Correctness of Isabelle's Cyclicity Checker

Implementability of Overloading in Proof Assistants

Ondřej Kunčar
Technische Universität München
kuncar@in.tum.de

Abstract

Overloaded constant definitions are an important feature of the proof assistant Isabelle because they allow us to provide Haskell-like type classes to our users. There has been an ongoing question as to under which conditions we can practically guarantee that overloading is a safe theory extension, i.e., preserves consistency or is conservative. The natural condition is that a rewriting system generated by overloaded definitions must always terminate. The current system imposes restrictions on accepted overloaded definitions and decides the termination by an algorithm that is part of the trusted code base of Isabelle. Therefore we aim to prove its correctness.

Thanks to our work we discovered not only completeness shortcomings but also a correctness issue—we could prove False. In our paper we present a modified version of the algorithm together with a proof of completeness and correctness of it.

Although our work deals with Isabelle, our paper provides a more general result: how to practically implement overloading in proof assistants.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—Decision problems

Keywords Isabelle; overloading; overloaded constant definition; termination; cyclicity checker; interactive theorem proving; higher-order logic

1. Introduction

Isabelle introduces two important features: *overloaded constant definitions* (or *overloading*) and *axiomatic type classes*. Both are described by Wenzel [7]. Although these two features are usually used in Isabelle together to create Haskell-like type classes in the user space, we study overloading separately since we approach this feature from the foundational (logical) perspective.

Overloading is an integral part of Isabelle. Without it, it would not be possible to implement usable Haskell-like type classes, a feature that brings flexibility and reusability to the formalization work and has been therefore used by Isabelle users on a daily basis for more than 20 years.

Yet there exists also a certain price that must be paid for the flexibility of overloading. The consistency preservation (or conservativity) argument for overloading (seen as a theory extension) is not as straightforward as it is for plain constant definitions. We cannot argue that the symbol that is being defined has never been used before. This brings questions about the trustworthiness for one of the most fundamental features of the system—constant definitions.

Overloading was originally introduced by Nipkow and Qian [4] together with axiomatic type classes in Isabelle91. In 1997, Wenzel [7] provided a new definition of a safe theory extension and showed under which conditions the overloading mechanism meets this safeness criterion. But he assumed a simplified version of the system where the user must provide all the overloadings for a certain constant c at once.

Obua [5] noticed that although different overloadings for the same constant are not allowed to overlap in Isabelle2005, there does not exist any other guarantee for the consistency. Overloaded definitions are treated as pure axioms and the user is responsible for their consistency. He provided an example, which is accepted by Isabelle2005:

consts
$c :: \text{'a} \Rightarrow \text{bool}$
defs
$c\ (x :: \text{'a list} \times \text{'a}) \equiv c\ (\text{snd}\ x\ \#\ \text{fst}\ x)$
$c\ (x :: \text{'a list}) \equiv \neg c\ (\text{tl}\ x,\ \text{hd}\ x)$

The two definitions do not overlap but they lead to an inconsistency:

$$c[x] = \neg c([],\ x) = \neg c[x]$$

Obua identified that if we see the definitions as a rewriting system, this system has to terminate, otherwise preservation of consistency cannot be guaranteed. He implemented a private extension to Isabelle, which uses either an internal termination checker or external termination checkers to prove that the provided overloading definitions terminate. But as Obua noted, the termination of such a rewriting system is not semi-decidable if we allow full generality.

Wenzel proposed another solution: he imposed a certain well-formedness restriction (composability, see Definition 5.7) on the definitions such that termination becomes decidable and the users can still obtain enough flexibility. Using some work of Haftmann, Obua, Urban and others, he implemented a decision procedure, which has been a part of the kernel since Isabelle2007 without any significant changes.

Although Haftmann and Wenzel [3] described very abstractly some parts of this solution, there exists no convincing, complete proof that the algorithm decides termination of the corresponding rewriting system. Yet this algorithm is part of the trusted code of Isabelle and guarantees preservation of consistency of the system. This paper aims to close this gap. As explained in Section 7, thanks to our work we identified three issues in the original algorithm,

one of which allowed us to prove False in Isabelle and two others were sources of non-termination of the algorithm.[1] Based on this, we present a modified version of the algorithm that does not suffer from these issues. The inconsistency issue was already addressed in Isabelle2014 and we plan to amend the other two issues.

Although our paper concentrates on the system that we work with, only a little is specific to Isabelle and thus our paper provides more general result: how to practically implement overloading in proof assistants.

Concerning other proof assistants, to the best of our knowledge, there exists no notion of overloading in ACL2, HOL4, HOL Light and PVS. Mizar provides overloading for functions, types and other entities of the system (see a description by Grabowski et al. [2]). Moreover, there are two types of overloading: ad-hoc and parameter overloading. The whole mechanism of retrieving the meaning of an overloaded symbol is involved but it holds that after the theory is processed, each overloaded symbol has been resolved to a unique logical symbol. Overloading in Coq was implemented by Sozeau and Oury [6] in the context of first-class type classes. The approach uses the dictionary construction, i.e., during processing a theory, a type class function call is replaced by a projection from a certain dependent record, which represents a type class and whose concrete instance is found by a special tactic for an instance search.

In this context, let us briefly discuss design decisions that led to the current implementation of overloading in Isabelle. One could argue that using the dictionary approach (e.g., done by Sozeau and Oury [6] in Coq) is a better choice because this reduces the amount of trusted code. But the dictionary approach is not expressible in all possible logics that are implemented in Isabelle; notably in the most prominent one — Isabelle/HOL. The obstacle is types that use overloaded constants in their definitions. There are many such types in Isabelle/HOL; let us mention one as an example: a type of all red-black trees $(\alpha_{\mathsf{linorder}} \times \beta)\ \mathsf{rbt}$, which is defined by a restriction on all binary trees. As it is indicated in the type, the definition of rbt depends on a linear order for α, i.e., on a definition of the overloaded constant $\leq_{\alpha\to\alpha\to\mathsf{bool}}$ from the type class $\mathsf{linorder}$. If we used the dictionary construction, the type rbt would depend on a term, which is not expressible in Isabelle/HOL.

Another alternative would be to use external termination provers for term rewriting systems. But our internal implementation presented in this paper is more efficient than invoking the machinery of an external tool and moreover, it is proved to always terminate for the class of applications of overloading that we consider in this paper and use in Isabelle.

The paper is organized as follows: Section 2 presents informally overloading in Isabelle and how this translates to a termination problem. Section 3 introduces formal background for our proof. Section 4 states formally what we want to prove. The actual proof is distributed over Sections 5 and 6. The former reduces a termination of an infinite relation to a finite problem and the latter shows how to decide the finite problem. Section 7 lists the issues that we found in the original algorithm during our work and finally Section 8 concludes the paper.

2. Overloading and Dependency Relation

Let us informally introduce overloading and how this relates to termination. Let us assume we have declared a constant 0 of type α; we write 0_α. Overloading means that we define the meaning of 0 for different type instances separately. Let us take an example introduced by Wenzel [7]:

$$0_{\mathsf{nat}} = \mathit{zero}$$
$$0_{\alpha\times\beta} = (0_\alpha, 0_\beta)$$
$$0_{\alpha\to\beta} = \lambda x_\alpha.\ 0_\beta$$

Notice that for example 0 for products is specified only partially—we do not know how 0 is defined for α and β. One can get quickly into trouble; e.g., consider this example:

$$0_{\mathsf{nat}} = \ldots\, 0_{\mathsf{int}} \ldots$$
$$0_{\mathsf{int}} = \ldots\, 0_{\mathsf{nat}} \ldots$$

The two definitions are cyclic. In general, we can misuse a cycle to create an inconsistency in the system: just define $c_{\mathsf{bool}} = \neg c_{\mathsf{bool}}$. Another example causing trouble is

$$0_{\alpha\times\beta} = \ldots\, 0_{(\alpha\times\beta)\times\beta} \ldots.$$

This example leads to an infinite descent if we try to figure out the meaning of the constant by unfolding. Definitions are justified by the notion that they can be eliminated by unfolding them, yielding a finite (even if huge) term that contains no defined constant. Although it may be harmless, infinite descent breaks that argument. How can we detect such definitions?

We define a dependency relation as a binary relation $\rightsquigarrow$ on constants as follows: if $c = t$ is an overloaded definition of c, where t is a term, then we define $c \rightsquigarrow d$ for all constants d in t. Intuitively speaking, $c \rightsquigarrow d$ states that c depends on d meaning c was defined in terms of d. The relation $\rightsquigarrow\!\!\!\!\rightsquigarrow$ is then a transitive and substitutive closure of $\rightsquigarrow$. As Obua [5] described, a natural condition for overloading to be a safe theory extension is the fact that $\rightsquigarrow\!\!\!\!\rightsquigarrow$ terminates, i.e., there exists no infinite sequence c_i such that $c_0 \rightsquigarrow\!\!\!\!\rightsquigarrow c_1 \rightsquigarrow\!\!\!\!\rightsquigarrow \ldots$. In our case, $0_{\mathsf{nat}} \rightsquigarrow\!\!\!\!\rightsquigarrow 0_{\mathsf{int}} \rightsquigarrow\!\!\!\!\rightsquigarrow 0_{\mathsf{nat}} \rightsquigarrow\!\!\!\!\rightsquigarrow \ldots$ is the infinite sequence for the cycle example and $0_{\alpha\times\beta} \rightsquigarrow\!\!\!\!\rightsquigarrow 0_{(\alpha\times\beta)\times\beta} \rightsquigarrow\!\!\!\!\rightsquigarrow 0_{((\alpha\times\beta)\times\beta)\times\beta} \rightsquigarrow\!\!\!\!\rightsquigarrow \ldots$ for the infinite descent example. Our main theorem (Theorem 1) states that under certain assumptions,[2] we can decide when $\rightsquigarrow\!\!\!\!\rightsquigarrow$ terminates.

In the rest of the paper, it will not be important for us how we got $\rightsquigarrow\!\!\!\!\rightsquigarrow$, i.e., we forget our original motivation with overloading. We will work with $\rightsquigarrow\!\!\!\!\rightsquigarrow$ abstractly and focus only on the question under which conditions we can decide that this relation terminates. We also abstract from the type of the relation. Here we related constants but as we argue in the next section, the relation can be of any type as long as a certain abstract interface is provided on this type.

3. Formal Background

We use the notation $(p_i, q_i)_{i\leq n}$ for sequence $(p_0, q_0), \ldots, (p_n, q_n)$. The image of a function under a set is defined as $f[A] = \{f(x) \mid x \in A\}$. If $f : A \to B$ and $C \subseteq A$, the restriction of f to C is a function $f\restriction_C : C \to B$ defined as $f\restriction_C(x) = f(x)$ for all $x \in C$.

We fix a countably infinite set of type variables TVar, ranged over by α, β and γ. A *type signature* is a pair (K, arOf), where:

- K is a finite set of symbols called *type constructors*
- $\mathsf{arOf} : K \to \mathbb{N}$ is a function associating an arity with each type constructor

We often write K instead of (K, arOf) to indicate a type signature. The set Type_K, ranged over by τ, of *types* associated with a type signature K is defined inductively as follows:

- If $\alpha \in \mathsf{TVar}$, then $\alpha \in \mathsf{Type}_K$
- If $\tau_1, \ldots, \tau_n \in \mathsf{Type}_K$ and $k \in K$ such that $\mathsf{arOf}(k) = n$, then $(\tau_1, \ldots, \tau_n)k \in \mathsf{Type}_K$

[1] The original algorithm was primarily designed to achieve consistency preservation. Termination was not guaranteed and non-termination was meant as a last resort measure against introducing inconsistency. Our work identifies these non-terminations and shows how to eliminate them and yet still preserve consistency.

[2] Termination of $\rightsquigarrow\!\!\!\!\rightsquigarrow$ is in general not decidable because for example the Post correspondence problem can be encoded [5].

The *size function* counts the number of type constructors in a type: $\mathsf{size}(\alpha) = 0$ and $\mathsf{size}((\tau_1, \ldots, \tau_n)k) = 1 + \Sigma_{1 \le i \le n}\, \mathsf{size}(\tau_i)$.

A (*type*) *substitution* is a function $\rho : \mathsf{TVar} \to \mathsf{Type}_K$ that is almost everywhere the identity. We let TSubst_K denote the set of (type) substitutions, ranged over by ρ, σ, η. Each $\rho \in \mathsf{TSubst}_K$ is naturally extended to a function $\rho : \mathsf{Type}_K \to \mathsf{Type}_K$ by defining $\rho((\tau_1, \ldots, \tau_n)\, k) = (\rho(\tau_1), \ldots, \rho(\tau_n))\, k$.

The identity substitution id is defined as expected: $\mathsf{id}(x) = x$. $\mathsf{FV}(\tau)$ denotes the *set of type variables* of τ—formally $\mathsf{FV}(\tau) = \{\alpha \mid \exists \beta.\, \rho(\tau) \neq \tau \text{ where } \rho = \alpha \mapsto \beta\}$.

For a substitution σ, a *domain* is a (finite) set of variables $\mathsf{dom}(\sigma) = \{\alpha \mid \sigma(\alpha) \neq \alpha\}$, and an image is a set of types $\mathsf{img}(\sigma) = \sigma[\mathsf{dom}(\sigma)]$.

A *renaming* is a substitution σ such that $\mathsf{dom}(\sigma) = \mathsf{img}(\sigma)$ (and therefore each renaming is a bijection).

We assume that we have a function $\mathsf{Ren}(R, C)$ that gives us a renaming σ such that $\mathsf{dom}(\sigma) = R \cup \sigma[R]$ and $\sigma[R] \cap C = \emptyset$ (i.e., σ renames variables in R not to clash with variables in C). If R and C are finite, $\mathsf{Ren}(R, C)$ is always defined since TVar is infinite.

We write that $\rho =_\tau \rho'$ if $\rho(\alpha) = \rho'(\alpha)$ for all $\alpha \in \mathsf{FV}(\tau)$. We say that ρ is *equivalent* to ρ' and write $\rho \approx \rho'$ if there exists a renaming η such that $\rho = \eta \circ \rho'$. We write that $\rho \approx_\tau \rho'$ if there exists a renaming η such that $\rho =_\tau \eta \circ \rho'$.

Lemma 3.1. a) $\mathsf{size}((\eta \circ \rho)(\tau)) = \mathsf{size}(\rho(\tau))$ *if* η *is a renaming.*
b) $\mathsf{size}(\rho(\tau)) > \mathsf{size}(\tau)$ *if* $\rho \not\approx_{\tau'} \mathsf{id}$ *and* $\mathsf{FV}(\tau') \subseteq \mathsf{FV}(\tau)$.

Proof. A substitution that is not equivalent to the identity on τ has to map at least one type variable of τ to a type constructor, whereas a renaming cannot. □

Lemma 3.2. *If* η *is a renaming, then* $\eta[\mathsf{FV}(\tau)] = \mathsf{FV}(\eta(\tau))$.

Proof. Straightforward from definitions. □

We say that τ_1 is an *instance* of τ_2 via a substitution of ρ, written $\tau_1 \le_\rho \tau_2$, if $\rho(\tau_2) = \tau_1$. We say that τ_1 is an instance of τ_2, written $\tau_1 \le \tau_2$, if there exists ρ such that $\tau_1 \le_\rho \tau_2$.

Two types τ_1 and τ_2 are called *orthogonal*, written $\tau_1 \ \# \ \tau_2$, if they have no common instance, i.e., for all τ it holds that $\tau \not\le \tau_1$ or $\tau \not\le \tau_2$. Or equivalently, τ_1 and τ_2 are orthogonal if and only if τ_1 and τ_2 cannot be unified after renaming variables in τ_2 apart from variables in τ_1.

Two types τ_1 and τ_2 have a non-trivial instance, written $\tau_1 \downarrow \tau_2$, if there exists τ' such that $\tau' \le \tau_1$, $\tau' \le \tau_2$ and $\tau_1 \not\le \tau_2$ and $\tau_2 \not\le \tau_1$. Notice if $\tau_1 \downarrow \tau_2$ and $\tau' \le_\rho \tau_1$ and $\tau' \le_{\rho'} \tau_2$, then $\rho \not\approx \mathsf{id}$ and $\rho' \not\approx \mathsf{id}$.

Lemma 3.3. *Let* $\rho \approx_{\tau_1} \rho'$*; then* $\rho(\tau_1) \le \tau_2 \longleftrightarrow \rho'(\tau_1) \le \tau_2$, $\rho(\tau_1) \ge \tau_2 \longleftrightarrow \rho'(\tau_1) \ge \tau_2$, $\rho(\tau_1) \ \# \ \tau_2 \longleftrightarrow \rho'(\tau_1) \ \# \ \tau_2$ *and* $\rho(\tau_1) \downarrow \tau_2 \longleftrightarrow \rho'(\tau_1) \downarrow \tau_2$.

Proof. It can be proved by an easy manipulation with substitutions. □

A *signature* Σ is a tuple $(K, \mathsf{arOf}, C, \mathsf{tyOf})$ where:

- (K, arOf) is a type signature
- C is a finite set of *constants*
- $\mathsf{tyOf} : C \to \mathsf{Type}_K$ is a function associating a type with every constant

A *constant instance* is a pair of a constant c and a type $\tau \le \mathsf{tyOf}(c)$, written c_τ. We write $\mathsf{CI}^{\downarrow}_{\Sigma}$ for the set of constant instances of a signature Σ.

Now we can easily lift the functions and relations FV, dom, img, Ren, $\le, \approx, \ \# \ , \downarrow$ and size from Type_K to $\mathsf{CI}^{\downarrow}_{\Sigma}$. We do not have to do the lifting separately for each of them because there exists a more elegant way: all of these functions and relations were defined in terms of three concepts: equality $=$, application of a substitution, and the size function. Therefore in order to complete the lifting from Type_K to $\mathsf{CI}^{\downarrow}_{\Sigma}$, it suffices if we define $= : \mathsf{CI}^{\downarrow}_{\Sigma} \to \mathsf{CI}^{\downarrow}_{\Sigma} \to \mathsf{bool}$, $\mathsf{App} : (\mathsf{Type}_K \to \mathsf{Type}_K) \to \mathsf{CI}^{\downarrow}_{\Sigma} \to \mathsf{CI}^{\downarrow}_{\Sigma}$ and $\mathsf{size} : \mathsf{CI}^{\downarrow}_{\Sigma} \to \mathbb{N}$, which we do as follows:

1. $c_{\tau_1} = d_{\tau_2}$ iff $c = d$ and $\tau_1 = \tau_2$.
2. $\mathsf{App}\, \rho\, c_\tau = c_{\rho(\tau)}$
3. $\mathsf{size}(c_\tau) = \mathsf{size}(\tau)$

For readability reasons, we still write $\rho(p)$ for $\mathsf{App}\, \rho\, p$.

We can generalize the construction that we did now for $\mathsf{CI}^{\downarrow}_{\Sigma}$ even more: we can replace $\mathsf{CI}^{\downarrow}_{\Sigma}$ by a general universe $\mathcal{U}_\Sigma$ as long as three functions $= : \mathcal{U}_\Sigma \to \mathcal{U}_\Sigma \to \mathsf{bool}$, $\mathsf{App} : (\mathsf{Type}_K \to \mathsf{Type}_K) \to \mathcal{U}_\Sigma \to \mathcal{U}_\Sigma$ and $\mathsf{size} : \mathcal{U}_\Sigma \to \mathbb{N}$ are provided.[3] The reason why we care so much about this generality is that although the dependency relation $\rightsquigarrow$ in Isabelle2014 relates constants ($\mathsf{CI}^{\downarrow}_{\Sigma}$), we would like to propose to track richer dependencies, namely constants and types ($\mathsf{CI}^{\downarrow}_{\Sigma} \uplus \mathsf{Type}_K$).[4] Therefore we use $\mathcal{U}_\Sigma$ (ranged over by p, q, r and s) instead of $\mathsf{CI}^{\downarrow}_{\Sigma}$ in the rest of our text.

Given any binary relation R on a set $\mathcal{U}_\Sigma$, we write R^+ for its transitive closure and $R^\downarrow$ for its substitutive closure, defined as follows: $p\, R^\downarrow\, q$ iff there exist $p', q' \in \mathcal{U}_\Sigma$ and $\rho \in \mathsf{TSubst}_\Sigma$ such that $p = \rho(p')$, $q = \rho(q')$ and $p'\, R\, q'$. We say that R is *substitutive* if it closed under substitution, i.e., $p\, R^\downarrow\, q$ implies $\rho(p)\, R^\downarrow\, \rho(q)$ for all substitutions ρ.

Lemma 3.4. $R^\downarrow$ includes R and is substitutive.

Proof. Immediate using the identity and composition of substitutions. □

Let $\rightsquigarrow\!\!\!\!\!\rightsquigarrow$ be $\rightsquigarrow^{\downarrow +}$, i.e., the transitive closure of $\rightsquigarrow^{\downarrow}$.

Lemma 3.5. $\rightsquigarrow\!\!\!\!\!\rightsquigarrow$ is substitutive (and hence so is the transitive, substitutive closure of $\rightsquigarrow$).

Proof. By induction on the definition of transitive closure. □

We say that a relation R on a set $\mathcal{U}_\Sigma$ *terminates* if there exists no sequence $(p_i)_{i \in \mathbb{N}}$ such that $p_i\, R\, p_{i+1}$ for all i.

4. What We Want to Prove

Theorem 1. *There exists a predicate* P *on binary relations on* $\mathcal{U}_\Sigma$ *such that for finite relations* $\rightsquigarrow$ *the following holds:*

- $P(\rightsquigarrow)$ *is decidable*
- $P(\rightsquigarrow)$ *implies that* $\rightsquigarrow\!\!\!\!\!\rightsquigarrow$ *terminates*
- P *contains interesting relations* $\rightsquigarrow$

The last condition is a bit vague. We will make it more precise later. Intuitively, we are not interested in P that is false everywhere but P should cover all of our use cases of overloading in Isabelle.

[3] Strictly speaking, in full generality, $=$, App and size have to fulfill some assumptions; e.g., $\mathsf{App}\, \mathsf{id} = \mathsf{id}$ and $\mathsf{App}\, (\rho \circ \sigma) = (\mathsf{App}\, \rho) \circ (\mathsf{App}\, \sigma)$, and size has properties like in Lemma 3.1.

[4] Without going much into the details, tracking dependencies of constants and types would allow us to implement type definitions in Isabelle/HOL as a consistency preserving theory extension. We plan this for future work.

5. From Non-Termination to Cyclicity

Since $\rightsquigarrow\!\!\!\!\!\rightarrow$ is the transitive and substitutive closure of $\rightsquigarrow$, $\rightsquigarrow\!\!\!\!\!\rightarrow$ is generally infinite even if $\rightsquigarrow$ is finite. In this section, we show that the problem of termination of infinite $\rightsquigarrow\!\!\!\!\!\rightarrow$ is equivalent to a finite problem on $\rightsquigarrow$, namely that $\rightsquigarrow$ is acyclic. We will define cyclicity formally later, but informally it means that we can find a finite sequence $r_0 \rightsquigarrow\!\!\!\!\!\rightarrow r_1 \rightsquigarrow\!\!\!\!\!\rightarrow \ldots \rightsquigarrow\!\!\!\!\!\rightarrow r_n$ such that $r_n \leq r_0$. If we find a cycle, it is easy to show that $\rightsquigarrow\!\!\!\!\!\rightarrow$ does not terminate. To prove the other direction is involved.

If $\rightsquigarrow\!\!\!\!\!\rightarrow$ does not terminate, it means there exists an infinite sequence $(p_i)_{i\in\mathbb{N}}$ such that $p_0 \rightsquigarrow\!\!\!\!\!\rightarrow p_1 \rightsquigarrow\!\!\!\!\!\rightarrow p_2 \rightsquigarrow\!\!\!\!\!\rightarrow \ldots$. We could hope that we could find a cycle as a subsequence of $(p_i)_{i\in\mathbb{N}}$ but the following example shows that this is not always the case: consider $\rightsquigarrow$ defined as $\alpha \rightsquigarrow \alpha\ \mathsf{list}$. Now $\mathsf{nat} \rightsquigarrow\!\!\!\!\!\rightarrow \mathsf{nat\ list} \rightsquigarrow\!\!\!\!\!\rightarrow \mathsf{nat\ list\ list} \rightsquigarrow\!\!\!\!\!\rightarrow \ldots$ is a non-terminating sequence but no subsequence of it is a cycle since all elements of this sequence are incomparable. But another sequence $\alpha \rightsquigarrow\!\!\!\!\!\rightarrow \alpha\ \mathsf{list} \rightsquigarrow\!\!\!\!\!\rightarrow \alpha\ \mathsf{list\ list} \rightsquigarrow\!\!\!\!\!\rightarrow \ldots$ contains a lot of cycles (e.g., $\alpha\ \mathsf{list} \leq \alpha$). Our intuition is that if we want to find a cycle, we have to search in sequences that are as general instances as possible. We formalize this intuition now.

Definition 5.1. *$\rightsquigarrow$ is* monotone *if for each p, q such that $p \rightsquigarrow q$, we have $\mathsf{FV}(q) \subseteq \mathsf{FV}(p)$.*

Lemma 5.2. *Let us assume that*

- *$p_i \rightsquigarrow q_i$ for all $i \leq n$,*
- *$\rightsquigarrow$ is monotone.*

If $(\rho_i)_{i\leq n}$ is a sequence of substitutions such that we have $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all $i < n$, then $\mathsf{FV}(\rho_i(p_i)) \supseteq \mathsf{FV}(\rho_i(q_i)) \supseteq \mathsf{FV}(\rho_j(p_j)) \supseteq \mathsf{FV}(\rho_j(q_j))$ holds for all $0 \leq i < j \leq n$.

Proof. If $\mathsf{FV}(p) \supseteq \mathsf{FV}(q)$, then $\mathsf{FV}(\rho(p)) \supseteq \mathsf{FV}(\rho(q))$ for any ρ. Therefore it suffices to prove $\mathsf{FV}(\rho_i(q_i)) \supseteq \mathsf{FV}(\rho_j(p_j))$, which can be done by backward induction. □

Monotonicity is a natural notion if we remember the original motivation for $\rightsquigarrow$: given a definition of an overloaded constant $c = t$, it must hold that $\mathsf{FV}(t) \subseteq \mathsf{FV}(c)$. Monotonicity gives $\rightsquigarrow$ more regular structure, which allows us to simplify some definitions and is crucial in some coming proofs—especially the following consequence of Lemma 5.2: if we know that ρ' has some effect on $\rho_n(g_n)$ (i.e., $\mathsf{dom}(\rho') \cap \mathsf{FV}(\rho_n(g_n)) \neq \emptyset$), we know that ρ' has also some effect on $\rho_0(p_0)$.

Definition 5.3. *We say that a sequence of substitutions $(\rho_i)_{i\leq n}$ is a* solution *to the sequence $(p_i, q_i)_{i\leq n}$ if $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all $i < n$.*

We say that a solution $(\rho_i)_{i\leq n}$ to $(p_i, q_i)_{i\leq n}$ is the most general solution *if for any other solution $(\rho'_i)_{i\leq n}$ there exists a sequence of substitutions $(\eta_i)_{i\leq n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$.*

Thanks to monotonicity, we can talk only about p_is and omit q_is in the last definition because we will be able to derive from $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ also $\rho'_i(q_i) = (\eta_i \circ \rho_i)(q_i)$ provided $\rightsquigarrow$ is monotone and $p_i \rightsquigarrow q_i$.

Lemma 5.4. *If $(\rho_i)_{i\leq n}$ and $(\rho'_i)_{i\leq n}$ are both the most general solutions to the sequence $(p_i, q_i)_{i\leq n}$, then $\rho_i \approx_{p_i} \rho'_i$ for all $i \leq n$.*

Proof. Since $(\rho_i)_{i\leq n}$ and $(\rho'_i)_{i\leq n}$ are both the most general solutions, there exist $(\eta_i)_{i\leq n}$ and $(\eta'_i)_{i\leq n}$ such that $\rho_i(p_i) = (\eta_i \circ \rho'_i)(p_i)$ and $\rho'_i(p_i) = (\eta'_i \circ \rho_i)(p_i)$ for all $i \leq n$. Now $\rho_i(p_i) = ((\eta_i \circ \eta'_i) \circ \rho_i)(p_i)$ and $\rho'_i(p_i) = ((\eta'_i \circ \eta_i) \circ \rho'_i)(p_i)$. Thus $\eta_i \circ \eta'_i =_{\rho_i(p_i)} \mathsf{id}$ and $\eta'_i \circ \eta_i =_{\rho'_i(p_i)} \mathsf{id}$ and therefore $\eta_i \restriction_{\mathsf{FV}(\rho'_i(p_i))}$ is a bijection between $\mathsf{FV}(\rho'_i(p_i))$ and $\mathsf{FV}(\rho_i(p_i))$. There exists surely a bijection $\hat{\eta}$ between $\mathsf{FV}(\rho_i(p_i)) \setminus \mathsf{FV}(\rho'_i(p_i))$ and $\mathsf{FV}(\rho'_i(p_i)) \setminus \mathsf{FV}(\rho_i(p_i))$. Then the function $\eta_i \restriction_{\mathsf{FV}(\rho'_i(p_i))} \circ \hat{\eta}$ is a renaming that witnesses $\rho_i \approx_{p_i} \rho'_i$. □

We defined a notion of a most general solution and proved that most general solutions are unique modulo renaming. This notion formalizes our intuition that we should look for cycles in sequences that are as general instances as possible. Most general solutions define such sequences. Let us get back to our example: let $(p_i, q_i)_{i\leq 1} = (\alpha, \alpha\ \mathsf{list}), (\alpha, \alpha\ \mathsf{list})$. Then $(\rho_i)_{i\leq 1} = \alpha \mapsto \alpha\ \mathsf{list}, \alpha \mapsto \alpha\ \mathsf{list\ list}$ is the most general solution to $(p_i, q_i)_{i\leq 1}$ and yields a sequence $\alpha \rightsquigarrow\!\!\!\!\!\rightarrow \alpha\ \mathsf{list} \rightsquigarrow\!\!\!\!\!\rightarrow \alpha\ \mathsf{list\ list}$, which contains cycles. On the other hand, $(\rho'_i)_{i\leq 1} = \alpha \mapsto \mathsf{nat\ list}, \alpha \mapsto \mathsf{nat\ list\ list}$ is only a solution but not the most general one. $(\rho'_i)_{i\leq 1}$ yields this sequence $\mathsf{nat} \rightsquigarrow\!\!\!\!\!\rightarrow \mathsf{nat\ list} \rightsquigarrow\!\!\!\!\!\rightarrow \mathsf{nat\ list\ list}$, which does not contain any cycle.

Given a non-terminating sequence $p_0 \rightsquigarrow\!\!\!\!\!\rightarrow p_1 \rightsquigarrow\!\!\!\!\!\rightarrow p_2 \rightsquigarrow\!\!\!\!\!\rightarrow \ldots$, how do we construct a most general solution to its subsequences? And does a most general solution always exist? We will prove that if we already have a most general solution to first n elements of the infinite sequence, we can always extend this most general solution to $n+1$ elements. I.e., we will provide an inductive description of most general solutions. In order to achieve this, we need to first introduce some additional notions.

Definition 5.5. *We say that sequences $(p_i, q_i)_{i\leq n}$ and $(\rho_i)_{i\leq n}$ form a* path *starting at k under $\rightsquigarrow$ and write $(p_i, q_i, \rho_i)_{k\leq i\leq n}^{\rightsquigarrow}$ if*

- *$p_i \rightsquigarrow q_i$ for all $k \leq i \leq n$,*
- *$\rho_k \approx_{p_k} \mathsf{id}$,*
- *$(\rho_i)_{k\leq i\leq n}$ is a solution to $(p_i, q_i)_{k\leq i\leq n}$.*

If $k = 0$, we usually omit this index.

A path is a sequence together with its solution, which is allowed to only rename the first element of the sequence (i.e., not to apply a non-trivial substitution).

Definition 5.6. *We say that $\rightsquigarrow$ is* cyclic *if there exists a path $(p_i, q_i, \rho_i)_{i\leq n}^{\rightsquigarrow}$ and $\rho_n(q_n) \leq p_0$.*

The formal definition of a cycle admits only a strict subset of cycles that we informally introduced at the beginning of this section, namely they have to be paths. For example, if $\alpha \rightsquigarrow \alpha\ \mathsf{list}$, then $\beta, \beta\ \mathsf{list}$ is a cycle[5] for us, whereas $\alpha\ \mathsf{list}$, $\alpha\ \mathsf{list\ list}$ is not.

Definition 5.7. *$\rightsquigarrow$ is* composable *if for all p and q such that $p \rightsquigarrow q$ and for each path $(p_i, q_i, \rho_i)_{i\leq n}^{\rightsquigarrow}$, it holds that either $\rho_n(q_n) \leq p$, or $p \leq \rho_n(q_n)$, or $\rho_n(q_n) \ \#\ p$.*

Composability is an important restriction on $\rightsquigarrow$. It reduces the search space when we are looking for a most general solution. Later we will prove that each sequence defined by a most general solution has as a suffix a path. Therefore if we already have a most general solution to n first elements, the composability tells us that there exist three cases concerning the extension of this most general solution: in two cases we can still (possibly) extend the sequence ($\rho_n(q_n) \leq p$ or $p \leq \rho_n(q_n)$) and in one case we cannot ($\rho_n(q_n) \ \#\ p$). But we prove in the following lemma that if there exists already some solution, the case $\rho_n(q_n) \ \#\ p$ cannot occur.

Lemma 5.8. *Let us assume that*

- *$(\rho_i)_{i\leq n+1}$ is a solution to $(p_i, q_i)_{i\leq n+1}$,*
- *$p_i \rightsquigarrow q_i$ for all $i \leq n+1$,*
- *$\rightsquigarrow$ is monotone and composable,*
- *$(\rho'_i)_{i\leq n}$ is the most general solution to $(p_i, q_i)_{i\leq n}$,*
- *there exists $k \leq n$ such that $(p_i, q_i, \rho'_i)_{k\leq i\leq n}^{\rightsquigarrow}$.*

[5] We should write a path formally as $(\beta, \beta\ \mathsf{list}, \alpha \mapsto \beta)^{\rightsquigarrow}$ but we simplify our (heavy) notation in an informal description.

Then $\rho'_n(q_n) \leq p_{n+1}$ or $p_{n+1} \leq \rho'_n(q_n)$.

Proof. From composability it follows that $\rho'_n(q_n) \leq p_{n+1}$, or $p_{n+1} \leq \rho'_n(q_n)$, or $\rho'_n(q_n) \# p_{n+1}$. Since $(\rho'_i)_{i\leq n}$ is the most general solution, there exists $(\eta_i)_{i\leq n}$ such that $\rho_i(p_i) = (\eta_i \circ \rho'_i)(p_i)$ for all $i \leq n$ and by monotonicity also $\rho_i(q_i) = (\eta_i \circ \rho'_i)(q_i)$ for all $i \leq n$. Therefore we can rewrite $\rho_n(q_n) = \rho_{n+1}(p_{n+1})$ to $\eta_n(\rho'_n(q_n)) = \rho_{n+1}(p_{n+1})$, which means there exists a common instance of $\rho'_n(q_n)$ and p_{n+1} and thus only two cases $\rho'_n(q_n) \leq p_{n+1}$ or $p_{n+1} \leq \rho'_n(q_n)$ can occur. □

The last lemma shows that an existence of some solution (see this solution as a subsequence of our non-terminating sequence) and composability guarantee that we are left with two cases. The two following lemmas show that the extension is always possible in either of the cases and give us concrete instructions how to do it; i.e., how to extend a most general solution from n to $n+1$ elements.

Lemma 5.9. *Let*

- *$(\rho_i)_{i\leq n}$ be the most general solution to $(p_i, q_i)_{i\leq n}$,*
- *$p_i \rightsquigarrow q_i$ for all $i \leq n$,*
- *$\rightsquigarrow$ be monotone,*
- *$\rho_n(q_n) \leq_{\rho'} p_{n+1}$.*

Then $(\rho_i)_{i\leq n}, \rho'$ is the most general solution to $(p_i, q_i)_{i\leq n+1}$.

Proof. The sequence $(\rho_i)_{i\leq n}, \rho'$ is a solution to $(p_i, q_i)_{i\leq n+1}$. We prove that it is the most general solution. Let $(\rho'_i)_{i\leq n+1}$ be a solution to $(p_i, q_i)_{i\leq n+1}$. Then $(\rho'_i)_{i\leq n}$ is surely a solution to $(p_i, q_i)_{i\leq n}$. Therefore there exists $(\eta_i)_{i\leq n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ for all $i \leq n$ and by monotonicity also $\rho'_i(q_i) = (\eta_i \circ \rho_i)(q_i)$. From this and $\rho'_n(q_n) = \rho'_{n+1}(p_{n+1})$ (since $(\rho'_i)_{i\leq n+1}$ is a solution), it follows that $\eta_n(\rho_n(q_n)) = \rho'_{n+1}(p_{n+1})$ and since $\rho_n(q_n) = \rho'(p_{n+1})$, we get finally $\eta_n(\rho'(p_{n+1})) = \rho'_{n+1}(p_{n+1})$. Define $\eta_{n+1} := \eta_n$. □

Lemma 5.10. *Let*

- *$(\rho_i)_{i\leq n}$ be the most general solution to $(p_i, q_i)_{i\leq n}$,*
- *$p_i \rightsquigarrow q_i$ for all $i \leq n$,*
- *$\rightsquigarrow$ be monotone,*
- *$\rho_n(q_n) \geq_{\rho'} p_{n+1}$.*

There exists a substitution $\hat{\rho}$ such that the sequence $(\hat{\rho} \circ \rho_i)_{i\leq n}, \mathsf{id}$ is the most general solution to $(p_i, q_i)_{i\leq n+1}$ and $\hat{\rho} =_{\rho_n(q_n)} \rho'$.

Proof. Since the formal proof is technical, we explain some ideas of it first on a little example: Let us have:

$$(p_i, q_i)_{i\leq 1} = (\alpha \times \beta, \beta), (\alpha\ \mathsf{list}, \alpha)$$
$$(\rho_i)_{i\leq 0} = \mathsf{id}$$

$(\rho_i)_{i\leq 0}$ is trivially the most general solution to $(p_i, q_i)_{i\leq 0}$. $q_0 \leq_{\rho'} p_1$, where $\rho' = \beta \mapsto \alpha\ \mathsf{list}$. Let us define $(\rho_i^?)_{i\leq 1}$, a candidate for a most general solution to $(p_i, q_i)_{i\leq 1}$, as $(\rho_i^?)_{i\leq 1} = \rho' \circ \rho_0, \mathsf{id}$. That is to say, we try setting $\hat{\rho} = \rho'$. $(\rho_i^?)_{i\leq 1}$ is a solution to $(p_i, q_i)_{i\leq 1}$ because it yields a sequence

$$(\alpha \times \alpha\ \mathsf{list}, \alpha\ \mathsf{list}), (\alpha\ \mathsf{list}, \alpha). \tag{1}$$

But in general it is not a most general solution: Let us take $(\rho'_i)_{i\leq 1} = [\alpha \mapsto \mathsf{int}, \beta \mapsto \mathsf{nat\ list}], \alpha \mapsto \mathsf{nat}$, which is a solution to $(p_i, q_i)_{i\leq 1}$ because it yields this sequence

$$(\mathsf{int} \times \mathsf{nat\ list}, \mathsf{nat\ list}), (\mathsf{nat\ list}, \mathsf{nat}). \tag{2}$$

If $(\rho_i^?)_{i\leq 1}$ were the most general solution, we should be able to find $(\eta_i)_{i\leq 1}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i^?)(p_i)$ for all $i = 0, 1$. But this is not possible since by comparing sequences (1) and (2) we derive $\eta_0(\alpha) = \mathsf{int}$ and $\eta_0(\alpha) = \mathsf{nat}$.

We have to first rename the type variable from p_0 that is not in q_0 such that it does not clash with variables from $\rho'(q_0)$ and q_0: we use a renaming $\sigma = \alpha \mapsto \gamma$. Observe that $(\sigma \circ \rho_i)_{i\leq 0}$ is a solution to $(p_i, q_i)_{i\leq 0}$. We define $\hat{\rho} = \rho' \circ \sigma$ and then a new candidate for the most general solution $\hat{\rho} \circ \rho_0, \mathsf{id}$ yields

$$(\gamma \times \alpha\ \mathsf{list}, \alpha\ \mathsf{list}), (\alpha\ \mathsf{list}, \alpha).$$

Now we can find $(\eta_i)_{i\leq 1}$: we obtain $\eta(\gamma)$ from comparing $(\sigma \circ \rho_i)_{i\leq 0}$ and $(\rho'_i)_{i\leq 0}$ (the former is a most general solution and the latter a solution) and $\eta(\alpha)$ from comparing $\rho'(q_0) = p_1$ and $\rho'_1(p_1)$ (which trivially yields $\eta(\alpha) = \rho'_1(\alpha)$).

Now we proceed to carry out the formal proof. Assume

$$\mathsf{dom}(\rho') \subseteq \mathsf{FV}(\rho_n(q_n)). \tag{3}$$

If it were not the case, we would use $\rho' \restriction_{\mathsf{FV}(\rho_n(q_n))}$ instead of ρ'.

Let $R := \mathsf{FV}(\rho_0(p_0)) \setminus \mathsf{FV}(\rho_n(q_n))$ and $C := \mathsf{FV}(\rho_n(q_n)) \cup \mathsf{FV}((\rho' \circ \rho_n)(q_n))$. We define $\sigma := \mathsf{Ren}(R, C)$ and obtain the two following properties of σ from the definition of Ren:

$$\mathsf{dom}(\sigma) \cap \mathsf{FV}(\rho_n(q_n)) = \emptyset \tag{4}$$
$$\sigma[R] \cap \mathsf{FV}((\rho' \circ \rho_n)(q_n)) = \emptyset \tag{5}$$

Let $\tilde{\rho}_i := \sigma \circ \rho_i$ for each $i \leq n$. Obviously $(\tilde{\rho}_i)_{i\leq n}$ is a solution to the sequence $(p_i, q_i)_{i\leq n}$. Now we prove that it is the most general solution: let us take another solution $(\rho'_i)_{i\leq n}$; thus there exists $(\eta_i)_{i\leq n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$. Define $\tilde{\eta}_i := \eta_i \circ \sigma^{-1}$. Then $\rho'(p_i) = (\tilde{\eta}_i \circ \tilde{\rho}_i)(p_i)$.

From (4) it follows that $\tilde{\rho}_n(q_n) \geq_{\rho'} p_{n+1}$. Let $(\overline{\rho}_i)_{i\leq n+1} := (\rho' \circ \tilde{\rho}_i)_{i\leq n}, \mathsf{id}$. The sequence $(\overline{\rho}_i)_{i\leq n+1}$ is clearly a solution to $(p_i, q_i)_{i\leq n+1}$. We prove that it is the most general solution. Let $(\rho'_i)_{i\leq n+1}$ be a solution to $(p_i, q_i)_{i\leq n+1}$ then $(\rho'_i)_{i\leq n}$ is clearly a solution to $(p_i, q_i)_{i\leq n}$. Thus $(\eta_i)_{i\leq n}$ exists such that

$$\rho'_i(p_i) = (\eta_i \circ \tilde{\rho}_i)(p_i) \text{ for all } i \leq n. \tag{6}$$

We define $(\overline{\eta})_{i\leq n+1}$ as follows: if $i = n+1$ then $\overline{\eta}_{n+1} := \rho'_{n+1}$, otherwise ($i \leq n$)

$$\overline{\eta}_i(x) := \begin{cases} \rho'_{n+1}(x) & \text{if } x \in \mathsf{FV}((\rho' \circ \rho_n)(q_n)) \\ \eta_i(x) & \text{otherwise.} \end{cases}$$

We prove by backward induction that $\rho'_i(p_i) = (\overline{\eta}_i \circ \overline{\rho}_i)(p_i)$ for all $i \leq n+1$. Base case ($i = n+1$): Since $\overline{\rho}_{n+1} = \mathsf{id}$, we get immediately $\rho'_{n+1} = \overline{\eta}_{n+1} \circ \overline{\rho}_{n+1}$. Inductive step: $0 \leq i < n+1$ and $\rho'_{i+1}(p_{i+1}) = (\overline{\eta}_{i+1} \circ \overline{\rho}_{i+1})(p_{i+1})$. We prove

$$\eta_i(x) = (\overline{\eta}_i \circ \rho')(x) \text{ for all } x \in \mathsf{FV}(\tilde{\rho}_i(p_i)) \tag{7}$$

by the case distinction:

a) $x \in \mathsf{FV}(\tilde{\rho}_i(p_i)) \setminus \mathsf{FV}(\rho_n(q_n))$: therefore $\rho'(x) = x$ by (3) and since $x \in \sigma[R]$ (by Lemmas 3.2 and 5.2), $\eta_i(x) = \overline{\eta}_i(x)$ follows from (5). Therefore $\eta_i(x) = (\overline{\eta}_i \circ \rho')(x)$.

b) $x \in \mathsf{FV}(\rho_n(q_n))$: $(\eta_i \circ \tilde{\rho}_i)(q_i) = (\overline{\eta}_{i+1} \circ \rho' \circ \tilde{\rho}_i)(q_i)$ holds since

$$\begin{aligned} (\eta_i \circ \tilde{\rho}_i)(q_i) &= \rho'_i(q_i) && \text{by mon. and (6)} \\ &= \rho'_{i+1}(p_{i+1}) && (\rho'_i)_{i\leq n+1} \text{ is sol.} \\ &= (\overline{\eta}_{i+1} \circ \overline{\rho}_{i+1})(p_{i+1}) && \text{by IH} \\ &= (\overline{\eta}_{i+1} \circ \rho' \circ \tilde{\rho}_i)(q_i) && (\overline{\rho_i})_{i\leq n+1} \text{ is sol.} \end{aligned}$$

Thus $\eta_i(y) = (\overline{\eta}_{i+1} \circ \rho')(y)$ for all $y \in \mathsf{FV}(\tilde{\rho}_i(q_i))$. But by Lemma 5.2 and by (4) $x \in \mathsf{FV}(\tilde{\rho}_i(q_i))$. Therefore $\eta_i(x) = (\overline{\eta}_{i+1} \circ \rho')(x) = (\rho'_{n+1} \circ \rho')(x) = (\overline{\eta}_i \circ \rho')(x)$.

We know that $\rho'_i(p_i) = (\eta_i \circ \tilde{\rho}_i)(p_i)$ and by using (7) we get $\rho'_i(p_i) = (\overline{\eta}_i \circ \rho' \circ \tilde{\rho}_i)(p_i) = (\overline{\eta}_i \circ \overline{\rho}_i)(p_i)$, which concludes the proof that $(\overline{\rho}_i)_{i\leq n+1}$ is the most general solution.

Let $\hat{\rho} := \rho' \circ \sigma$ then obviously $(\overline{\rho}_i)_{i\leq n+1} = ((\rho' \circ \tilde{\rho}_i)_{i\leq n}, \mathsf{id}) = ((\hat{\rho} \circ \rho_i)_{i\leq n}, \mathsf{id})$. The equality $\hat{\rho} =_{\rho_n(q_n)} \rho'$ follows from (4). □

Now nothing prevents us from combining the previous results and proving that there is always a most general solution if some solution already exists.

Lemma 5.11. *Let us assume that*

- $(\rho_i)_{i\leq n}$ *is a solution to* $(p_i, q_i)_{i\leq n}$,
- $p_i \rightsquigarrow q_i$ *for all* $i \leq n$,
- $\rightsquigarrow$ *is monotone and composable,*

then there exist a most general solution $(\rho'_i)_{i\leq n}$ *to* $(p_i, q_i)_{i\leq n}$ *and* $k \leq n$ *such that* $(p_i, q_i, \rho'_i)_{k\leq i\leq n}^{\rightsquigarrow}$.

Proof. By induction on the length of the sequence. Base case $n = 0$: define $\rho'_0 = \mathsf{id}$. Inductive step $n = i + 1$: we assume that $(\rho_j)_{j\leq i+1}$ is a solution to $(p_j, q_j)_{j\leq i+1}$. Then $(\rho_j)_{j\leq i}$ is surely a solution to $(p_j, q_j)_{j\leq i}$ and thus by the induction hypothesis we obtain the most general solution $(\rho'_j)_{j\leq i}$ to $(p_j, q_j)_{j\leq i}$ and $k \leq i$ such that $(p_j, q_j, \rho'_j)_{k\leq j\leq i}^{\rightsquigarrow}$.

By Lemma 5.8, only two cases can occur:

a) $\rho'_i(q_i) \leq_{\rho'} p_{i+1}$, then by Lemma 5.9 $(\rho'_j)_{j\leq i}, \rho'$ is the most general solution to $(p_j, q_j)_{j\leq i+1}$ and still $\rho'_k \approx_{p_k} \mathsf{id}$.
b) $\rho'_i(q_i) \geq_{\rho'} p_{i+1}$, then by Lemma 5.10 there exists $\hat{\rho}$ such that the sequence $\hat{\rho} \circ \rho'_1, \ldots, \hat{\rho} \circ \rho'_i, \mathsf{id}$ is the most general solution to $(p_j, q_j)_{j\leq i+1}$. Then $k = i + 1$ and obviously $\rho'_k \approx_{p_k} \mathsf{id}$.

□

We proved even more: a sequence defined by a most general solution has always as a suffix a path. This is an important result for us because we claimed that we can look for cycles in sequences produced by most general solutions and we did define a cycle such that each cycle is a path. Thus this suffix is a candidate for a cycle. To find a real cycle among these candidates, we extend this suffix potentially ad infinitum in our proof, i.e., we find an infinite sequence where each prefix is a path. To capture this idea we introduce new notions.

Definition 5.12. *We write* $(p_i, q_i)_{i\leq n} \preceq p$ *if there exists a most general solution* $(\rho_i)_{i\leq n}$ *to* $(p_i, q_i)_{i\leq n}$ *and if* $\rho_n(q_n) \leq p$. *We define* $(p_i, q_i)_{i\leq n} \succeq p$ *analogously.*

Corollary 5.13. *Let* $\rightsquigarrow$ *be composable and monotone, and let us have sequences* $(p_i, q_i)_{i\leq n+1}$ *and* $(\rho_i)_{i\leq n+1}$ *such that* $(\rho_i)_{i\leq n+1}$ *is a solution to* $(p_i, q_i)_{i\leq n+1}$ *and* $p_i \rightsquigarrow q_i$ *for all* $i \leq n + 1$. *Then* $(p_i, q_i)_{i\leq n} \preceq p_{i+1}$ *or* $(p_i, q_i)_{i\leq n} \succeq p_{i+1}$.

Proof. From Lemmas 5.8 and 5.11. □

Definition 5.14. *A sequence* $(p_0, q_0), (p_1, q_1), \ldots$ *is called* ascending *if* $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \preceq p_i$ *holds for all* $i \geq 1$.

A sequence $(\alpha, \alpha\ \mathsf{list}), (\alpha, \alpha\ \mathsf{list}), \ldots$ is an example of an ascending sequence. A most general solution $(\rho_i)_{i\leq n}$ to a prefix of length $n + 1$ of this sequence is defined as $\rho_i = \alpha \mapsto \alpha\ \mathsf{list}^n$ for all $i \leq n$.

Lemma 5.15. *Let* $\rightsquigarrow$ *be monotone,* $(p_0, q_0), (p_1, q_1), \ldots$ *be an ascending sequence and* $p_i \rightsquigarrow q_i$ *for all i. Then for all n it holds that there exists the most general solution* $(\rho_i)_{i\leq n}$ *to* $(p_i, q_i)_{i\leq n}$ *such that* $(p_i, q_i, \rho_i)_{i\leq n}^{\rightsquigarrow}$.

Proof. We fix n. Let $(\rho_i)_{i\leq n}$ be the most general solution to $(p_i, q_i)_{i\leq n}$. $(\rho_i)_{i\leq n}$ always exists because of $(p_i, q_i)_{i\leq n} \preceq p_{n+1}$.

We prove by backward induction on n that $(\rho_j)_{j\leq i}$ is the most general solution to $(p_j, q_j)_{j\leq i}$ for all $i \leq n$. Base case: trivial. Inductive step ($n = i$, $i > 0$): $(\rho_j)_{j\leq i}$ is the most general solution to $(p_j, q_j)_{j\leq i}$. From $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \preceq p_i$ it follows that there exists $(\rho'_j)_{j\leq i-1}$, a most general solution to $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1})$ such that $\rho'_{i-1}(q_{i-1}) \leq_{\rho'} p_i$. By Lemma 5.9 we know that $(\rho'_j)_{j\leq i-1}, \rho'$ is the most general solution to $(p_j, q_j)_{j\leq i}$ but then Lemma 5.4 gives us $\rho_j \approx_{p_j} \rho'_j$ for all $j \leq i - 1$. Hence $(\rho_j)_{j\leq i-1}$ is the most general solution to $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1})$.

Now we can see that $\rho_0 \approx_{p_0} \mathsf{id}$ since ρ_0 is the most general solution to (p_0, q_0). □

An ascending sequence is a formalization of the notion that we mentioned before, i.e., a sequence whose each prefix is a path. Lemma 5.11 tells us that in a sequence given by a most general solution, there always exists a suffix that is a path. But this lemma does not say much about the length of such a suffix. Let us inspect the proof of Lemma 5.11: there are two cases in which we extend the most general solution. In the first case, we extend also the suffix that is a path. But in the second case ($\rho'_i(q_i) \geq_{\rho'} p_{i+1}$), the path gets reset to a sequence of length one. The following lemma shows that the second case can happen only finitely many times and thus we can always find an ascending sequence.

Lemma 5.16 (The Key Technical Lemma)**.** *Assume that* $\rightsquigarrow$ *is composable and monotone, and that* $(p_i)_{i\in\mathbb{N}}$, $(q_i)_{i\in\mathbb{N}}$ *and* $(\rho_i)_{i\in\mathbb{N}}$ *are sequences such that* $p_i \rightsquigarrow q_i$ *and* $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ *for all i. Then there exists k such that the sequence* $(p_k, q_k), (p_{k+1}, q_{k+1}), \ldots$ *is ascending.*

Proof. First informally: When we construct a most general solution to a prefix of $(p_i, q_i)_{i\leq\mathbb{N}}$, each extension done by Lemma 5.10 means that we apply a substitution $\hat{\rho}$ to the first element of the sequence. This means that the size of the first element increases (because of monotonicity). But the size of the first element cannot increase ad infinitum for this reason: it must hold $(\hat{\rho_1} \circ \ldots \circ \hat{\rho_k})(p_0) \geq \rho_0(p_0)$ since a prefix of $(\rho_i)_{i\leq\mathbb{N}}$ is a solution to a corresponding prefix of $(p_i, q_i)_{i\leq\mathbb{N}}$ and $\hat{\rho_1} \circ \ldots \circ \hat{\rho_k}$ is from a most general solution.

We do the proof by contradiction. No such k exists, which means there exists an infinite sequence $(i_j)_{j\in\mathbb{N}}$ such that $i_0 = 0$ and $(p_0, q_0), \ldots, (p_{i_j-1}, q_{i_j-1}) \not\preceq p_{i_j}$ for all $j > 0$ and $(i_j)_{j\in\mathbb{N}}$ iterates all these cases (maximality). By Corollary 5.13 we get that $(p_0, q_0), \ldots, (p_{i_j-1}, q_{i_j-1}) \succeq p_{i_j}$. Let $(\rho^j_i)_{i\leq j}$ denote the most general solution to $(p_0, q_0), \ldots, (p_j, q_j)$, which always exists by Lemma 5.11. Let Q_j denote $\rho^{i_j-1}_{i_j-1}(q_{i_j-1})$. And finally let $\rho'_0 = \mathsf{id}$ and ρ'_j denote the substitution such that $Q_j \geq_{\rho'_j} p_{i_j}$ for all $j > 0$.

Now we prove by induction on j that $\rho^i_0 \approx_{p_0} \hat{\rho}'_j \circ \cdots \circ \hat{\rho}'_0$ for all $i_j \leq i < i_{j+1}$ and for some $\hat{\rho}'_l$ such that $\hat{\rho}'_l \approx_{Q_l} \rho'_l$ for all $l \leq j$. Base case $j = 0$: from the maximality of $(i_j)_{j\in\mathbb{N}}$ it follows that i_1 is the first index i when $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \succeq p_i$. Therefore $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \preceq p_i$ for all $i < i_1$ and therefore $\rho^i_0 \approx_{p_0} \mathsf{id} = \rho'_0$ for all $i < i_1$ (formally by induction and Lemmas 5.4 and 5.9). Inductive step $j = l$ and $l > 0$: $\rho^i_0 \approx_{p_0} \hat{\rho}'_{l-1} \circ \cdots \circ \hat{\rho}'_0$ for all $i_{l-1} \leq i < i_l$. Thus $(p_0, q_0), \ldots, (p_{i_l-1}, q_{i_l-1}) \succeq p_{i_l}$ holds and by Lemmas 5.4 and 5.10 we obtain $\rho^{i_l}_0 \approx_{p_0} \hat{\rho}'_l \circ \cdots \circ \hat{\rho}'_0$ such that $\hat{\rho}'_l \approx_{Q_l} \rho'_l$. Because $(i_j)_{j\in\mathbb{N}}$ is maximal, $(p_0, q_0), \ldots, (p_{i-1}, q_{i-1}) \preceq p_i$ for all $i_l < i < i_{l+1}$ and $\rho^i_0 \approx_{p_0} \hat{\rho}'_l \circ \cdots \circ \hat{\rho}'_0$ (again formally by induction and Lemmas 5.4 and 5.9).

We reason for each $j > 0$: $(p_0, q_0), \ldots, (p_{i_j-1}, q_{i_j-1}) \not\preceq p_{i_j}$ and $(p_0, q_0), \ldots, (p_{i_j-1}, q_{i_j-1}) \succeq p_{i_j}$ gives us that $\rho'_j \not\approx_{Q_j} \mathsf{id}$. By Lemma 5.2 we get that $\mathsf{FV}(Q_j) \subseteq \mathsf{FV}(\rho_0^{i_j-1}(p_0))$. Using this, the fact that $\hat{\rho}'_j \approx_{Q_j} \rho'_j \not\approx_{Q_j} \mathsf{id}$ and Lemma 3.1 we obtain $\mathsf{size}(\rho_0^{i_j}(p_0)) > \mathsf{size}(\rho_0^{i_j-1}(p_0))$ and finally again from Lemma 3.1

and $\rho_0^{i_j-1} \approx_{p_0} \rho_0^{i_{j-1}}$, it follows that

$$\mathsf{size}(\rho_0^{i_j}(p_0)) > \mathsf{size}(\rho_0^{i_{j-1}}(p_0)). \tag{8}$$

But since each $(\rho_i^{i_j})_{i\le i_j}$ is the most general solution to a prefix of $(p_0,q_0),\dots,$ there exists η_j for each $(\rho_i^{i_j})_{i\le i_j}$ such that $\rho_0(p_0) = \eta_j(\rho_0^{i_j}(p_0))$. Since $\mathsf{size}(\rho_0(p_0))$ is already fixed and (8) tells us that $\mathsf{size}(\rho_0^{i_j}(p_0))$ is an increasing sequence (in j), there must be j' such that $\mathsf{size}(\rho_0^{i_{j'}}(p_0)) > \mathsf{size}(\rho_0(p_0))$, which prevents an existence of $\eta^{j'}$ and is a contradiction with the fact that $(\rho_i)_{i\le i_{j'}}$ is a solution to $(p_0,q_0),\dots,(p_{i_{j'}},q_{i_{j'}})$. □

Please note that composability is crucial for the last lemma. Consider this example: the dependency relation $\rightsquigarrow$ is defined as $(\alpha \times \mathsf{nat})\ \mathsf{list} \rightsquigarrow (\mathsf{nat}\times\alpha)\ \mathsf{list}$ and we have a non-terminating sequence $((\alpha\times\mathsf{nat})\ \mathsf{list}, (\mathsf{nat}\times\alpha)\ \mathsf{list}), ((\alpha\times\mathsf{nat})\ \mathsf{list}, (\mathsf{nat}\times\alpha)\ \mathsf{list}),\dots$. There is a most general solution $(\rho_i)_{i\le n}$ to any prefix of this sequence defined as $\rho_i = \alpha \mapsto \mathsf{nat}$ for all $i \le n$. But there is no ascending sequence since a prefix of an ascending sequence must be a path. But $\rightsquigarrow$ is not composable since the only path is $((\alpha\times\mathsf{nat})\ \mathsf{list}, (\mathsf{nat}\times\alpha)\ \mathsf{list}, \mathsf{id})^{\rightsquigarrow}$ and $(\alpha\times\mathsf{nat})\ \mathsf{list} \downarrow (\mathsf{nat}\times\alpha)\ \mathsf{list}$.

We can look at the last lemma also from a different perspective: if there exists a non-terminating sequence, we can find p and q such that $p \rightsquigarrow q$ and such that there exists a non-terminating sequence starting from p. We have to consider only finitely many such p's and q's since $\rightsquigarrow$ is finite, so there is no need to consider the infinitely many possible instantiations.

An ascending sequence is a key ingredient that allows us to prove the main result of this section because an ascending sequence always gives rise to a cycle.

Lemma 5.17. *Let us assume that $\rightsquigarrow$ is finite, monotone and composable, then the following statements are equivalent:*

1. $\rightsquigarrow\!\!\!\!\rightsquigarrow$ *is non-terminating*
2. $\rightsquigarrow$ *is cyclic*

Proof. 2. $\Rightarrow$ 1. There exists a path $(p_i,q_i,\rho_i)_{i\le n}^{\rightsquigarrow}$. Since $\rightsquigarrow\!\!\!\!\rightsquigarrow$ is the transitive and substitutive closure of $\rightsquigarrow$, it must hold that $\rho_i(p_i) \rightsquigarrow\!\!\!\!\rightsquigarrow \rho_i(q_i)$ for all $i \le n$ and $\rho_i(q_i) = \rho_{i+1}(p_{i+1})$ for all $i < n$. Thus $p_0 \rightsquigarrow\!\!\!\!\rightsquigarrow q_0 \rightsquigarrow\!\!\!\!\rightsquigarrow \rho_1(q_1) \rightsquigarrow\!\!\!\!\rightsquigarrow \cdots \rightsquigarrow\!\!\!\!\rightsquigarrow \rho_{n-1}(q_{n-1}) \rightsquigarrow\!\!\!\!\rightsquigarrow \rho_n(q_n)$. Since $\rho_n(q_n) \le_\rho p_0$, we have $p_0 \rightsquigarrow\!\!\!\!\rightsquigarrow \rho(p_0)$ and thus we can conclude $p_0 \rightsquigarrow\!\!\!\!\rightsquigarrow \rho(p_0) \rightsquigarrow\!\!\!\!\rightsquigarrow \rho(\rho(p_0)) \rightsquigarrow\!\!\!\!\rightsquigarrow \rho(\rho(\rho(p_0))) \rightsquigarrow\!\!\!\!\rightsquigarrow \cdots$.

1. $\Rightarrow$ 2. If $\rightsquigarrow\!\!\!\!\rightsquigarrow$ does not terminate, there exists a non-terminating sequence. We unfold transitive edges in this sequence and work only with $\rightsquigarrow^{\downarrow}$. Thus there exist sequences $(p'_i)_{i\in\mathbb{N}}$ and $(q'_i)_{i\in\mathbb{N}}$ such that $p'_i \rightsquigarrow^{\downarrow} q'_i$ and $q'_i = p'_{i+1}$ for all i. The non-terminating sequence is then $p'_0 \rightsquigarrow^{\downarrow} p'_1 \rightsquigarrow^{\downarrow} p'_2 \rightsquigarrow^{\downarrow} \cdots$. From the definition of $\rightsquigarrow^{\downarrow}$ follows that there exist sequences $(p_i)_{i\in\mathbb{N}}$, $(q_i)_{i\in\mathbb{N}}$ and $(\rho_i)_{i\in\mathbb{N}}$ such that $p_i \rightsquigarrow q_i$ and $\rho_i(p_i) = p'_i$ and $\rho_i(q_i) = q'_i$ and since $q'_i = p'_{i+1}, \rho_i(q_i) = \rho_{i+1}(p_{i+1})$ holds for all i.

The key step of the proof is to find an ascending sequence $(p_k,q_k),(p_{k+1},q_{k+1}),\dots$ such that there exists also $k' > k$ and $(p_k,q_k) = (p_{k'},q_{k'})$: We define $\Theta_k = \{(p_l,q_l).\ l \ge k\}$. Since $\rightsquigarrow$ is a finite relation, Θ_k is a finite set for all k (recall $p_i \rightsquigarrow q_i$) and is never empty. We use Lemma 5.16 to obtain k_1 such that the sequence $(p_{k_1},q_{k_1}),(p_{k_1+1},q_{k_1+1}),\dots$ is ascending. If there does not exist a $k'_1 > k_1$ such that $(p_{k_1},q_{k_1}) = (p_{k'_1},q_{k'_1})$, we use Lemma 5.16 for subsequences $(p_i)_{i>k_1}$, $(q_i)_{i>k_1}$ and obtain the respective k_2. Because $\Theta_{k_1} \supsetneq \Theta_{k_2}$, the whole process must stop after at most $|\Theta_0|$ steps by finding k and $k' > k$ such that $(p_k,q_k),(p_{k+1},q_{k+1}),\dots$ is an ascending sequence and $(p_k,q_k) = (p_{k'},q_{k'})$.

We define sequences $(r_i)_{i\le n}$ and $(s_i)_{i\le n}$ as following: $n = k' - k - 1$, $r_i = p_{k+i}$ and $s_i = q_{k+i}$ for all $i \le n$. Since $(p_k,q_k),(p_{k+1},q_{k+1}),\dots$ is an ascending sequence, we get the most general solution $(\rho_i)_{i\le n}$ to $(r_i,s_i)_{i\le n}$ such that $(r_i,s_i,\rho_i)_{i\le n}^{\rightsquigarrow}$ (by Lemma 5.15). From $(p_k,q_k) = (p_{k'},q_{k'})$, it follows that $(r_0,s_0),\dots,(r_n,s_n) \preceq r_0$ and finally we get $\rho_n(s_n) \le r_0$ by Lemmas 3.3 and 5.4. This concludes the proof that $\rightsquigarrow$ is cyclic. □

6. From Cyclicity to a Decision Procedure

Since acyclicity of a finite $\rightsquigarrow$ is a finite problem, there should be a decision procedure for this problem. We introduce this procedure in Algorithm 1 and prove it correct.

Definition 6.1. *We say that $\rightsquigarrow$ is* orthogonal *if for all p,q,p' and q' such that $(p,q) \neq (p',q')$ it holds that if $p \rightsquigarrow q$ and $p' \rightsquigarrow q'$ then $p \ \#\ p'$.*

Orthogonality is another restriction on $\rightsquigarrow$. As we prove in the following lemma, this constraint guarantees that if two paths start from the same value p_0, then these paths are the same modulo equivalent substitutions. We need this property to restrict once more the search space of our algorithm.

Lemma 6.2. *Let $\rightsquigarrow$ be orthogonal and monotone. If there exist two paths $(p_i,q_i,\rho_i)_{i\le n}^{\rightsquigarrow}$ and $(p'_i,q'_i,\rho'_i)_{i\le n}^{\rightsquigarrow}$ such that $p_0 = p'_0$, then $p_i = p'_i$, $q_i = q'_i$ and $\rho_i \approx_{p_i} \rho'_i$ for all $i \le n$.*

Proof. By contradiction: Let $k \le n$ be the smallest k such that:

- Either $p_k \neq p'_k$: $k > 0$, $\rho_{k-1} \approx_{p_{k-1}} \rho'_{k-1}, \rho_{k-1}(q_{k-1}) \le_{\rho_k} p_k$ and $\rho'_{k-1}(q_{k-1}) \le_{\rho'_k} p'_k$. But this contradicts $p_k \ \#\ p'_k$ (by Lemma 3.3).
- Or $q_k \neq q'_k$: but $p_k = p'_k$, which contradicts $p_k \ \#\ p'_k$.
- Or $\rho_k \not\approx_{p_k} \rho'_k$: a) $k = 0$: contradiction with $\rho_0 \approx_{p_0} \mathsf{id}$ and $\rho'_0 \approx_{p_0} \mathsf{id}$. b) $k > 0$: $\rho_{k-1} \approx_{p_{k-1}} \rho'_{k-1}, \rho_{k-1}(q_{k-1}) \le_{\rho_k} p_k$ and $\rho'_{k-1}(q_{k-1}) \le_{\rho'_k} p_k$. By Lemma 3.3 we get a contradiction with uniqueness of $\le_\rho$, i.e., if $p \le_\rho q$ and $p \le_{\rho'} q$, then $\rho =_q \rho'$.

□

We assume that we have a unification algorithm on types, which we again extend to $\mathcal{U}_\Sigma$. This algorithm is used internally in function HASCOMMONINSTANCE in Algorithm 1. Function HASCOMMONINSTANCE is used to test composability and orthogonality.

Lemma 6.3. $\neg$HASCOMMONINSTANCE(p,q) *iff* $p \ \#\ q$.

Proof. p and q have a common instance iff there exists p' such that $p' \le_\rho p$ and $p' \le_{\rho'} q$. Since ρ and ρ' can be different, we have to rename variables in q in the procedure. The unification then decides if there exists a common instance. □

Let us somewhat informally describe our cyclicity decision procedure presented in Algorithm 1. First of all, a syntactic comment: our algorithm contains ghost variables, which help us analyze the algorithm but are not used during the real computation. Commands where the ghost variables are used are always prefixed by character #. We use two ghost variables i and R_i, whose meaning we will explain later.

The core of the computation of the program happens in the main loop, i.e., between lines 48 and 58. We call the loop *reduction phase*. The value of i is an iteration counter of the reduction phase.

The variable dep is the only input of our algorithm and does not get changed during the whole computation. This is the relation that we are supposed to check whether it contains a cycle or not. In order to do it, we start discovering for each p from dep[6] a path starting from p and we store the beginning and end of such a path

[6] more precisely, for each p such that there exists q and $(p,q) \in dep$

in dep_+. That is to say, if $(p,q) \in dep_+$, this means we have discovered a path from p to q. This path is a candidate for a cycle and therefore after each iteration we check if $q \leq p$ in the function IsAcyclic. Moreover, we store in dep_+ only the longest path from p that we have discovered so far. The function ReduceStep extends all paths from dep_+ by one step if it is possible. If not, p is marked as final. Thanks to Lemma 6.2, there always exists a unique extension of dep_+ modulo renaming. That is to say, when we look for (p',q') on line 24, there is at most one such a pair.

We store only the beginning and end of a path starting from p in dep_+, because this is enough information for the algorithm to check if this path comprises a cycle. But for the analysis of the algorithm, we want to know also intermediate steps of such a path and therefore we store these steps in ghost variable $R_i(p)$. The two following lemmas show that $R_i(p)$ gets defined in the ith iteration if p is not final and that $R_i(p)$ is a path starting from p of length $i+1$ and that the algorithm does not miss any path. If dep_+ does not change after ReduceStep, it means no further cycle candidate exists and we can report that dep is acyclic.

Lemma 6.4. *Let $p_0 \rightsquigarrow q$. Then p_0 is not final in the nth iteration of* Check($\rightsquigarrow$) *iff $R_n(p_0)$ gets defined in the nth iteration of* Check($\rightsquigarrow$). *Moreover, if $R_n(p_0)$ is defined, there exists a path $(p_i,q_i,\rho_i)_{i\leq n}^{\rightsquigarrow}$ such that $R_n(p_0) = (p_i,q_i,\rho_i)_{i\leq n}$.*

Proof. $R_n(p_0)$ can get defined only in the nth iteration of the algorithm, since the ghost variable i is strictly increasing. Each p is not final at the beginning of computation and when it gets final, it stays final for the rest of computation. Then clearly p_0 gets final in the nth iteration iff $R_n(p_0)$ does not get defined in the nth iteration.

We prove that defined $R_n(p_0)$ comprises a path by induction on n. Base case ($n = 0$): $R_0(p_0)$ is defined at the beginning of the algorithm and defines a trivial path. Inductive step ($n = i+1$): p_0 does not get final in the $(i+1)$st step of the algorithm and therefore it was not final in the ith step either. We take the sequence $R_i(p_0) = (p_j,q_j,\rho_j)_{j\leq i}$ defined in the ith step and we know that $(p_j,q_j,\rho_j)_{j\leq i}^{\rightsquigarrow}$. We take q that is considered on line 23 in the $(i+1)$st step such that $(p_0,q) \in dep_+$. It is clear that $q = \rho_i(q_i)$ and there must ρ_{i+1} and p_{i+1} such that $\rho_i(q_i) \leq_{\rho_{i+1}} p_{i+1}$, otherwise p_0 would get final. Then $R_{i+1}(p_0)$ gets defined in the $(i+1)$st step to $(p_j,q_j,\rho_j)_{j\leq i+1}$ and $(p_j,q_j,\rho_j)_{j\leq i+1}^{\rightsquigarrow}$. □

In the light of the previous lemma we will write $R_n(p) = (p_i,q_i,\rho_i)_{i\leq n}^{\rightsquigarrow}$ from now on.

Lemma 6.5. *Let $\rightsquigarrow$ be orthogonal and monotone. If there exists a path $(p_i,q_i,\rho_i)_{i\leq n}^{\rightsquigarrow}$, then $R_n(p_0)$ gets defined in the nth iteration of* Check($\rightsquigarrow$).

Proof. We do the proof by induction on n. Base case ($n = 0$): $R_0(p_0)$ is defined at the beginning of the algorithm. Inductive step $n = i+1$: Let $(p_j,q_j,\rho_j)_{j\leq i+1}^{\rightsquigarrow}$, then $(p_j,q_j,\rho_j)_{j\leq i}^{\rightsquigarrow}$ is also a path and we can use the induction hypothesis and obtain by Lemma 6.4 the path $R_i(p_0) = (p'_j,q'_j,\rho'_j)_{j\leq i}^{\rightsquigarrow}$ defined in the ith step. We take q that is considered on line 23 in the $(i+1)$st step such that $(p_0,q) \in dep_+$. It is clear that $q = \rho'_i(q'_i)$. By Lemma 6.2 we get that $p_j = p'_j$, $q_j = q'_j$ and $\rho_j \approx_{p_j} \rho'_j$ for all $j \leq i$ and from the definition of $(p_j,q_j,\rho_j)_{j\leq i+1}^{\rightsquigarrow}$ that $\rho_i(q_i) \leq_{\rho_{i+1}} p_{i+1}$. By Lemma 3.3 we can finally derive that $\rho'_i(q'_i) \leq p_{i+1}$ and therefore p_0 cannot get final in the $(i+1)$st step and $R_{i+1}(p_0)$ gets defined in this step (by Lemma 6.4). □

Algorithm 1 can either return **success** (we write Check($\rightsquigarrow$) = **success**) or **fail** (we write Check($\rightsquigarrow$) = **fail**) or not terminate (we write Check($\rightsquigarrow$) = $\uparrow$).

As we know from Section 5, we reduce termination to cyclicity only under some assumptions, where the most important one is

Algorithm 1 The main algorithm

```
 1: function HasCommonInstance(p, q)
 2:     q' ← rename q apart from p
 3:     return p and q' can be unified
 4: end function
 5:
 6: function IsOrthogonal(dep)
 7:     return ∀(p,q),(p',q') ∈ dep. if (p,q) ≠ (p',q') then
            ¬HasCommonInstance(p, p')
 8: end function
 9:
10: function IsMonotone(dep)
11:     return ∀(p,q) ∈ dep. FV(q) ⊆ FV(p)
12: end function
13:
14: function IsAcyclic(dep+)
15:     return ∀(p,q) ∈ dep+. q ≰ p
16: end function
17:
18: function IsComposable(q, dep)
19:     return ∀(p',q') ∈ dep.
            HasCommonInstance(q, p') ⇒ q ≥ p'
20: end function
21:
22: function ReduceStep(dep, dep+)
23:     for all (p,q) ∈ dep+ such that final(p) = false do
24:         if can find (p',q') ∈ dep such that q ≤ρ p' then
25:             dep+ ← dep+ \ (p,q) ∪ (p, ρ(q'))
26:             #Ri(p) ← Ri−1(p), (p',q',ρ)
27:         else
28:             final(p) ← true
29:             if ¬IsComposable(q, dep) then
30:                 return fail
31:             end if
32:         end if
33:     end for
34:     return dep+
35: end function
36:
37: function Check(dep)
38:     #i = 0
39:     for all (p,q) ∈ dep. final(p) ← false
40:     #for all (p,q) ∈ dep. R0(p) ← (p,q,id)
41:     if ¬IsOrthogonal(dep) then
42:         return fail
43:     end if
44:     if ¬IsMonotone(dep) then
45:         return fail
46:     end if
47:     dep+ ← dep
48:     loop
49:         #i ← i + 1
50:         dep'+ ← ReduceStep(dep, dep+)
51:         if dep+ = dep'+ then
52:             exit loop
53:         end if
54:         dep+ ← dep'+
55:         if ¬IsAcyclic(dep+) then
56:             return fail
57:         end if
58:     end loop
59:     return success
60: end function
```

composability. Our algorithm also checks composability of dep, which is done during the reduction phase for two reasons: 1) It is too late to do it after the phase because when the composability does not hold, the reduction phase may fail to terminate. 2) We cannot do it before because composability must be checked for all possible paths and we have to be sure that there exists no infinite path by checking also acyclicity dynamically, which is the goal of the reduction phase.

In order to make our algorithm more efficient, we check composability only for paths that start at p that is final. The next lemma shows that this suffices. The key observation is that if we can extend a path, the composability still locally holds.

Lemma 6.6. *If* CHECK$(\rightsquigarrow) \neq$ ***fail****, then $\rightsquigarrow$ is monotone, composable, orthogonal and acyclic.*

Proof. $\rightsquigarrow$ is monotone, since this property is checked directly at the beginning of the algorithm. The same holds for orthogonality by using Lemma 6.3.

We prove the composability by contradiction: There exist a path $(p_i, q_i, \rho_i)_{i\leq n}^{\rightsquigarrow}$ and p and q such that $p \rightsquigarrow q$ and $\rho_n(q_n) \downarrow p$. Then from Lemmas 6.4 and 6.5 it follows that $R_n(p_0) = (p_i', q_i', \rho_i')_{i\leq n}^{\rightsquigarrow}$ was defined in the nth iteration of CHECK$(\rightsquigarrow)$ and by Lemma 6.2 that $p_i = p_i'$, $q_i = q_i'$ and $\rho_i \approx_{p_i} \rho_i'$ for all $i \leq n$. Then $\rho_n'(q_n') \downarrow p$ follows from Lemma 3.3. In the $(n+1)$st iteration two cases can occur:

- There does not exist any p' and q' such that $p' \rightsquigarrow q'$ and $\rho_n'(q_n') \leq p'$. Then ISCOMPOSABLE$(\rho_n'(q_n'), \rightsquigarrow)$ is executed. This function checks that $\rho_n'(q_n') \geq p'$ or $\rho_n'(q_n') \# p'$ (by Lemma 6.3) for all (p', q') such that $p' \rightsquigarrow q'$. This is a contradiction with $\rho_n'(q_n') \downarrow p$.
- There exist p' and q' such that $p' \rightsquigarrow q'$ and $\rho_n'(q_n') \leq p'$. But then for all $p'' \neq p'$ and q'' such that $p'' \rightsquigarrow q''$, $\rho_n'(q_n') \# p''$ holds (otherwise $\rightsquigarrow$ would not be orthogonal). But this is again a contradiction with $\rho_n'(q_n') \downarrow p$.

We prove the acyclicity by contradiction as well: If $\rightsquigarrow$ is cyclic, there exists a path $(p_i, q_i, \rho_i)_{i\leq n}^{\rightsquigarrow}$ such that $\rho_n(q_n) \leq p_0$. By Lemmas 6.4 and 6.5 we obtain another path $R_n(p_0) = (p_i', q_i', \rho_i')_{i\leq n}^{\rightsquigarrow}$ defined in the nth step of the algorithm and by Lemmas 3.3 and 6.2 finally $\rho_n'(q_n') \leq p_0'$. But this means that the cyclicity check on line 55 returns **fail**.

□

Let $P(\rightsquigarrow)$ abbreviate the conjunction of the following properties:

- $\rightsquigarrow$ is monotone,
- $\rightsquigarrow$ is composable,
- $\rightsquigarrow$ is orthogonal,
- $\rightsquigarrow$ is acyclic.

Lemma 6.7. *If $\rightsquigarrow$ is finite then the following holds:*

- CHECK$(\rightsquigarrow)$ *always terminates,*
- CHECK$(\rightsquigarrow)$ = ***success*** *if and only if $P(\rightsquigarrow)$.*

Proof. We start by proving termination by contradiction. The only way how CHECK$(\rightsquigarrow)$ can fail to terminate is when the program never exits the reduction phase. This happens when the check on line 51 is always false,[7] i.e., when the function REDUCESTEP always changes dep_+, which means we can always find $(p', q') \in dep$ on line 24 such that $q \leq_\rho p'$. That means there exists an infinite sequence $R_0(p), R_1(p), \ldots$ for a certain p. From this sequence, we can easily construct an infinite sequence $\rho_0(p_0) \rightsquigarrow\!\!\!\!\!\rightarrow \rho_1(p_1) \rightsquigarrow\!\!\!\!\!\rightarrow \rho_2(p_2) \rightsquigarrow\!\!\!\!\!\rightarrow \cdots$. Since CHECK$(\rightsquigarrow)$ does not return **fail**, $\rightsquigarrow$ is monotone, composable, orthogonal and acyclic by Lemma 6.6 and thus we can invoke Lemma 5.17 and get that $\rightsquigarrow$ is cyclic, which is a contradiction.

Now we continue by the proof of the equivalence. Left-to-right: By Lemma 6.6. Right-to-left: Since we proved that the algorithm terminates, we know that CHECK$(\rightsquigarrow) \neq$ **fail** implies CHECK$(\rightsquigarrow)$ = **success** and therefore it suffices to prove that CHECK$(\rightsquigarrow) \neq$ **fail**. The algorithms can return fail only on lines 42, 45, 30 and 56. We prove by contradiction that the program cannot return **fail** on any of those lines:

- If **fail** is returned on line 42, it means that $\rightsquigarrow$ is not orthogonal (by Lemma 6.3).
- If **fail** is returned on line 45, it means that $\rightsquigarrow$ is not monotone.
- If **fail** is returned on line 30 in the nth iteration, there exist p and a path $R_{n-1}(p) = (p_i, q_i, \rho_i)_{n-1}^{\rightsquigarrow}$ by Lemma 6.4 such that for all p' and q' such that $p' \rightsquigarrow q'$, we have $\rho_{n-1}(q_{n-1}) \not\leq p'$. Moreover, it has to hold that $\neg$ISCOMPOSABLE$(\rho_{n-1}(q_{n-1}), \rightsquigarrow)$ and therefore there exist p' and q' such that $p' \rightsquigarrow q'$ and $\neg(\rho_{n-1}(q_{n-1}) \# p')$ and $\rho_{n-1}(q_{n-1}) \not\geq p'$. But this means $\rho_{n-1}(q_{n-1}) \downarrow p'$ and therefore $\rightsquigarrow$ is not composable.
- If **fail** is returned on line 56 in the nth iteration, there exists p_0 that is not final[8] and a path $R_n(p_0) = (p_i, q_i, \rho_i)_n^{\rightsquigarrow}$ (by Lemma 6.4) such that $\rho_n(q_n) \leq p_0$. But this means that $\rightsquigarrow$ is cyclic.

□

Theorem 1. *There exists a predicate P on binary relations on $\mathcal{U}_\Sigma$ such that for finite relations $\rightsquigarrow$ the following holds:*

- *$P(\rightsquigarrow)$ is decidable*
- *$P(\rightsquigarrow)$ implies that $\rightsquigarrow\!\!\!\!\!\rightarrow$ terminates*
- *P contains interesting relations $\rightsquigarrow$*

Proof. Lemma 6.7 shows that P is decidable by the program CHECK. Lemma 5.17 shows that $P(\rightsquigarrow)$ implies termination of $\rightsquigarrow\!\!\!\!\!\rightarrow$.

Now we proceed to the last question: do the restriction to relations $\rightsquigarrow$ that are monotone, composable, and orthogonal still allows for suitable expressiveness for overloaded definitions? Monotonicity and orthogonality are such natural conditions that one would expect that any reasonable (overloaded) definition must fulfill them. We will argue now that composability still admits all main use cases of overloading in Isabelle:

- In the context of type classes, only what Haftmann and Wenzel [3] call *restricted* overloading is allowed: constants can be declared only with a linear polymorphic type, e.g., $c_{\alpha\,\tau}$. Overloaded definitions have this form $c_{(\overline{\alpha}\,k)\,\tau} = \ldots c_{\alpha_i\,\tau} \ldots$, i.e., if c appears on the right hand side, it uses some α_i from $\overline{\alpha}$. Such definitions generate only composable dependency relations.
- Our experience shows that all cases of unrestricted overloading that have been required by users so far also fulfill composability. A classical example would be a basic concept of nth power of composition of f_α (written f^n) defined in Isabelle/HOL. This operation is then overloaded for unary functions ($f_{\alpha\to\alpha}$), binary relations ($f_{\alpha\to\alpha\to\mathsf{bool}}$) and relations as sets ($f_{(\alpha\times\alpha)\ \mathsf{set}}$).

□

[7] and also the check on line 55 is always false

[8] Otherwise the cycle would have been detected in the previous iteration.

7. Issues with the Original Algorithm

During inspection of the original cyclicity checker and during the subsequent formalization, we identified three issues:

- Completeness issue: The original algorithm (Isabelle2014) does not always terminate because the cyclicity check (function IsAcyclic) misses some cycles and therefore the reduction phase might loop. Leaving out details, consider this minimal example:

  $$a_{\alpha\,\mathsf{list}\times\beta} \rightsquigarrow a_{\alpha\,\mathsf{list}\times\alpha}.$$

  The algorithm concludes that $\rightsquigarrow$ must be acyclic since α in $a_{\alpha\,\mathsf{list}\times\alpha}$ is also contained in $a_{\alpha\,\mathsf{list}\times\beta}$. But $\rightsquigarrow$ is obviously cyclic. We use instead solely an instance test ($q \not\leq p$) in our modified algorithm.

- Completeness issue: In the original algorithm (Isabelle2014), composability is checked[9] at the end of the algorithm after the reduction phase is finished. But if the composability does not hold, the reduction phase may fail to terminate, as in the following example:

  $$a_{\mathsf{int}} \rightsquigarrow b_{\mathsf{int}\times\mathsf{nat}}, b_{\alpha\times\mathsf{nat}} \rightsquigarrow c_{\alpha\times\mathsf{nat}}, c_{\mathsf{int}\times\alpha} \rightsquigarrow b_{\mathsf{int}\times\alpha}$$

  $\rightsquigarrow$ is not composable because $c_{\alpha\times\mathsf{nat}} \downarrow c_{\mathsf{int}\times\alpha}$. But this is never detected in the original algorithm for this reason: starting from a_{int} the reduction phase does not terminate since no cycle is detected ($c_? \not\leq a_{\mathsf{int}}$ and $b_? \not\leq a_{\mathsf{int}}$).

  We modified the algorithm such that composability is checked during the reduction phase. But the change is subtler than just moving the original test into the reduction phase because then the complexity increases from $\mathcal{O}(|dep|^2)$ to $\mathcal{O}(|dep|^3)$. Therefore we test composability only for p's that are final, i.e., for which the reduction phase terminates. This suffices by Lemma 6.6 and preserves quadratic complexity.

- Correctness issue: Our colleague Andrei Popescu found the following soundness issue caused by the cyclicity checker:

  consts c :: 'a ⇒ bool
  consts d :: ('a × 'b) ⇒ bool

  defs *c_def*: c (*x*::'a) ≡ d (undefined:: ('a × 'a))
  defs *d_def*: d (*x*::('a × nat)) ≡ ¬c (undefined::'a)

  This input is accepted by Isabelle2013-2 and leads to an inconsistency since the following can be proved:

  $$\mathsf{c}(\mathsf{undefined}::\mathsf{nat}) = \neg\mathsf{c}(\mathsf{undefined}::\mathsf{nat})$$

  The derived dependency relation is as follows:

  $$c_{\alpha\to\mathsf{bool}} \rightsquigarrow d_{(\alpha\times\alpha)\to\mathsf{bool}}, d_{(\alpha\times\mathsf{nat})\to\mathsf{bool}} \rightsquigarrow c_{\alpha\to\mathsf{bool}}$$

  $\rightsquigarrow$ is not composable since $d_{(\alpha\times\alpha)\to\mathsf{bool}} \downarrow d_{(\alpha\times\mathsf{nat})\to\mathsf{bool}}$. There exists a cycle (substitute nat for α) but this cycle is not detected by the cyclicity check and the reduction phase terminates. The issue is that the composability check (function IsComposable) in the original algorithm was implemented as follows:

  $$\forall(p', q') \in dep.\ \textsc{HasCommonInstance}(q, p') \Rightarrow$$
  type of q has the same shape as $\mathsf{tyOf}(q)$

  And indeed the type of $d_{(\alpha\times\alpha)\to\mathsf{bool}}$ has the same shape as d's declared type $(\alpha \times \beta) \to \mathsf{bool}$.

  The issue was amended for Isabelle2014 release by changing the condition from having the same shape to

  $$\text{type of } q \text{ is alpha-equivalent to } \mathsf{tyOf}(q).$$

[9] By $\forall(p, q) \in dep_+$. IsComposable(q, dep).

Our work clarifies this issue in two ways:

- The proof shows that the check from Isabelle2014 is correct because it is strictly stronger than the check that we propose in this paper: if q and p' have a common instance and the type of q is alpha-equivalent to $\mathsf{tyOf}(q)$, then $q \geq p'$ by orthogonality.
- The current composability check can be generalized to $q \geq p'$, which would allow more instances of overloading to be accepted and does not require any other change of the algorithm.

We plan to address all of the described issues in the next release of Isabelle. Isabelle theory files illustrating all three issues can be found on the author's web page [1].

8. Conclusion

We have seen that the question as to when overloaded constant definitions are a safe theory extension is a subtle problem. The key property is to prove that the dependency relation generated by these definitions terminates. We inspected Isabelle's algorithm that checks the termination property and identified two sources of non-termination and a soundness issue.

We presented a modified algorithm in this paper and proved its completeness and correctness. Thus we improved trustworthiness of Isabelle. Our proof uses hardly any specific features of Isabelle and therefore we presented a general approach how to implement overloading in proof assistants.

Our future work is to formalize the theory that we presented here in Isabelle/HOL and obtain verified code by Isabelle's code generator.

Acknowledgments

I would like to thank Andrei Popescu for inspiration, support and letting me present the soundness bug. Jasmin Blanchette, Johannes Hölzl, Tobias Nipkow, Dmitriy Traytel and Makarius Wenzel read earlier versions of this paper on short notice and made helpful comments. Remarks and questions of anonymous referees led to several improvements and corrections.

References

[1] URL `http://www21.in.tum.de/~kuncar/documents/issues`.

[2] A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a Nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

[3] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2006.

[4] T. Nipkow and Z. Qian. Reduction and Unification in Lambda Calculi with Subtypes. In D. Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 66–78. Springer, 1992.

[5] S. Obua. Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In F. Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2006.

[6] M. Sozeau and N. Oury. First-Class Type Classes. In O. A. Mohamed, C. M. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[7] M. Wenzel. Type Classes and Overloading in Higher-Order Logic. In E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.