments to a proof sketch in Section 3.6.

For full technical details, this work is complemented by a technical report [7].

3.1 Type Instantiation, Type Size and Type Variables

In this section we describe why we deviate from Kunčar's original proof. One argument claims that non-identical type instantiation increases the size of a type [14, Lemma 3.1b]. A counterexample to this claim[3] is the instantiation of a function type $\alpha \to \beta$ with the type substitution $\rho = \alpha \mapsto \beta$, that unifies the two type variables $\rho(\alpha \to \beta) = \beta \to \beta$ and does not change the size: $\mathsf{size}(\alpha \to \beta) = \mathsf{size}(\beta \to \beta)$.

We circumnavigate this problem by observing change of size and number of type variables [7, Lemma 4]. A type substitution ρ may increase the size:

$$\vdash \mathsf{size}\ p \leq \mathsf{size}\ (\rho\ p)$$

However, if ρ is not invertible on $\mathsf{FV}(p)$ and the size is invariant $\mathsf{size}(p) = \mathsf{size}(\rho(p))$ then the number of type variables decreases strictly $|\mathsf{FV}(p)| > |\mathsf{FV}(\rho(p))|$.

That ρ has no inverse although the size is unchanged means that ρ only unifies type variables $\alpha, \beta \in \mathsf{FV}(p)$ (such that $\rho(\alpha) = \rho(\beta)$) and instantiates type variables to nullary types. Both nullary types and type variables have the same size, e. g. $\mathsf{size}\ \mathsf{Bool} = \mathsf{size}\ \alpha = 1$.

Assuming $q = \rho(p)$, i. e. q is an instance of p witnessed by ρ, then we can rephrase "ρ is not invertible on $\mathsf{FV}(p)$" as $\neg(q \leq p)$ which means that there is no type substitution that witnesses that $\rho(p)$ is an instance of p.

$$\vdash q \geq p \wedge \neg(p \geq q) \wedge \mathsf{size}\ q = \mathsf{size}\ p \Rightarrow |\mathsf{FV}\ p| < |\mathsf{FV}\ q|$$

Later in Section 3.4 we will call this a *strict* $\geq$-*extension*, as $q \geq p$ and $\neg(q \leq p)$.

[3] Note, that the stated problem also surfaces for Kunčar's slightly different definition of size.

3.2 Most General Solutions

Recall that solutions of dependencies witness elements in $\rightsquigarrow^{\downarrow *}$, as defined in Section 2.4.3. In this section we define *most general* solutions. For example, the sequence $(\alpha\ \mathsf{list}, \alpha), (\beta\ \mathsf{list}, \beta)$ has the solution $\rho_0 = (\alpha \mapsto \mathsf{Bool\ list}), \rho_1 = (\beta \mapsto \mathsf{Bool})$, which gives rise to the dependencies $\mathsf{Bool\ list\ list} \rightsquigarrow^{\downarrow} \mathsf{Bool\ list} \rightsquigarrow^{\downarrow} \mathsf{Bool}$. However, another more general solution would be $(\alpha \mapsto \beta\ \mathsf{list}), \mathsf{id}$.

A solution $(\rho'_i)_{i\leq n}$ for $(p_i, q_i)_{i\leq n}$ is a *most general solution*, written $\mathsf{mg\_sol\_seq}\ \rho'\ pq$, if any other solution $(\rho_i)_{i\leq n}$ is an instance of $(\rho'_i)_{i\leq n}$, i. e. there exist type substitutions $(\eta_i)_{i\leq n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ holds for $i \leq n$. (Note, that $\rho_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ entails $\rho_i(q_i) = (\eta_i \circ \rho_i)(q_i)$ due to monotonicity.)

A solution $(\rho_i)_{i\leq n}$ is most general if ρ_0 is invertible, e. g. when $\rho_0 = \mathsf{id}$.

$$\vdash \mathsf{monotone} \rightsquigarrow \wedge\ \mathsf{sol\_seq}\ \rho\ pq \wedge 0 < |pq| \wedge pq \subseteq \rightsquigarrow \wedge\ \mathsf{invertible\_on}\ (\rho_0)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_0))) \Rightarrow \mathsf{mg\_sol\_seq}\ \rho\ pq$$

Each path is a most general solution, and hence every path in $\rightsquigarrow\rightsquigarrow^{\downarrow *}$ is most general.

For a monotone dependency relation any two most general solutions $(\rho_i)_{i\leq n}$ and $(\rho'_i)_{i\leq n}$ are equivalent up to renaming, i. e. there exists a renaming η with $\rho'_i(p_i) = (\eta \circ \rho_i)(p_i)$ for $i \leq n$. Thus variable names in most general solutions can be freely renamed:

$$\vdash \mathsf{mg\_sol\_seq}\ \rho\ pq \wedge \mathsf{var\_renaming}\ \eta \Rightarrow \mathsf{mg\_sol\_seq}\ (\mathsf{map}\ (\lambda x.\ (\ \eta\ \circ\ x\ ))\ \rho)\ pq.$$

3.3 Restricting to Suffixes of Solutions

The first main implication of composability is, that any sequence in $\rightsquigarrow^{\downarrow +}$ has a corresponding suffix in $\rightsquigarrow\rightsquigarrow^{\downarrow *}$. More precisely, any solution has a most general solution with an invertible type substitution at some index k. After normalising the most general solution ρ', we could even assume that ρ'_k is the identity type substitution.

$$\vdash 0 < |pq| \wedge \mathsf{sol\_seq}\ \rho\ pq \wedge pq \subseteq \rightsquigarrow \wedge\ \mathsf{monotone} \rightsquigarrow \wedge\ \mathsf{composable\_dep} \rightsquigarrow \Rightarrow \exists \rho'\ k.\ \mathsf{mg\_sol\_seq}\ \rho'\ pq \wedge\ \mathsf{invertible\_on}\ (\rho'_k)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_k))) \wedge k < |pq|$$

This theorem has an important implication: for $pq = (p_i, q_i)_{i\leq n}$, it entails that the suffix $(\rho'_i)_{k\leq i\leq n}$ is a most general solution for $(p_i, q_i)_{k\leq i\leq n}$, because ρ'_k is invertible and w. l. o. g. a renaming, and hence composability applies to any possible extension of this suffix solution.

3.4 Prolonging a Solution by One Element

Searching for infinite sequences in $\rightsquigarrow^{\downarrow +}$ means at each breadth level n finding all extensions of a sequence of length n by one element in the $\rightsquigarrow^{\downarrow}$ relation. There are four possibilities for extending a solution $(\rho_i)_{i\leq n}$ of a sequence $(p_i, q_i)_{i\leq n}$ in the dependency relation, by one more element $p \rightsquigarrow q$.

Either the sequence cannot be prolonged by an instance of $p \rightsquigarrow q$ because $\rho_n(q_n)$ and p are orthogonal, i. e. $\rho_n(q_n)\#p$, or there are unifying type substitutions σ, σ' such that $\sigma(\rho_n(q_n)) = \sigma'(p)$. The following three cases arise.

1. σ is invertible and $\rho_n(q_n) = \sigma^{-1}(\sigma'(p))$. As we showed [7, Lemma 5], then $(\rho_i)_{i\leq n}, (\sigma^{-1} \circ \sigma')$ is a (most general) solution of $(p_i, q_i)_{i\leq n}, (p, q)$. This also includes the case that additionally σ' is invertible. We say $\leq$-*extension* for this case, as $\rho_n(q_n) \leq p$.
2. The substitution σ' is invertible, and σ is not invertible and $\sigma'^{-1}(\sigma(\rho_n(q_n))) = p$. Then by [7, Lemma 7] the type substitutions $(\sigma'^{-1} \circ \sigma \circ \rho_i)_{i\leq n}, \mathsf{id}$ form a (most general) solution

of $(p_i, q_i)_{i\leq n}, (p, q)$. We say *strict* $\geq$*-extension* for this case, as $\rho_n(q_n) \geq p$ and the type substitution $\sigma'^{-1} \circ \sigma$ that witnesses this instantiation is not invertible.

3. Both σ and σ' are not invertible. Then the type substitutions $(\sigma \circ \rho_i)_{i\leq n}, \sigma'$ form a solution of $(p_i, q_i)_{i\leq n}, (p, q)$, which is most general if σ and σ' are most general unifiers.

The following theorem summarises that, assuming composability, the case of Item 3 does not occur. If a solution for the sequence of dependency pairs pq can be extended, then a most general solution of pq, that is invertible at some index k, can be extended in only two manners, that together comprise Item 1 and Item 2.

$$\begin{aligned}\vdash\ &\mathsf{sol\_seq}\ \rho\ (pq +\!\!+ [(p,q)]) \wedge pq \subseteq \rightsquigarrow \wedge (p,q) \in \rightsquigarrow \wedge \mathsf{composable\_dep}\ \rightsquigarrow \wedge \\ &\mathsf{monotone}\ \rightsquigarrow \wedge \mathsf{mg\_sol\_seq}\ \rho'\ pq \wedge \mathsf{invertible\_on}\ (\rho'_k)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_k))) \wedge k < |\rho'| \Rightarrow \\ &(\mathsf{last}\ \rho')\ (\mathsf{snd}\ (\mathsf{last}\ pq)) \geq p \vee p \geq (\mathsf{last}\ \rho')\ (\mathsf{snd}\ (\mathsf{last}\ pq))\end{aligned}$$

By applying the theorem from Section 3.3 to $\mathsf{sol\_seq}\ (\mathsf{front}\ \rho)\ pq$ (any prefix of a solution is a solution), the assumptions $\mathsf{mg\_sol\_seq}\ \rho'\ pq$ and $\mathsf{invertible\_on}\ (\rho'_k)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_k)))$ can be discharged, as long as $|pq| > 0$.

3.5 Restriction to Only $\leq$-Extensions

Non-termination of $\rightsquigarrow^{\downarrow+}$ means that there is an infinite sequence $(p_i, q_i)_{i\in\mathbb{N}} \subseteq\rightsquigarrow$ with an infinite solution. Kunčar observes [14, Lemma 5.16] that the following holds. We correct the argument in [7, Theorem 11].

> For a composable and monotone dependency relation $\rightsquigarrow$, if the sequence $(p_i, q_i)_{i\in\mathbb{N}} \subseteq\rightsquigarrow$ has an infinite solution, then the following holds. There exists an index k, such that for each $k' > k$ the sequence $(p_i, q_i)_{i<k'}$ has a most general solution whose extension with $(p_{k'}, q_{k'})$ is a $\leq$-extension.

It suffices to show this claim for the sequence $(p_i, q_i)_{k\leq i<k'}$ instead (by an argument involving Section 3.3). Using composability, by contradiction there exists for each k a smallest index k' such that for any most general solution $(\rho_i)_{i<k'}$ of $(p_i, q_i)_{i<k'}$ the extension step by $(p_{k'}, q_{k'})$ is a strict $\geq$-extension. We briefly illustrate why these infinitely many strict $\geq$-extension steps lead to a contradiction, by observing the change of each solution step at index 0 using Section 2.2.

Assume $(\rho'_i)_{i\leq k'}$ is the most general solution of the longer sequence $(p_i, q_i)_{i\leq k'}$. For a $\leq$-extension, type sizes and number of type variables do not change: $\mathsf{size}(\rho_0(p_0)) = \mathsf{size}(\rho'_0(p_0))$ and $|\,\mathsf{FV}(\rho_0(p_0))| = |\,\mathsf{FV}(\rho'_0(p_0))|$. For a strict $\geq$-extension at index k', by monotonicity we can transfer the reasoning about type size and type variables from index k' to index 0: the type size may increase $\mathsf{size}(\rho_0(p_0)) \leq \mathsf{size}(\rho'_0(p_0))$, and if instead holds $\mathsf{size}(\rho_0(p_0)) = \mathsf{size}(\rho'_0(p_0))$ then the number of free variables decreases $|\,\mathsf{FV}(\rho_0(p_0))| > |\,\mathsf{FV}(\rho'_0(p_0))|$. During each of the infinitely many strict $\geq$-extensions the number of type variables at p_0 after instantiation can only decrease to zero. The contradiction is, that also the size of the type substitution applied to p_0 may not strictly increase infinitely. After finitely many strict $\geq$-extensions the type size of the respective most general solution at index 0 will be larger than the type size of the original infinite solution, that witnessed non-termination.

3.6 Non-termination Implies Cyclicity

By the arguments in Section 3.5, any infinite sequence in $\rightsquigarrow^{\downarrow+}$ entails a corresponding sequence in $\rightsquigarrow\rightsquigarrow^{\downarrow+}$ that contains $\leq$-extensions steps only.

As $\rightsquigarrow$ is finite, in an infinite sequence $(p_i, q_i)_{i\in\mathbb{N}} \subseteq \rightsquigarrow$ after some index k, a repetition must exist, i. e. for every i with $i > k$, (p_i, q_i) occurs again in $(p_j, q_j)_{j>i}$. We apply the theorem from Section 3.5 to the sequence $(p_i, q_i)_{i\geq k} \subseteq \rightsquigarrow$ with solution $(\rho_i)_{i\geq k}$. Hence at some index $k' > k$ we have a most general solution $(\rho'_i)_{i\geq k'}$ such that $\rho'_{k'}$ only renames the variables of $p_{k'}$ and only $\leq$-extension steps occur. We assume that $\rho'_{k'} = \mathsf{id}$, otherwise variables in the solution can be renamed (cf. Section 2.4.3). Let $m > k'$ be an index at which $(p_{k'}, q_{k'})$ occurs again, i. e. $(p_{k'}, q_{k'}) = (p_m, q_m)$, then $\rho'_{m-1}(q_{m-1}) \leq p_m = p_{k'}$ gives a cycle:

$$p_{k'} \rightsquigarrow q_{k'} = \rho'_{k'+1}(p_{k'+1}) \rightsquigarrow^{\downarrow} \ldots \rightsquigarrow^{\downarrow} \rho'_{m-1}(q_{m-1}) \leq p_m = p_{k'}$$

3.7 Comment on the Formalisation

From a proof-engineering perspective the formalisation of the arguments in our correction to the key lemma (cf. [7, Theorem 11] and [14, Lemma 5.16]) and also the implication "non-termination implies cyclicity" ([14, Lemma 5.17]) involves reasoning about infinite sequences and their infinite subsequences that satisfy certain properties, for which we used the Hilbert choice operator. One overall technical obstacle is the correct handling of type variables, for example some substitutions must be considered up to variable renaming and we need to verify a unification algorithm.

4 Algorithmically Checking for Cycles

In this section we describe the main idea and the correctness properties of a clocked breadth-first search that checks for composability and cycles in dependency relations. We invent a clocked algorithm of our own, because the restriction of *orthogonality* of dependency relations, that tremendously decreases the search space (as suggested by Kunčar [14]) does not apply to dependencies induced by theories of overloading definitions. We discuss the most important function, and conclude this section with an evaluation. We extract provably correct code, and check Isabelle/HOL theories for cycles in definitions.

4.1 Main Idea

As argued in Section 3.6, it suffices to search $\rightsquigarrow\rightsquigarrow^{\downarrow+}$ for cycles, in order to check non-termination of $\rightsquigarrow^{\downarrow+}$, given composability.

The basis for the cyclicity check is the following corollary. For a finite monotone dependency relation $\rightsquigarrow$, if at each depth composability and acyclicity hold, the type-substitutive closure of the dependency relation is terminating.

$$\vdash \mathsf{wellformed} \rightsquigarrow \wedge \mathsf{monotone} \rightsquigarrow \wedge \mathsf{finite}\ (\rightsquigarrow) \wedge (\forall\, n.\ \mathsf{composable\_len} \rightsquigarrow n) \wedge (\forall\, n.\ \neg\mathsf{cyclic\_len} \rightsquigarrow n) \Rightarrow \mathsf{terminating} \rightsquigarrow^{\downarrow+}$$

We recall from the definitions of $\mathsf{composable\_len} \rightsquigarrow n$ and $\mathsf{cyclic\_len} \rightsquigarrow n$, that these consider the paths $\rightsquigarrow (\rightsquigarrow^{\downarrow})^n$, which consist of only $\leq$-extensions, cf. Section 3.4.

The algorithm checks $\mathsf{composable\_len} \rightsquigarrow n$ and $\neg\mathsf{cyclic\_len} \rightsquigarrow n$ in a breadth-first manner for all $n \geq 1$ up to a given depth limit. (The case $n = 0$ is trivial.) The depth limit is required, because the search might not terminate if the dependencies $\rightsquigarrow\rightsquigarrow^{\downarrow+}$ are infinite. We discuss the depth limit for some practical examples in Section 4.5.

4.2 Central Components of the Algorithm

In this section we describe the algorithm dep_steps and discuss its core function composable_one in further detail.

To check a dependency relation $\rightsquigarrow$, the cyclicity checker algorithm dep_steps is called as dep_steps $\rightsquigarrow$ k $\rightsquigarrow$ with a maximal depth k. The function call dep_steps $\rightsquigarrow$ k $\mathcal{R}$ recursively extends each path that is stored in $\mathcal{R}$ by a dependency step $\rightsquigarrow$, and terminates either when $\mathcal{R}$ is empty, thus no extension is possible and all earlier steps were composable and acyclic. Or otherwise the recursion terminates if the depth counter k decreases to 0, and thus there are paths longer than k that the algorithm did not check for composability nor for cyclicity.

A call to dep_steps $\rightsquigarrow$ k $\rightsquigarrow$ results in one of the following outcomes, whose correctness we discuss in Section 4.3.

- Maybe_cyclic if the recursion depth k was too small, cf. Section 4.3.2.
- Acyclic _ if the relation is acyclic, cf. Section 4.3.3.
- Cyclic_step $(p,\_,p')$ if a cycle $p \rightsquigarrow\rightsquigarrow^{\downarrow +} p'$ with $p \geq p'$ exists, cf. Section 4.3.4,
- Non_comp_step (p,q,pq') if a non-composable path exists, i. e. $p \rightsquigarrow\rightsquigarrow^{\downarrow +} q$ with $pq' \in \rightsquigarrow$ and $\neg(q \geq \mathsf{fst}\ pq')$, $\neg(\mathsf{fst}\ pq' \geq q)$, $\neg(q \# \mathsf{fst}\ pq')$, cf. Section 4.3.4.

The function dep_steps $\rightsquigarrow$ k $\mathcal{R}$ folds another function dep_step $\rightsquigarrow$ at most k times over $\mathcal{R}$. For a relation $\mathcal{R}$ corresponding to $\rightsquigarrow$ $(\rightsquigarrow^{\downarrow})^n$ (for some breadth $n \geq 0$) the call to dep_step $\rightsquigarrow$ $\mathcal{R}$ computes all $\leq$-extensions for the current breadth n and checks the resulting relation $\rightsquigarrow$ $(\rightsquigarrow^{\downarrow})^{n+1}$ for cyclicity. Each of the possible extension steps of each of the paths $x\mathcal{R}y$ (i. e. $x \rightsquigarrow (\rightsquigarrow^{\downarrow})^n y$) by $p \rightsquigarrow q$ is computed according to Section 3.4 by composable_one y p, which may fail whenever $\mathcal{R}$ is not composable.

```
composable_one y p ≝
 case unify y p of
   None ⇒ Ignore
 | Some (s_y,s_p) ⇒
   let sp_inv = invertible_on s_p (FV p) ; sy_inv = invertible_on s_y (FV y) in
     if sp_inv ∧ ¬sy_inv then Ignore
     else if ¬sp_inv ∧ ¬sy_inv then Uncomposable
     else if ¬sp_inv ∧ sy_inv then Continue s_p
     else Continue []
```

Figure 1 Definition of the function composable_one, that is called by dep_steps.

The function composable_one y p, as defined in Figure 1, attempts to unify the two symbols y and p, and calculates if the resulting type substitutions are invertible. The possible return values are either Ignore, Uncomposable or Continue ρ for some type substitution ρ.

An Ignore return value signifies either orthogonality of y and p or a strict $\geq$-extension, i. e. $y \geq p$ and $\neg(y \leq p)$. If the unifying type substitutions are both not invertible, the relation is uncomposable. A return value Continue ρ means that $y \approx \rho\, p$, and the path can be continued as $x \rightsquigarrow (\rightsquigarrow^{\downarrow})^{n+1} (\rho\, q)$. Following the $\leq$-extension steps each calculated path is checked for cyclicity, i. e. if $x \geq (\rho\, q)$ holds.

4.3 Correctness

The major result about our algorithm dep_steps is its correctness. In Section 4.1 we motivated that for a monotone dependency relation that is acyclic and composable at all lengths, the type-substitutive transitive closure of the dependency relation is terminating. This theorem shows that we check composability and acyclicity using our algorithm dep_steps:

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ (\rightsquigarrow) \wedge \mathsf{finite}\ (\rightsquigarrow) \wedge \mathsf{dep\_steps}\ {\rightsquigarrow}\ (\mathsf{SUC}\ k)\ {\rightsquigarrow} = \mathsf{Acyclic}\ k' \Rightarrow \mathsf{terminating}\ {\rightsquigarrow}^{\downarrow +}$$

We outline some proof ideas and discuss further soundness and correctness properties.

The correctness proof is unsurprising but technically involved.

4.3.1 The Recursion Invariant of dep_steps

We establish a simple invariant $\mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ \mathcal{R}\ j\ \mathcal{R}'$ that captures that $\mathcal{R}$ reduces to $\mathcal{R}'$ in $(i-j)$ applications of $\mathsf{dep\_step}\ {\rightsquigarrow}$, that is, $(i-j)$ composable and non-cyclic steps in the dependency relation $\rightsquigarrow$. We further on regard the case of $\mathcal{R} = {\rightsquigarrow}$.

The invariant $\mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ {\rightsquigarrow}\ j\ \mathcal{R}'$ entails that $\mathcal{R}' = {\rightsquigarrow}\ ({\rightsquigarrow}^{\downarrow})^{i-j}$ modulo renaming.

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ (\rightsquigarrow) \wedge \mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ {\rightsquigarrow}\ j\ \mathcal{R}' \Rightarrow \forall x.\ \mathsf{wellformed}\ [x] \Rightarrow ((\exists y.\ (\mathsf{fst}\ x, y) \in \mathcal{R}' \wedge y \approx \mathsf{snd}\ x) \iff \mathsf{has\_path\_to}\ (\rightsquigarrow)\ (\mathsf{SUC}\ (i-j))\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x))$$

Modulo renaming, $\mathcal{R}'$ contains exactly the paths in $\rightsquigarrow$ of length $i-j+1$.

For a non-trivial, monotone dependency relation we characterise the invariant exactly.

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ (\rightsquigarrow) \wedge \neg\mathsf{null}\ {\rightsquigarrow} \wedge j \leq i \Rightarrow ((\exists \mathcal{R}'.\ \mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ {\rightsquigarrow}\ j\ \mathcal{R}') \iff (1 < i-j \Rightarrow \exists x\ y.\ \mathsf{has\_path\_to}\ (\rightsquigarrow)\ (i-j)\ x\ y) \wedge \forall k'.\ 0 < k' \wedge k' \leq i-j \Rightarrow \mathsf{composable\_len}\ (\rightsquigarrow)\ k' \wedge \neg\mathsf{cyclic\_len}\ (\rightsquigarrow)\ (\mathsf{SUC}\ k'))$$

The first conjunct within the conclusion $\exists x\ y.\ \mathsf{has\_path\_to}\ {\rightsquigarrow}\ (i-j)\ x\ y$ expresses non-emptiness of the previous search depth. If the search depth $i-j$ is strictly larger than one, then there exists a path in $\rightsquigarrow\ ({\rightsquigarrow}^{\downarrow})^{i-j-1}$. The second conjunct says that for each depth k' such that $0 < k' \leq i-j$, all paths in the relation $\rightsquigarrow\ ({\rightsquigarrow}^{\downarrow})^{k'-1}$ are composable and all paths in $\rightsquigarrow\ ({\rightsquigarrow}^{\downarrow})^{k'}$ are acyclic.

The lengths of the paths for composability and acyclicity differ by one, as within dep_step a composable relation is first extended, and then the resulting (one step longer) paths are checked for cyclicity.

4.3.2 Example: Deriving Correctness for Non-exhaustive Search

We establish when $\mathsf{dep\_steps}\ {\rightsquigarrow}\ k\ {\rightsquigarrow}$ outputs Maybe_cyclic: it can happen when $\rightsquigarrow\ ({\rightsquigarrow}^{\downarrow})^{k}$ is composable and acyclic, and contains a path that can be prolonged. In other words k-many iterative calls to $\mathsf{dep\_step}\ {\rightsquigarrow}$ yield a non-empty result, which in-turn shall mean that the supplied search depth was too small to cover all paths, and the calculated relation is non-empty and could contain the initial segment of a cycle. We establish this claim in terms of the invariant that we introduced in Section 4.3.1.

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \Rightarrow (\mathsf{dep\_steps}\ {\rightsquigarrow}\ k\ {\rightsquigarrow} = \mathsf{Maybe\_cyclic} \iff \exists \mathcal{R}'.\ \mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ k\ {\rightsquigarrow}\ 0\ \mathcal{R}' \wedge \neg\mathsf{null}\ \mathcal{R}')$$

Further, we establish that composability and acyclicity hold for the respective intermediate search steps up to the recursion level k, and secondly, that the relation $\mathcal{R}' = \rightsquigarrow (\rightsquigarrow^{\downarrow})^k$ is non-empty. The latter is witnessed by a path of length $k+1$.

$$\begin{aligned}
&\vdash \mathsf{wellformed}\ \rightsquigarrow \wedge\ \mathsf{monotone}\ (\rightsquigarrow) \Rightarrow \\
&\quad (\mathsf{dep_steps}\ \rightsquigarrow\ k\ \rightsquigarrow = \mathsf{Maybe_cyclic} \iff \\
&\qquad (\forall\, l.\ 0 < l \wedge l \leq k \Rightarrow \mathsf{composable_len}\ (\rightsquigarrow)\ l \wedge \neg\mathsf{cyclic_len}\ (\rightsquigarrow)\ (\mathsf{SUC}\ l)) \wedge \\
&\qquad \exists\, x\ y.\ \mathsf{has_path_to}\ (\rightsquigarrow)\ (\mathsf{SUC}\ k)\ x\ y)
\end{aligned}$$

4.3.3 Correctness for Acyclicity

For a sufficiently large k the function dep_steps detects that the non-trivial, monotone relation $\rightsquigarrow$ is acyclic, returning Acyclic k' for some integer $k' \leq k$. The value k' is such that all paths in the calculated relation $\rightsquigarrow\rightsquigarrow^{\downarrow+}$ are at most of length $k - k'$ and no path of length $k - k' + 1$ (or larger) exists. All paths of smaller length are composable and acyclic.

$$\begin{aligned}
&\vdash \mathsf{wellformed}\ \rightsquigarrow \wedge\ \mathsf{monotone}\ (\rightsquigarrow) \wedge \neg\mathsf{null}\ \rightsquigarrow \wedge\ 0 < k \Rightarrow \\
&\quad (\mathsf{dep_steps}\ \rightsquigarrow\ k\ \rightsquigarrow = \mathsf{Acyclic}\ k' \iff \\
&\qquad k' \leq k \wedge (\forall\, x\ y.\ \neg\mathsf{has_path_to}\ (\rightsquigarrow)\ (\mathsf{SUC}\ (k - k'))\ x\ y) \wedge \\
&\qquad (1 < k - k' \Rightarrow \exists\, x\ y.\ \mathsf{has_path_to}\ (\rightsquigarrow)\ (k - k')\ x\ y) \wedge \\
&\qquad \forall\, l.\ 0 < l \wedge l \leq k - k' \Rightarrow \mathsf{composable_len}\ (\rightsquigarrow)\ l \wedge \neg\mathsf{cyclic_len}\ (\rightsquigarrow)\ (\mathsf{SUC}\ l))
\end{aligned}$$

Acyclicity holds for paths of all lengths, including length one (which is not included in the equivalence). Such are cycles in $\rightsquigarrow$, and these entail cycles of length 2, and thus $\neg$cyclic_len $\rightsquigarrow$ 1 follows from $\neg$cyclic_len $\rightsquigarrow$ 2. Overall holds composable_dep $\rightsquigarrow$ and $\neg$cyclic_dep $\rightsquigarrow$.

4.3.4 Soundness for *Non-composable* and *Cyclic*

By the above two correctness results (in Sections 4.3.2 and 4.3.3) the algorithm dep_steps is sound and complete. For illustration, we state the soundness for when the algorithm witnesses a non-composable path or detects a cycle. For better readability we omit acyclicity and composability of previous search lengths from the conclusion.

$$\begin{aligned}
&\vdash \mathsf{wellformed}\ \rightsquigarrow \wedge\ \mathsf{monotone}\ (\rightsquigarrow) \wedge \\
&\quad \mathsf{dep_steps}\ \rightsquigarrow\ (\mathsf{SUC}\ k)\ \rightsquigarrow = \\
&\qquad \mathsf{Non_comp_step}\ (p,q,pq') \Rightarrow \\
&\qquad \exists\, n.\ n \leq k \wedge \\
&\qquad\quad \mathsf{has_path_to}\ (\rightsquigarrow)\ (\mathsf{SUC}\ n)\ p\ q \wedge \\
&\qquad\quad pq' \in \rightsquigarrow \wedge \neg(q \geq \mathsf{fst}\ pq') \wedge \\
&\qquad\quad \neg(\mathsf{fst}\ pq' \geq q) \wedge \neg(q\ \#\ \mathsf{fst}\ pq')
\end{aligned}$$

$$\begin{aligned}
&\vdash \mathsf{wellformed}\ \rightsquigarrow \wedge\ \mathsf{monotone}\ (\rightsquigarrow) \wedge \\
&\quad \mathsf{dep_steps}\ \rightsquigarrow\ (\mathsf{SUC}\ k)\ \rightsquigarrow = \\
&\qquad \mathsf{Cyclic_step}\ (p,q,p') \Rightarrow \\
&\qquad \exists\, n.\ n \leq k \wedge \\
&\qquad\quad \mathsf{has_path_to}\ (\rightsquigarrow)\ (\mathsf{SUC}\ (\mathsf{SUC}\ n))\ p\ p' \wedge \\
&\qquad\quad p \geq p'
\end{aligned}$$

When an uncomposable step is detected, then there exists a path $p \rightsquigarrow (\rightsquigarrow^{\downarrow})^n\ q$ that is not composable (cf. Section 2.5). As stated, when a cyclic step is detected, then there is a path in $\rightsquigarrow (\rightsquigarrow^{\downarrow})^{n+1}$ that ends in an instance of the starting symbol, and by definition cyclic_dep $\rightsquigarrow$ holds (cf. Section 2.4.4).

4.4 A Verified Theorem Prover Kernel

In earlier work [1] we implemented parts of a theorem prover that supports overloading of constant definitions, and is partially based on a verified implementation of HOL Light

(nicknamed Candle) [13]. A verified cyclicity checker is the missing puzzle piece of this verified theorem prover. Whenever an overloaded constant is added to a proof development the resulting theory needs to have a terminating dependency relation.

A theory of definitions *ctxt* has the relational dependencies dependency *ctxt* that are computed by dependency_compute *ctxt*. The following corollary (of the results from Section 3 and Section 4.3.3) allows to discharge termination of dependency *ctxt* by a call to the dep_steps algorithm on the dependencies dependency_compute *ctxt*.

$$\begin{aligned}&\vdash \mathsf{let} \rightsquigarrow\ =\ \mathsf{dependency\_compute}\ \mathit{ctxt}\ \mathsf{in}\\ &\quad \mathsf{dep\_steps} \rightsquigarrow (\mathsf{SUC}\ k) \rightsquigarrow = \mathsf{Acyclic}\ k' \wedge \mathsf{good\_constspec\_names}\ \mathit{ctxt} \Rightarrow\\ &\qquad \mathsf{terminating\ dependency}\ \mathit{ctxt}^{\downarrow +}\end{aligned}$$

The premise good_constspec_names *ctxt* states that all type variables on the right-hand side of a constant definition must occur on the left-hand side. This ensures that the dependency relation is monotonic. All HOL kernel implementations that we are aware of enforce this restriction, since without it, HOL is inconsistent (see Section 2.4.1).

To obtain a verified kernel, we integrate our cyclicity check into a shallowly embedded monadic HOL kernel derived from [13], and extract a correct-by-construction CakeML implementation using existing tools for proof-producing synthesis [11]. By the previous theorem, we can replace the termination assumption with a call to the (monadic) cyclicity checker and prove kernel soundness, i. e. a successful check entails a valid update of a theory.

To our knowledge, this yields the first verified theorem prover kernel that both supports overloading of constant definitions and has mechanised semantics [1] with a formal proof that any theory is consistent. The proof of consistency for theories relies on the consistency of Zermelo-Fraenkel set theory. From our joint work with Weber follow (formally verified) model-theoretic conservativity guarantees [8].

4.5 Checking Cyclicity of Isabelle/HOL Theories

In addition to verifying the theory and deriving a correct algorithm, we extracted the dependencies of theories from Isabelle/HOL and checked if their dependency graphs are acyclic and composable. We focus on the theory Main, that extends HOL with libraries for e. g. orderings, lattices, transitive closure, sets and natural numbers [16].

We composed an (unverified) dependency parser written in CakeML with the (verified) CakeML implementation of our cyclicity checker, and extracted an executable binary using the CakeML compiler. The translator ensures that the cyclicity checking function of the binary has the same correctness properties as the monadic variant, which in turn is equivalent to the HOL4 implementation from Section 4.2.

The results in Figure 2 show for each Isabelle/HOL theory the number of extracted dependencies, the result output of the checker, the runtime in seconds or hours on an Intel Core i7 processor, and the length of the longest checked path. All of these results state that the extracted dependencies contain no cycles and are composable. The reported maximal length of covered paths of 37 shows that for these realistic scenarios the maximum depth limit can be chosen small. Implementing a cyclicity checker that is optimised for performance was not the objective of this work, which shows in the runtimes. Checking the 45,738 dependencies of the complete Isabelle/HOL main library is not feasible, but we establish that the subset of dependencies from constants to constants is acyclic. This approach is unsound, but gives an idea of the algorithm's performance.

In the checked theories overloading is mainly occurring due to type classes, e. g. the `size_class` exhibits a constant `Nat.size_class.size` of type $\alpha \to$ `nat`, and the list type is

Theory	#Dependencies	Output	Runtime	Longest path
HOL	165	Acyclic	0.01s	7
Orderings	764	Acyclic	0.4s	13
Set	2657	Acyclic	13s	14
Fun	2773	Acyclic	13s	14
Transitive_Closure*	2195	Acyclic	8s	20
Transitive_Closure	7159	Acyclic	14h	35
Main*	12913	Acyclic	12h	37
Main	45738	-	-	-

Figure 2 Results of checking exported dependencies of Isabelle theories. The asterisk * denotes that the dependencies only include dependencies of constants on constants. The runtimes are from single runs.

a class instance that defines a `size` constant at the type α list $\rightarrow$ nat.

The runtimes motivate that a cyclicity checker should be checking dependencies incrementally, which we discuss further in Section 6, because incremental checking corresponds to the incremental nature of theory extension.

5 Related Work

Like other theorem provers that do not support overloading, the verified implementation of HOL Light into the CakeML framework [13], nicknamed Candle, achieves acyclic definitions by a simple syntactic check: Only already defined constants and types are allowed to occur in the definition of a new symbol.

In the Coq theorem prover a termination check corresponds to finding a type hierarchy acyclic. When type-checking a term, hierarchy (in)equality constraints are collected, whose conjunction needs to be satisfiable. As Sozeau and Tabareau argue [22], checking these conjunctions is decidable. Earlier a proof of contradiction seemed to be originating from a bug in this cyclicity checker [6]. The current Coq implementation [12] relies on an incremental cyclicity checking algorithm by Bender et al. [4], that combines forward- and backward-search and uses a non-decreasing integer level invariant for chains in the graph. Guéneau et al. [9] verified a similar algorithm in addition to some of the complexity properties. We discuss incremental extension to our algorithm in Section 6. In contrast to Sozeau and Tabareau, checking termination of interesting dependency relations in our context is not decidable. To add another difference, their rewriting system allows unfolding of a constant by its definition, whereas our cyclicity checker can also check dependencies from theories with more expressive definitions [2], e. g. implicit definitions.

The current algorithm that checks for cycles in Isabelle theories[4] is authored by Wenzel. It is unclear how the implementation relates to Kunčar's work [14], and our cyclicity checker. A later proof checker for Isabelle/HOL by Nipkow and Roßkopf [17] treats definitions as axioms, i. e. does not check for cycles.

With the intent to obtain a terminating and efficient cycle detection algorithm, Kunčar suggests in [14] that dependency relations should satisfy *orthogonality*. The orthogonality criterion implies that paths do not diverge, hence any two paths from a symbol pass through the same symbols if the paths are of same length [14, Theorem 6.2]. That means that

[4] https://isabelle.sketis.net/repos/isabelle/file/Isabelle2021/src/Pure/defs.ML

composability only needs to hold for paths that cannot be extend any further, so called *final* paths. Orthogonality is no suiting criterion for dependencies of definitions (cf. [15, 1]): any definition of a symbol may depend on more than one other symbol [7].

6 Future Work

With the suggested algorithm we already implement the interesting and correctness critical optimisation to avoid inversion of bijective renamings. For future work we identify three main areas of improvements.

First, we should investigate further restrictions of dependency relations that avoid checking composability of every path and second, the dependency relation that is generated from theories should be minimised. For example, dependencies introduced by some built-in symbols can be omitted from the graph.

Third, theory extension is incremental, and such should the check of cyclicity (and composability) be. Whenever a theory is extended by a definition, only a few dependencies are added by the new definition, which implies that big parts of the dependency graph remain unchanged. Together with Weber [8] we have identified those parts of the dependency graph that change by the introduction of a new definition. An incremental cyclicity check clearly profits from such an analysis.

As an example, assume a theory with terminating dependencies that contains the constant $\mathsf{length}_{\alpha\ \mathsf{list}\to\mathsf{nat}}$ (that returns the length of a list) with the dependency $\mathsf{length}_{\alpha\ \mathsf{list}} \rightsquigarrow \alpha\ \mathsf{list}$. When this theory is extended by a new polymorphic constant $\mathsf{size}_{\alpha\to\mathsf{nat}}$ and the definition

$$\mathsf{size}_{\alpha\ \mathsf{list}} \equiv \mathsf{length}_{\alpha\ \mathsf{list}}$$

two new dependencies $\mathsf{size}_{\alpha\ \mathsf{list}} \rightsquigarrow \mathsf{length}_{\alpha\ \mathsf{list}}$ and $\mathsf{size}_{\alpha\ \mathsf{list}} \rightsquigarrow \alpha\ \mathsf{list}$ are introduced. A cyclicity check may stop after covering the path from $\mathsf{size} \rightsquigarrow \mathsf{length}$, because it is already known that paths from length are composable and acyclic. Thus any path starting from $\mathsf{size}_{\alpha\ \mathsf{list}} \rightsquigarrow \alpha\ \mathsf{list}$ need not to be covered at all.

7 Conclusion

In this paper we have presented the theory and implementation of a formally verified cyclicity checker and its use in a verified theorem prover kernel that supports (ad-hoc) overloaded constant definitions. We demonstrated a verified binary cyclicity checker on theories of Isabelle/HOL and established that the definitions are acyclic. The verified binary was synthesised from our verified version through the CakeML infrastructure [23].

This work closes a gap in the foundation of Isabelle/HOL, in two ways. First, we establish the formalised theory for checking dependencies. Second, we discharge an assumption from earlier work [1], which strengthens the consistency by mechanised semantics.

Our verified kernel could be used as a verified proof checker for simple Isabelle/HOL theories. After accounting for differences Isabelle/HOL's and our kernel's logic, like axiomatic type classes, and implementing and verifying a performant cyclicity checker, our theorem prover kernel could proof-check Isabelle/HOL theories, and their dependencies.

References

1 Johannes Åman Pohjola and Arve Gengelbach. A Mechanised Semantics for HOL with Ad-hoc Overloading. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain,*

May 22-27, 2020, volume 73 of *EPiC Series in Computing*, pages 498–515. EasyChair, 2020. doi:10.29007/413d.

2 Rob Arthan. HOL constant definition done right. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 531–536. Springer, 2014. doi:10.1007/978-3-319-08970-6_34.

3 Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauß, and Klaus U. Schulz. Unification Theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001. doi:10.1016/b978-044450813-3/50010-2.

4 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.

5 Jared Curran Davis. *A self-verifying theorem prover*. The University of Texas at Austin, 2009.

6 Maxime Dénès. [Coq-Club] Propositional extensionality is inconsistent in Coq, Dec 2013. Coq Club Mailinglist. URL: https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html.

7 Arve Gengelbach and Johannes Åman Pohjola. Towards Correctly Checking for Cycles in Overloaded Definitions. Technical Report 2021-001, Department of Information Technology, Uppsala University, March 2021. URL: http://www.it.uu.se/research/publications/reports/2021-001/.

8 Arve Gengelbach, Johannes Åman Pohjola, and Tjark Weber. Mechanisation of Model-theoretic Conservative Extension for HOL with Ad-hoc Overloading. In Claudio Sacerdoti Coen and Alwen Tiu, editors, *Proceedings Fifteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2020, Paris, France, 29th June 2020*, volume 332 of *EPTCS*, pages 1–17, 2020. doi:10.4204/EPTCS.332.1.

9 Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.18.

10 John Harrison. Towards self-verification of HOL light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006. doi:10.1007/11814771_17.

11 Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2018. doi:10.1007/978-3-319-94205-6_42.

12 Jacques-Henri Jourdan. Coq pull request #90, Jul 2015. URL: https://github.com/coq/coq/pull/89.

13 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation. *J. Autom. Reason.*, 56(3):221–259, 2016. doi:10.1007/s10817-015-9357-x.

14 Ondřej Kunčar. Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 85–94. ACM, 2015. doi:10.1145/2676724.2693175.

15 Ondřej Kunčar and Andrei Popescu. A Consistent Foundation for Isabelle/HOL. *J. Autom. Reason.*, 62(4):531–555, 2019. doi:10.1007/s10817-018-9454-8.

16 Tobias Nipkow. What's in Main, December 2021. URL: https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/main.pdf.

17 Tobias Nipkow and Simon Roßkopf. Isabelle's Metalogic: Formalization and Proof Checker. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2021. doi:10.1007/978-3-030-79876-5_6.

18 Steven Obua. Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, pages 212–226. Springer, 2006. doi:10.1007/11805618_16.

19 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

20 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020. doi:10.1145/3371076.

21 Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.

22 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014. doi:10.1007/978-3-319-08970-6_32.

23 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi:10.1017/S0956796818000229.

24 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

25 Makarius Wenzel. The Isabelle/Isar Reference Manual, December 2021. URL: https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/isar-ref.pdf.

26 Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997. doi:10.1007/BFb0028402.