A Coalgebraic Foundation for Coinductive Union Types

M. Bonsangue[1,2], J. Rot[1,2,⋆], D. Ancona[3], F. de Boer[2,1], and J. Rutten[2,4]

[1] LIACS — Leiden University
[2] Formal Methods — Centrum Wiskunde en Informatica
[3] DIBRIS — Universita di Genova
[4] ICIS — Radboud University Nijmegen

Abstract. This paper introduces a coalgebraic foundation for coinductive types, interpreted as sets of values and extended with set theoretic union. We give a sound and complete characterization of semantic subtyping in terms of inclusion of maximal traces. Further, we provide a technique for reducing subtyping to inclusion between sets of finite traces, based on approximation. We obtain inclusion of tree languages as a sound and complete method to show semantic subtyping of recursive types with basic types, product and union, interpreted coinductively.

1 Introduction

Basically all programming languages today support recursion to manipulate inductively defined data structures such as linked lists and trees. Whereas induction deals with finite but unbounded data, its dual, coinduction, deals with possibly infinite data. The relevant distinction here concerns traditional algebraic data structures which can be fully unfolded by a recursive program, and coalgebraic data structures which can be manipulated while they unfold, even if this process may never terminate. The interest in theoretical foundations for coinductive types and reasoning techniques is rapidly growing. Practical applications of coinductive types are found in the world of functional languages with lazy evaluation. Moreover a coinductive interpretation of structural recursively defined types with record, product and union type constructors allows one to assign types to coinductive data, such as infinite and circular lists of objects in object-oriented languages [3]. Union types allow a more precise analysis than disjoint sum [6], for example to type constructs like if-then-else. Consider, for instance, the recursive type definition below.

$$x_1 \mapsto \mathsf{null} \vee < \mathsf{elm}\colon \mathsf{int},\ \mathsf{nxt}\colon x_1 > \ . \tag{1}$$

Here null and int are primitive type constants for representing the empty list and the integer values, respectively, and $< \mathsf{elm}\colon x,\ \mathsf{nxt}\colon y >$ represents the (tagged) product of the type variables x and y.

⋆ The research of this author has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

Intuitively, the type defined by (1) is a recursive type representing all finite and infinite linked lists of integer values. More formally, the type definition in (1) can be interpreted both syntactically and semantically. Syntactically, (1) can be interpreted as the set of finite and infinite closed terms over the alphabet consisting of the constants null and int, obtained by unfolding. Semantically, the set theoretic interpretation of the type definition (1) is based on a given semantic interpretation of type constructors. The usual interpretation of null and int is the set containing the empty list and the set of all integers, respectively. The product type constructor then corresponds to the Cartesian product, and the union type to set theoretic union. Recursion is interpreted by fixed points. Since the interpretations of the union and product type constructors are monotonic functions, by the Knaster-Tarski theorem we have that (1) admits both the least and the greatest fixed point, that is, the equation can be interpreted either inductively, or coinductively. The inductive interpretation yields the set of integer linked lists of finite length. Notably, cyclic and other infinite lists are not captured. In contrast, the coinductive interpretation consists of finite and infinite lists.

Moreover, the inductive interpretation of a type definition

$$x_2 \mapsto < \mathsf{elm}\text{: } \mathsf{int}, \ \mathsf{nxt}\text{: } x_2 > \tag{2}$$

is the empty set. In a setting where cyclic lists can be built (e.g., in an object-oriented program) it is unsound to give an inductive type as above to cyclic lists. In fact, in the semantic subtyping approach an empty type cannot be inhabited by any value, otherwise the system becomes unsound: any such value can have an arbitrary type, by subsumption. In order to guarantee soundness either cyclic values are banned, or cyclic values are allowed but have less precise types. For instance, an acceptable inductive type for a cyclic list would be x_1 from (1). This, however, is not very precise, since accessing the n-th element of the list in a type safe way would require n non-emptiness checks which are useless in the case of a cyclic list.

The above argument shows that we have to consider a coinductive interpretation of recursive types (yielding, for example, for x_2, the set of infinite lists), and define subtyping semantically as set inclusion of coinductive interpretations. The main challenge is to provide an equivalent syntactic interpretation of recursive type declarations, and a corresponding sound and complete method for proving subtyping. Note that such a syntactic representation cannot be inductive either, because we are dealing with infinite terms. Existing coinductive proof methods such as [3–5] are incomplete and involve complex soundness proofs.

The theory of *coalgebras* has emerged as a general framework for a transparent and uniform study of coinduction (the basics are recalled in Section 2). Our aim therefore is to develop a coalgebraic approach to coinductive types, providing a single framework for the formalization of both canonical syntactic interpretations and equivalent semantic interpretations.

To achieve this goal we first focus on the basic notion of coinductive types without union (Section 3). This allows us to derive a natural syntactic interpretation of coinductive types by final coalgebras with bisimulation as a sound and complete proof method for equivalence of coinductive types. Further, this basic class of coinductive types allows us to focus on the general development of a final coalgebra of values from which we derive, in our framework, a semantic interpretation equivalent to the syntactic one.

The main challenge for a coalgebraic formalization of *union* types is to capture the distributivity of the union constructor over the product type constructor. In the setting of coalgebras this problem is reflected in the difference between bisimilarity and trace semantics. Our solution uses a coalgebraic approach to trace semantics based on [21, 18, 10], to extend the case of types without union to a precise characterization of semantic subtyping as inclusion between *subsets* of the final coalgebra, thus incorporating union types (Section 4).

Finally, we show how to reduce subtyping to inclusion between sets of finite traces, based on *approximation* of maximal traces by finite ones (Section 5). Such a reduction does not hold for arbitrary types of systems, but we devise a general coinductive proof technique for showing that it does apply in mildly restricted settings. This technique is instantiated to Moore automata and tree automata, yielding sound and complete methods for proving subtyping.

The contributions of this paper are as follows. We provide a structural and natural coalgebraic semantics for semantic subtyping of coinductive union types, which is parametric in the type constructors and abstracts away from a specific choice of syntax. We extend the theory of coalgebraic trace semantics with a novel coinductive method for finitely approximating maximal traces. We apply this technique to give the first sound and complete method for deciding semantic subtyping of coinductively interpreted recursive types with product and union.

2 Coalgebras

For an extensive introduction to the theory of universal coalgebra see [23]. We denote by Set the category of sets and functions and by Id the identity functor. Given a functor $F\colon \mathsf{Set} \to \mathsf{Set}$, an *$F$-coalgebra* is a pair (X, c) of a set X and a function $c\colon X \to FX$. A *homomorphism* between two coalgebras (X, c) and (Y, d) is a function $h\colon X \to Y$ such that $d \circ h = Fh \circ c$. An *F-bisimulation* between two F-coalgebras (X, c) and (Y, d) is a relation $R \subseteq X \times Y$ that can be equipped with an F-coalgebra structure γ turning both projections $\pi_l\colon R \to X$ and $\pi_r\colon R \to Y$ into coalgebra homomorphisms. Two elements $x \in X$ and $y \in Y$ are *F-bisimilar*, denoted by $x \sim_F y$, if there exists a bisimulation R containing the pair (x, y). If F is clear from the context we write $\sim$ instead of $\sim_F$.

Example 2.1. Let A be a set. For the functor $A \times \mathsf{Id}$, a coalgebra consists of a set X and a function $\langle o, \delta\rangle\colon X \to A \times X$. Here $\langle o, \delta\rangle$ denotes the pairing of the *output* function $o\colon X \to A$ and the *next state* function $\delta\colon X \to X$. Given sets A and B, coalgebras for the functor $LX = B + (A \times X)$ are representations of infinite lists over A and finite lists over A with termination in B.

A (single-sorted) *signature* $\Sigma = (\Sigma_n)_{n\in\mathbb{N}}$ can be represented by a polynomial Set functor defined by $H_\Sigma(X) = \coprod_{n\in\mathbb{N}} \Sigma_n \times X^n$. A Σ-coalgebra over the set of variables X is given by a function assigning each $x \in X$ to a term $\sigma(x_1, \ldots, x_n)$, where $\sigma \in \Sigma_n$ is an operator of arity $n \geq 0$, and $x_i \in X$ for all $1 \leq i \leq n$.

For a given functor F, the *final coalgebra* (Ω, ξ_F) (if it exists) is a canonical domain of behaviour of F-coalgebras, with the property that for any F-coalgebra (X, c) there exists a unique homomorphism $h \colon X \to \Omega$ into it [23]. Final coalgebras exist under mild conditions on the functor.

Example 2.2. The carrier of the final coalgebra for the functor $A \times \mathsf{Id}$ consists of the set of all infinite lists over A. For $LX = B + (A \times X)$, the final coalgebra consists of all finite lists in A^*B and infinite lists in A^ω. It is thus given by the set $A^\star B \cup A^\omega$ with coalgebra map $\zeta \colon A^\star B \cup A^\omega \to B + (A \times (A^\star B \cup A^\omega))$ defined by $\zeta(b) = b$ and $\zeta(aw) = \langle a, w \rangle$ for all $a \in A$, $b \in B$ and $w \in A^\star B \cup A^\omega$. The final coalgebra of a signature functor H_Σ is given by $\omega \colon T_\Sigma^\infty \to \Sigma(T_\Sigma^\infty)$ where T_Σ^∞ is the set of all finite and infinite Σ-trees (see, for instance, [1]).

One of the central elements of the theory of coalgebras is the (proof) principle of *coinduction*, which says that bisimilar states are mapped to the same element of the final coalgebra: if $x \sim y$ then $h(x) = h(y)$. Establishing bisimulations is a concrete proof method for bisimilarity, and thus, by the above principle, for equality in the final coalgebra. If the functor preserves weak pullbacks, a rather mild condition satisfied by all of the above examples, the converse holds as well [23], i.e., $h(x) = h(y)$ implies $x \sim y$. In the following sections we implicitly assume all functors to preserve weak pullbacks.

3 A Semantic Approach to Coinductive Types

In this section we propose a framework for coinductive types without union. We use two functors F and G as follows: F-coalgebras are interpreted as (recursive) *type* definitions, whereas G-coalgebras are (recursive) *value* definitions. We assume that the final coalgebras of F and G exist. The carrier $\mathbb{T}$ of the final F-coalgebra $(\mathbb{T}, \xi_F)$ consists of all coinductive types. The carrier $\mathbb{V}$ of the final G-coalgebra $(\mathbb{V}, \xi_G)$ is the set of all coinductive values.

Example 3.1. A type definition such as $x \mapsto <$ elm: int, nxt: $y >$ together with $y \mapsto <$ elm: bool, nxt: $x >$, can be given as a coalgebra for $\{\mathsf{int}, \mathsf{bool}\} \times \mathsf{Id}$. The homomorphism into the final coalgebra maps x to $\mathsf{int}, \mathsf{bool}, \mathsf{int}, \mathsf{bool}, \ldots \in \mathbb{T}$.

An infinite recursive definition $p_i \mapsto (i, q_i)$ and $q_i \mapsto (\mathit{true}, p_{i+1})$ for $i \in \mathbb{N}$ can be represented as a coalgebra for the functor $(\mathbb{N} + \mathbb{B}) \times \mathsf{Id}$, where $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ is the set of Boolean values, and $\mathbb{N}$ is the set of non-negative integers. Then p_0 is mapped to the infinite list $0, \mathit{true}, 1, \mathit{true}, 2, \ldots \in \mathbb{V}$ in the final coalgebra.

The functors F and G will be connected by a *natural transformation.* A natural transformation $\alpha \colon G \Rightarrow F$ associates to every set X a function $\alpha_X \colon GX \to FX$ such that for any function $f \colon X \to Y$ we have $Ff \circ \alpha_X = \alpha_Y \circ Gf$. In order to assign types to values we assume given a natural transformation $\alpha \colon G \Rightarrow F$, which represents an assignment of types to basic values. We will exhibit an example below, but first we set up the general framework, which coinductively assigns types to values. More precisely, by applying the natural transformation α to the final G-coalgebra, we turn it into an F-coalgebra and thus obtain a unique F-coalgebra homomorphism from coinductive values to coinductive types. This is depicted in the middle of the diagram below.

$$\begin{tikzcd}
V \arrow[r, dashed, "\nu"] \arrow[d, "d"] & \mathbb{V} \arrow[r, dashed, "\tau"] \arrow[d] & \mathbb{T} \arrow[dd] & X \arrow[l, dashed, "\rho"'] \arrow[dd, "c"] \\
GV \arrow[r, "G\nu"] \arrow[d, "\alpha_V"] & G\mathbb{V} \arrow[d, "\alpha_{\mathbb{V}}"] & & \\
FV \arrow[r, "F\nu"] & F\mathbb{V} \arrow[r, "F\tau"] & F\mathbb{T} & FX \arrow[l, "F\rho"']
\end{tikzcd} \tag{3}$$

The map τ defined by finality gives the assignment of types to values. The left and the right side of the diagram are representations:

– Given a *value representation* $d\colon V \to GV$ we let $\nu\colon V \to \mathbb{V}$ be the unique coalgebra homomorphism and extend d, again using α, to a F-coalgebra. This is depicted in the two commuting squares on the left side of the diagram.

– Given a *type representation* $c\colon X \to FX$ we let $\rho\colon X \to \mathbb{T}$ be the unique homomorphism into $\mathbb{T}$, as depicted on the right side of the diagram. A *typing* relation between V and X is then defined in the obvious way: given $p \in V$ and $x \in X$ we let $p : x$ iff $\tau(\nu(p)) = \rho(x)$.

Example 3.2. Continuing the above Example 3.1, we can define $\alpha\colon ((\mathbb{N} + \mathbb{B}) \times \mathsf{Id}) \Rightarrow (\{\mathsf{int}, \mathsf{bool}\} \times \mathsf{Id})$ for every set S simply by putting $\alpha_S((n, s)) = (\mathsf{int}, s)$ and $\alpha_S((b, s)) = (\mathsf{bool}, s)$ for all $n \in \mathbb{N}$, $b \in \mathbb{B}$, and $s \in S$. For the concrete type and value definitions x and p_0 respectively, of Example 3.1, it is easy to check that $\tau(\nu(p_0)) = \rho(x)$, so $p_0 : x$ as expected. In fact, as we will see below, this can be checked by establishing a bisimulation.

In the above approach the meaning of a type declaration $c\colon X \to FX$ is given by finality, in terms of the unique homomorphism $\rho\colon X \to \mathbb{T}$. It is thus independent of the language of values. Next we interpret types semantically, as sets of values, and subsequently we relate the two interpretations.

Definition 3.1. *Types are interpreted as sets of values by* $[\![-]\!]\colon \mathbb{T} \to \mathcal{P}(\mathbb{V})$, *defined as the inverse of* τ, *i.e.,* $[\![t]\!] = \{v \in \mathbb{V} \mid \tau(v) = t\}$ *for any* $t \in \mathbb{T}$.

It follows from the above definition that if $[\![t_1]\!] = [\![t_2]\!]$ and both $[\![t_1]\!]$ and $[\![t_2]\!]$ are non-empty, then $t_1 = t_2$. Types are inhabited by values (thus non-empty) if the natural transformation $\alpha\colon G \Rightarrow F$ mapping values to types is surjective in all of its components, i.e., α_X is surjective for any set X.

Lemma 3.1. *If* α *is a surjective natural transformation then* τ *is surjective.*

Corollary 3.1. *If* α *is a surjective natural transformation then* $t_1 = t_2$ *if and only if* $[\![t_1]\!] = [\![t_2]\!]$ *for all* $t_1, t_2 \in \mathbb{T}$.

Note that if $[\![t_1]\!] \subseteq [\![t_2]\!]$ then $[\![t_1]\!] = [\![t_2]\!]$. Subtyping will become relevant in the next section, where we consider subsets of $\mathbb{T}$.

To see why surjectivity is a natural condition, consider the type definitions x and y from Example 3.1, and $\alpha\colon (\mathbb{N} \times \mathsf{Id}) \Rightarrow (\{\mathsf{int}, \mathsf{bool}\} \times \mathsf{Id})$ given by $\alpha_S(n, s) = (\mathsf{int}, s)$. In this case clearly $\rho(x) \neq \rho(y)$, whereas $[\![\rho(x)]\!] = \emptyset = [\![\rho(y)]\!]$.

Equality of types coincides with *bisimilarity*, by coinduction: $\mathbb{T}$ is a final coalgebra. Thus we obtain the following soundness and completeness result.

Theorem 3.1. *Using the setting of* (3)*, let* $c\colon X \to FX$ *be a coalgebra, and* α *surjective. For all* $x, y \in X$ *we have* $[\![\rho(x)]\!] = [\![\rho(y)]\!]$ *iff* $\rho(x) = \rho(y)$ *iff* $x \sim_F y$*.*

If F is a polynomial functor (constructed by finite sum and product) and we restrict to type declarations using only finitely many variables (so that types essentially represent rational trees over a signature) then bisimulation is not only a sound and complete proof method for type equality, but it is also decidable [7]. We note that in the above framework, computing the typing relation can be seen as a special case of type equality (by turning G-coalgebras into F-coalgebras), and therefore it can also be computed using bisimulations.

4 Coinductive Union Types

In the previous section we have introduced a coalgebraic semantics, where types, i.e., elements of the final coalgebra $\mathbb{T}$, are equal if and only if they represent the same sets of values. Types and values can be represented by coalgebras, and bisimulation provides a concrete proof principle for type equivalence. In the current section we are interested in extending these results to *union* types, that is, *subsets* of $\mathbb{T}$. By $\mathcal{P}(X)$ we denote the power set functor applied to a set X, i.e., the set of subsets of X; for a function $f\colon X \to \mathcal{P}(Y)$ we write $f^\sharp\colon \mathcal{P}(X) \to \mathcal{P}(Y)$ for its direct image. In the previous section we have coinductively constructed a map $\tau\colon \mathbb{V} \to \mathbb{T}$ from values to types, from which the semantics $[\![t]\!]$ of a type $t \in \mathbb{T}$ as a set of values can be defined simply by using the inverse. In order to have a natural counterpart of Theorem 3.1 in the setting of *subtyping* we extend the semantics to sets of types using direct image $[\![-]\!]^\sharp\colon \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{V})$, i.e., $[\![S]\!]^\sharp = \{v \in \mathbb{V} \mid \tau(v) \in S\}$.

Theorem 4.1. *If* α *is a surjective natural transformation then* $T_1 \subseteq T_2$ *if and only if* $[\![T_1]\!]^\sharp \subseteq [\![T_2]\!]^\sharp$*, for all* $T_1, T_2 \subseteq \mathbb{T}$*.*

One of the main problems is to *represent* elements of $\mathcal{P}(\mathbb{T})$ as coalgebras. In the previous section we have seen how an F-coalgebra represents a type definition; it is natural to consider a $\mathcal{P}F$-coalgebra instead, in the case of union types, adding a top-level union constructor. The problem here is that the *branching* of $\mathcal{P}F$-coalgebras should not be considered. Indeed, $\mathcal{P}(\mathbb{T})$ is not the final coalgebra of $\mathcal{P}F$—in fact, $\mathcal{P}F$ does not even have a final coalgebra for cardinality reasons. But even if we restrict ourselves to $\mathcal{P}_f F$ (where $\mathcal{P}_f(X)$ is the set of finite subsets of X), then the final coalgebra consists of finitely branching synchronization trees labelled in a and quotiented by strong bisimilarity. Instead, we need the *trace semantics* of $\mathcal{P}F$-coalgebras. To this end we base ourselves on the coalgebraic trace semantics of [21].

Definition 4.1. *Let $c\colon X \to \mathcal{P}(FX)$ be a coalgebra, and $(\mathbb{T}, \xi_{\mathbb{T}})$ the final F-coalgebra. A* trace map *tr is a map that makes the following diagram commute:*

$$\begin{tikzcd}
X \arrow[r, "tr"] \arrow[d, "c"] & \mathcal{P}(\mathbb{T}) \arrow[d, "\mathcal{P}(\xi_{\mathbb{T}})"] \\
\mathcal{P}(FX) \arrow[r, "(\bar{F}tr)^{\sharp}"] & \mathcal{P}(F\mathbb{T})
\end{tikzcd}$$

where $\bar{F}(tr)$ is defined by relation lifting [21]. If the diagram does not commute but $\mathcal{P}(\xi_{\mathbb{T}}) \circ tr \subseteq (\bar{F}tr)^{\sharp} \circ c$, then we say tr is a quasi trace map.

Instead of recalling the definition of relation lifting, we introduce it by examples.

Example 4.1. Consider the functor $FX = B + (A \times X)$. Then $\mathbb{T} = A^*B \cup A^\omega$ (see Example 2.2). A coalgebra $c\colon X \to \mathcal{P}FX$ is a *nondeterministic Moore automaton.* A *trace map* is a map $tr\colon X \to \mathcal{P}(A^*B \cup A^\omega)$ such that for all $b \in B$: $b \in tr(x)$ iff $b \in c(x)$, and for all $aw \in A(A^*B \cup A^\omega)$: $aw \in tr(x)$ iff $(a, y) \in c(x)$ and $w \in tr(y)$ for some $y \in X$. For a quasi trace map, these equivalences are relaxed to implications from left to right.

Given any signature functor H_Σ (Example 2.1), a $\mathcal{P}H_\Sigma$-coalgebra is a *nondeterministic top-down tree automaton.* The trace map associated with a coalgebra $c\colon X \to \mathcal{P}(H_\Sigma X)$ satisfies the following: $\sigma \in tr(x)$ iff $\sigma \in \Sigma_0 \cap c(x)$, and $\sigma(k_1, \ldots, k_n) \in tr(x)$ iff $\langle \sigma, x_1, \ldots, x_n \rangle \in \Sigma_n \times X^n \cap c(x)$ and $k_i \in tr(x_i)$ for $1 \leq i \leq n$. Again, for a quasi trace map these are implications from left to right.

The set of maps of type $X \to \mathcal{P}(\mathbb{T})$ forms a complete lattice, by pointwise extension of the subset inclusion order on $\mathcal{P}(\mathbb{T})$. A trace map can be viewed as a fixpoint of a map on this complete lattice; since relation lifting is monotone, this is a monotone map, and therefore, by the Knaster-Tarski theorem, for a fixed $\mathcal{P}F$-coalgebra the greatest trace map as well as the least trace map exist (a similar approach is taken in [10]). To model *coinductive* types we are interested in this greatest trace map (in the sequel typically denoted by T and called *maximal traces*). Moreover, we get the following proof principle: if tr is a quasi trace map, then it is a post-fixed point of the above monotone map, so it is (pointwise) included in the greatest one: $tr \subseteq T$. This proof technique is applied in Section 5.

Example 4.2. Continuing Example 4.1, the *least* trace map t for a non-deterministic Moore automaton assigns to a state the standard definition of its finite traces in A^*B. The *greatest* trace map T assigns to a state the finite traces as well as the infinite traces in A^ω. For example, recall the type definition x_1 from equation (1) of the introduction, representing finite and infinite lists of integers, and x_2 from equation (2) representing infinite lists of integers. They clearly define Moore automata. For the least trace map t we have $t(x_1) = \mathsf{int}^*\mathsf{null}$ and $t(x_2) = \emptyset$. For the greatest trace map T we have $T(x_1) = t(x_1) \cup \mathsf{int}^\omega$ and $T(x_2) = \mathsf{int}^\omega$ (i.e. the desired coinductive types of definitions x_1 and x_2).

For a non-deterministic (top-down) tree automaton, the least trace map is simply the standard semantics of tree automata, assigning a tree language (of finite trees) to each state. The greatest trace map contains this language as well as all *infinite* trees such that, when parsed, the automaton does not block. These tree automata can be used to represent type definitions, similarly to Moore automata, but generalizing this to arbitrary (finite) use of the product constructor.

Corollary 4.1. *For any coalgebra $c\colon X \to \mathcal{P}(FX)$ and any $x, y \in X$ we have $T(x) \subseteq T(y)$ iff $[\![T(x)]\!]^\sharp \subseteq [\![T(y)]\!]^\sharp$ (given that α is surjective).*

Thus, subset inclusion between syntactic unfoldings of sets of types is sound and complete with respect to *semantic subtyping*, i.e., inclusion between types interpreted as sets of values. Unfortunately, since $\mathcal{P}(\mathbb{T})$ is not a final coalgebra, we do not obtain bisimilarity (or similarity) as a proof principle, as was the case in the framework of Section 3. We address the problem of proving subtyping in the following section.

5 Approximating Coinductive Union Types

By the main results of the previous section, semantic subtyping coincides with subtyping between sets of maximal traces, that is, syntactic unfoldings of type definitions. In this section we provide a generally applicable technique to reduce subtyping to inclusion between *finite* traces. This is based on finite *approximation* of maximal traces, which we introduce below.

We fix a functor $F\colon \mathsf{Set} \to \mathsf{Set}$ (preserving weak pullbacks) and a coalgebra $c\colon X \to \mathcal{P}FX$. In order to define approximation, consider the functor $F_\perp = F + \{\perp\}$, and the natural transformation $\gamma\colon \mathcal{P}F \Rightarrow \mathcal{P}F_\perp$ given by $\gamma_X(S) = S \cup \{\perp\}$. We can now turn c into the $F_\perp$-coalgebra $\gamma_X \circ c$. It is our aim to use the finite traces of $\gamma_X \circ c$ to approximate the maximal traces of c. We use the approach of [18] to finite trace semantics via finality in the category Rel, where objects are sets and morphisms are relations (represented as functions $X \to \mathcal{P}(Y)$).

Central to this approach is the *initial algebra* of $F_\perp$, which we denote by $\iota\colon F_\perp\mathbb{I} \to \mathbb{I}$. By Lambek's lemma ι is an isomorphism. Now, by [18, Theorem 3.8], $\mathbb{I}$ is the *final coalgebra* in Rel, for the functor $\bar{F}_\perp$ defined by relation lifting. Thus, for any $F_\perp$-coalgebra in Rel, that is, a $\mathcal{P}F_\perp$-coalgebra in Set, we obtain a unique map into $\mathcal{P}(\mathbb{I})$. Applying this to a coalgebra $\gamma_X \circ c\colon X \to \mathcal{P}F_\perp X$ as constructed above, we get a unique map $t_\perp\colon X \to \mathcal{P}(\mathbb{I})$ as in (4).

$$\begin{tikzcd}
X \arrow[r, "t_\perp"] \arrow[d, "c"] & \mathcal{P}(\mathbb{I}) \arrow[dd] \\
\mathcal{P}(FX) \arrow[d, "\gamma_X"] & \\
\mathcal{P}(F_\perp X) \arrow[r, "(\bar{F}_\perp t_\perp)^\sharp"] & \mathcal{P}(F_\perp \mathbb{I})
\end{tikzcd} \qquad (4)$$

Example 5.1. For a non-deterministic Moore automaton $c\colon X \to \mathcal{P}(B + (A \times X))$, the above construction yields the finite trace semantics for $\gamma_X \circ c$, which is the Moore automaton obtained by adding the output $\perp$ to each state. We regard a word $w\perp$ as a *prefix* of a word $v \in A^*B \cup A^\omega$ if $wv' = v$ for some v'; in this sense, $t_\perp(x)$ is prefix-closed.

Applying the above construction to $\mathbb{T}$, we get a map $\mathit{approx} \colon \mathbb{T} \to \mathcal{P}(\mathbb{I})$. This map, informally, computes the approximations of maximal traces. Consider now the following map defined from it: $\mathit{maxtr} \colon \mathcal{P}(\mathbb{I}) \to \mathcal{P}(\mathbb{T})$, given by $\mathit{maxtr}(S) = \{w \in \mathbb{T} \mid \mathit{approx}(w) \subseteq S\}$. The map *maxtr* computes the set of maximal traces represented by a set of approximations. The following lemma states that the function $t_\perp$ can be represented as the approximation of maximal traces.

Lemma 5.1. $t_\perp = \mathit{approx}^\sharp \circ T$.

This follows from the fact that $\mathbb{I}$ is final in Rel. As a simple consequence of this result and the fact that *maxtr* is defined as the (upper) inverse of *approx*, we now obtain the following:

Corollary 5.1. $T \subseteq \mathit{maxtr} \circ t_\perp$.

The converse of the above corollary does not hold in general. There is a standard counterexample (e.g., [17]): take a non-deterministic Moore automaton containing a state x that accepts all *finite* traces of the form $a^n b$ (for some b and all $n \geq 0$), but not the infinite trace $a^\omega = aaa\ldots$ (such an automaton can be realized using infinite branching). Then $\mathit{maxtr} \circ t_\perp(x)$ contains a^ω, whereas $T(x)$ does not.

To prove the converse for restricted classes of coalgebras, we use that T is a greatest fixpoint. Under the condition that $\mathit{maxtr} \circ t_\perp$ is a quasi trace map, we obtain the soundness and completeness of finite traces w.r.t. (semantic) subtyping.

Theorem 5.1. *Let $c \colon X \to \mathcal{P}(FX)$ be a coalgebra such that $\mathit{maxtr} \circ t_\perp$ is a quasi trace map. Then for any $x, y \in X$: $t_\perp(x) \subseteq t_\perp(y)$ iff $T(x) \subseteq T(y)$.*

Proof. Suppose $t_\perp(x) \subseteq t_\perp(y)$. If $\mathit{maxtr} \circ t_\perp$ is a quasi trace map then $\mathit{maxtr} \circ t_\perp \subseteq T$; combined with Corollary 5.1, this yields $\mathit{maxtr} \circ t_\perp = T$. Conversely, if $T(x) \subseteq T(y)$ then $\mathit{approx}^\sharp \circ T(x) \subseteq \mathit{approx}^\sharp \circ T(y)$, so $t_\perp(x) \subseteq t_\perp(y)$ by Lemma 5.1.

Moore automata. As shown in Example 4.1, non-deterministic Moore automata can be used to represent types for finite and infinite lists. However, in general they do not satisfy the condition of Theorem 5.1; we need to make an appropriate restriction on the branching behaviour. We say $c \colon X \to \mathcal{P}(B + (A \times X))$ is *image-finite* when for any $x \in X$ and any $a \in A$: $c(x)$ may contain finitely many elements of the form (a, x) (but infinitely many of B, and A may itself be infinite).

Proposition 5.1. *For any image-finite Moore automaton: $t_\perp(x) \subseteq t_\perp(y)$ iff $T(x) \subseteq T(y)$.*

Proof. Let c be image-finite. Using Example 4.1, we see that to prove that $\mathit{maxtr} \circ t_\perp$ is a trace map, is to prove that 1) $b \in \mathit{maxtr} \circ t_\perp(x)$ implies $b \in c(x)$, and 2) for all $aw \in A(A^* B \cup A^\omega)$: if $aw \in \mathit{maxtr} \circ t_\perp(x)$ then $(a, y) \in c(x)$ and $w \in \mathit{maxtr} \circ t_\perp(y)$ for some $y \in X$. The first part 1) is easy: $b \in \mathit{maxtr} \circ t_\perp(x)$ implies $b \in t_\perp(x)$, which in turn implies $b \in c(x)$. For 2), suppose $aw \in \mathit{maxtr} \circ t_\perp(x)$. Then $w \in \bigcup_{(a,y) \in c(x)} \mathit{maxtr} \circ t_\perp(y)$; by image-finiteness, this is a *finite* union. The case that w is finite is straightforward; suppose w is infinite. Then $\mathit{approx}(w)$ is infinite; and thus there is some y for which infinitely many prefixes of w are contained in $t_\perp(y)$. But $t_\perp(y)$ is prefix-closed; so $w \in \mathit{maxtr} \circ t_\perp(y)$.

Example 5.2. Consider the following type definition.

$$\begin{aligned} &x_3 \mapsto <\mathsf{elm\colon int,\ nxt\colon}\ x_4 > \\ &x_4 \mapsto \mathsf{null} \vee <\mathsf{elm\colon int,\ nxt\colon}\ x_4 > \vee <\mathsf{elm\colon bool,\ nxt\colon}\ x_4 > \ . \end{aligned} \tag{5}$$

In the coinductive interpretation this represents all finite and infinite lists of integers and booleans that start with an integer. Consider the types below:

$$\begin{aligned} &x_5 \mapsto <\mathsf{elm\colon int,\ nxt\colon}\ x_6 > \vee <\mathsf{elm\colon int,\ nxt\colon}\ x_7 > \vee <\mathsf{elm\colon int,\ nxt\colon}\ x_8 > \\ &x_6 \mapsto <\mathsf{elm\colon bool,\ nxt\colon}\ x_8 > \vee <\mathsf{elm\colon bool,\ nxt\colon}\ x_6 > \vee <\mathsf{elm\colon int,\ nxt\colon}\ x_6 > \\ &x_7 \mapsto <\mathsf{elm\colon int,\ nxt\colon}\ x_8 > \vee <\mathsf{elm\colon bool,\ nxt\colon}\ x_7 > \vee <\mathsf{elm\colon int,\ nxt\colon}\ x_7 > \\ &x_8 \mapsto \mathsf{null}\,. \end{aligned} \tag{6}$$

Here x_6 and x_7 represent infinite lists, as well as finite lists ending with bool and int, respectively. We can now prove that $T(x_3) \subseteq T(x_5)$ by reducing it to $t_\perp(x_3) \subseteq t_\perp(x_5)$, which is a simple case of language inclusion.

Tree automata. A tree automaton $c\colon X \to \mathcal{P}^+(H_\Sigma X)$ is said to be *image finite* if for all $x \in X$ and $\sigma \in \Sigma^\perp$ there are only finitely many tuples $\langle \sigma, x_1, \ldots x_n \rangle \in c(x)$, where n is the arity of σ.

Proposition 5.2. *For any image-finite tree automaton: $t_\perp(x) \subseteq t_\perp(y)$ iff $T(x) \subseteq T(y)$.*

The proof is a straightforward extension of the case of Moore automata. Thus, we obtain inclusion of *tree languages* (of finite trees) as a sound and complete method to show semantic subtyping of recursive types with product and union, interpreted coinductively. For regular tree languages, i.e., languages accepted by a top-down non-deterministic tree automaton with finitely many states, language inclusion (and thus subtyping) is decidable, although it is EXPTIME-complete [11].

6 Related Work

Axiomatizations and algorithms for subtyping on recursive types interpreted coinductively have been proposed by Amadio and Cardelli [2] in the context of functional programming; subsequently, a more concise sound and complete axiomatization has been proposed by Brandt and Henglein [9], with a novel rule for a finitary coinduction principle. In these papers types are interpreted as ideals in a universal domain, hence they do not follow the semantic subtyping approach where subtyping corresponds to the subset relation. Furthermore, types have no Boolean operators; as we will see, introducing union types makes sound and complete axiomatization of subtyping more challenging.

Damm [12] proves decidability of subtyping between recursive types with intersection, union, and function types, by reduction to the problem of inclusion between regular tree expressions. However, the paper does not consider record types, and, more importantly, types are interpreted inductively, rather than coinductively, over a rather complex metric space of ideals. As a consequence, the corresponding subtyping relation is not comparable with ours. Di Cosmo et al. [13] study subtyping of recursive types up to associativity and commutativity of products; their definition of subtyping is fully axiomatic, and only products and arrow types are considered, no Boolean operators. A nice introduction to the fundamental theory of recursive types and subtyping can be found in the work by Gapeyev et al. [16]; the survey does not consider Boolean operators, and subtyping is defined axiomatically, hence a type interpretation is not introduced.

Semantic subtyping in the presence of Boolean operators and product or record type constructors has been intensively studied in the context of the XDuce [20] and CDuce [6] programming languages. As in our case, the subtyping relation corresponds to a natural semantic notion: types denote sets of documents (that is, sets of finite trees), and subtyping coincides with inclusion between the sets denoted by two types. The main difference with coinductive types is their interpretation: types in both XDuce and CDuce are interpreted inductively, therefore a type definition as (2) corresponds to the empty set of values; as a matter of fact, types in XDuce and CDuce fail to capture cyclic values. Even though CDuce supports references, and, hence, it is possible to create cycles, the types that can be correctly assigned to cyclic values are "inductive".

Semantic subtyping with union and coinductive types has been studied in the context of precise static type analysis for object-oriented programming [3]. Sound but not complete axiomatizations of subtyping have been defined in [4, 5].

7 Future Work

The coalgebraic framework presented in this paper provides the basis for an extensive, structured investigation of subtyping for coinductive union types.

The subtyping relation could be refined by allowing subtyping between primitive types (e.g., nat is a subtype of int) as well as depth and width subtyping between records. Technically, this could be achieved by moving our framework from the category Set to the category of partially ordered sets.

The methods in [14, 8] allow to canonically derive sound and complete axiomatizations for the rational subset of the final coalgebra of a polynomial functor. For example, one can easily obtain a calculus for subtyping, by combining the axiomatisation of tree regular expressions of [14] with the approximation results of Section 5 of the present paper.

In our framework we abstracted from concrete calculi of expressions evaluating to values. It would be interesting to integrate the bialgebraic approach [22] (defining syntax and semantics of expressions) within our framework by allowing the specification of typing rules for each operator.

References

1. P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: A coalgebraic view. *Theoretical Computer Science*, 300(1–3):1–45, 2003.

2. R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), 1993.
3. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP 2009*, vol. 5653 of *LNCS*, pp. 2–26. Springer, 2009.
4. D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In *GandALF 2010*, vol. 25 of *EPTCS*.
5. D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. *FTfJP 2010*, ACM Digital Library, 2010.
6. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. *ICFP*, 2003.
7. M. Bonsangue, G. Caltais, E.-I. Goriac, D. Lucanu, J. Rutten, and A. Silva. Automatic equivalence proofs for non-deterministic coalgebras. In *Science of Computer Programming* 798(9):1324-1345, 2013.
8. M. Bonsangue, S. Milius, A. Silva. Sound and Complete Axiomatizations of Coalgebraic Language Equivalence. *ACM Trans. on Comp. Logic* 14(1):7, 2013.
9. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamentae Informatica*, 33(4), 1998.
10. C. Cîrstea. From Branching to Linear Time, Coalgebraically. *FICS 2013*, vol. 126 of *EPTCS*, pp. 11-27, 2013.
11. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* Available on: http://www.grappa.univ-lille3.fr/tata.
12. F. Damm. Subtyping with Union Types, Intersection Types and Recursive Types. *TACS'94 - Theoretical Aspects of Computer Software*, pp. 687-706, Springer, 1994.
13. R. Di Cosmo, F. Pottier, and D. Rémy. Subtyping Recursive Types Modulo Associative Commutative Products. *Typed Lambda Calculi and Applications, TLCA 2005*, pp. 179-193, Springer, 2005.
14. Z. Ésik. Axiomatizing the equational theory of regular tree languages. *Journal of Logic and Algebraic Programming* 79(2): 189-213, 2010.
15. A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of the ACM*, 2008.
16. V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed *The Journal of Functional Programming* 12(6): 511-548, 2002.
17. R. van Glabbeek. The linear time - branching time spectrum I. The semantics of concrete, sequential processes. In *Handbook of Process Algebra*, pp. 3-99, 2001.
18. I. Hasuo, B. Jacobs, A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*. 3:1-36, 2007.
19. H. Hosoya, J. Vouillon, B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.* 27(1):46–90, 2005
20. H. Hosoya, B. C. Pierce: XDuce. A statically typed XML processing language. *ACM Trans. Internet Techn.* 3(2):117–148, 2003
21. B. Jacobs. Trace Semantics for Coalgebras. In *CMCS '04*, *ENTCS* 106, 2004.
22. B. Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, 2011.
23. J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249, 2000.