Sound and Complete Subtyping between Coinductive Types for Object-Oriented Languages*

Davide Ancona and Andrea Corradi

DIBRIS, Università di Genova, Italy
davide.ancona@unige.it, andrea.corradi@dibris.unige.it

Abstract. Structural subtyping is an important notion for effective static type analysis; it can be defined either axiomatically by a collection of subtyping rules, or by means of set inclusion between type interpretations, following the more intuitive approach of semantic subtyping, which allows simpler proofs of the expected properties of the subtyping relation.

In object-oriented programming, recursive types are typically interpreted inductively; however, cyclic objects can be represented more precisely by coinductive types.

We study semantic subtyping between coinductive types with records and unions, which are particularly interesting for object-oriented programming, and develop and implement a sound and complete top-down direct and effective algorithm for deciding it. To our knowledge, this is the first proposal for a sound and complete top-down direct algorithm for semantic subtyping between coinductive types.

1 Introduction

Subtyping between structural types is an essential notion for effective static type analysis of object-oriented languages, and, in particular, of dynamically typed languages like JavaScript and Python.

In most cases the subtyping relation is defined axiomatically, then algorithms have to be defined and proved to be (at least) sound and complete (if the relation is decidable) w.r.t. the given axioms. Such approaches have some drawbacks: since the relation is specified in an axiomatic way, it may fail to convey the right intuition behind it, or it may not be completely clear whether the definition fully captures such an intuition (that is, if the axiomatization is sound and complete w.r.t. some intended model); furthermore, proving even simple properties, like transitivity, may be quite hard.

Semantic subtyping has been proposed as a possible solution to these problems for XDuce [13] and ℂDuce [12], two statically typed domain specific languages expressly designed for type safe manipulation of XML documents. In semantic

* Partly funded by the project MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 282–307, 2014.

subtyping types are interpreted as sets of values, following the intuition that a type specifies all possible values that an expression of that type may denote; consequently, subtyping corresponds to set inclusion between type interpretations. In this way, the definition of subtyping is more intuitive, and several properties can be easily deduced (for instance, transitivity always holds trivially). Semantic subtyping is particularly suited to naturally supports Boolean type constructors; for instance, in terms of type interpretation Boolean disjunction and conjunction correspond to union and intersection on sets of values. Boolean type constructors (in particular union types) allow types and type analysis to be more precise, but their expressive power makes the definition of a sound and complete decision procedure for subtyping more challenging.

Another feature that complicates subtyping (but that is also indispensable) is type recursion; syntactically, a recursive type corresponds to a *regular* (a.k.a. rational) tree defined by a finite set of guarded syntactic equations. In the semantic subtyping approach, semantic interpretation of recursive types requires to consider the syntactic equations defining a type as semantics equations specifying sets of values; such equations can be interpreted either inductively or coinductively. Let us consider, for instance, the recursive type τ defined by

$$\tau = \mathit{null} \vee \langle \mathit{el}{:}\mathit{int}, \mathit{nx}{:}\tau \rangle.$$

The type is the union of *null*, denoting the null reference, and $\langle \mathit{el}{:}\mathit{int}, \mathit{nx}{:}\tau \rangle$, denoting all records equipped at least with the two fields *el* and *nx* having type *int* and τ, respectively (that is, τ corresponds to a simple implementation of linked lists of integer values). When we turn to consider the semantic interpretation of τ, denoted by $[\![\tau]\!]$, because the Boolean type constructor $\vee$ corresponds to union of values, we get the following recursive equation:

$$[\![\tau]\!] = \{\text{null}\} \cup [\![\langle \mathit{el}{:}\mathit{int}, \mathit{nx}{:}\tau \rangle]\!]$$

which is equivalent to the equation

$$[\![\tau]\!] = \{\text{null}\} \cup \{\langle \mathit{el} \mapsto v_{\mathit{el}}, \mathit{nx} \mapsto v_{\mathit{nx}}, \ldots \rangle \mid v_{\mathit{el}} \in [\![\mathit{int}]\!], v_{\mathit{nx}} \in [\![\tau]\!]\}$$

where $\langle \mathit{el} \mapsto v_{\mathit{el}}, \mathit{nx} \mapsto v_{\mathit{nx}}, \ldots \rangle$ denotes a record value with fields *el* and *nx* associated with values v_{el} and v_{nx}, and with possibly other fields. If such an equation is interpreted inductively (hence, τ is the least solution), then all values v in $[\![\tau]\!]$ are inductive, and the operation $v.\mathit{nx}.\mathit{nx}.\ldots.\mathit{nx}$ is defined only for a finite number of consecutive selections of field *nx*. If the equation is interpreted coinductively (hence, τ is the greatest solution), then $[\![\tau]\!]$ contains also coinductive values v for which the operation $v.\mathit{nx}.\mathit{nx}.\ldots.\mathit{nx}$ is defined also for an infinite number of consecutive selections of field *nx*; in other words, $[\![\tau]\!]$ contains also cyclic values.

To better outline the difference between the inductive and coinductive interpretation of recursive types, let us consider the recursive type τ' defined by

$$\tau' = \langle \mathit{el}{:}\mathit{int}, \mathit{nx}{:}\tau' \rangle.$$

In this case we get the equation

$$[\![\tau]\!] = \{\langle el \mapsto v_{el}, nx \mapsto v_{nx}, \ldots\rangle \mid v_{el} \in [\![int]\!], v_{nx} \in [\![\tau]\!]\}.$$

In this case the least solution of the equation is $[\![\tau]\!] = \emptyset$ (inductive interpretation), whereas the greatest solution is $[\![\tau]\!] = V$, with $V \neq \emptyset$; more precisely, V is the set of all integer lists for which the operation $v.nx.nx.\ldots.nx$ is always correct for an infinite number of consecutive selections of field nx. Therefore, whereas τ' is not very useful if interpreted inductively, when interpreted coinductively it specifies an interesting property that is verified by all cyclic lists.

As explained in the next section, the ability of representing cyclic values (hence, to interpret recursive types coinductively) allow more precise type analysis in all those situations where type correctness depends on the fact that objects (or, more in generally, values) are cyclic. Furthermore, since termination cannot be usually guaranteed through type analysis, and coinductive interpretations of types contain both inductive and coinductive values, coinductive interpretation of types leads to more expressive type systems.

Subtyping on coinductive types has been initially proposed by Amadio and Cardelli [1] in the context of functional programming; subsequently, an equivalent but more concise definition has been proposed by Brandt and Henglein [10]. In both approaches the subtyping relation is defined axiomatically (no semantic subtyping) and Boolean type constructors are not considered.

Semantic subtyping has been extensively studied in the context of the languages XDuce and $\mathbb{C}$Duce [13,12], but recursive types are interpreted inductively, because values in those languages correspond to XML documents, hence they cannot be cyclic. For XDuce the decision problem for the subtype relation reduces to the inclusion problem between tree automata, which is known to be EXPTIME-complete [14]. Despite this negative result, it is still possible to define practical top-down algorithms which work directly on types, and are not based on determinization of tree automata [14].

More recently, sound but not complete subtyping rules have been proposed for coinductive types with records and unions [4,5] in the context of abstract compilation. Subsequently, the problem of semantic subtyping has been proved to reduce to the inclusion problem between tree automata also for the coinductive case [9]; such a result has been generalized in the framework of coalgebras. However, to our knowledge, no practical sound and complete algorithm has been proposed for deciding semantic subtyping of coinductive types with Boolean type constructors.

The main contribution of this paper is the definition of a practical top-down algorithm for deciding semantic subtyping for coinductively interpreted types in the presence of record and union types. Such an algorithm is derived by a set of subtyping rules that is proved to be sound and complete w.r.t. semantic subtyping. To do that we propose and use a new proof technique that can be fruitfully used for proving soundness results for coinductively defined judgments (or, dually, for proving completeness results for inductively defined judgments). A prototype implementation of the algorithm has been developed and has been made available.

The rest of the paper is structured in the following way. Section 2 shows how coinductive types allow more precise type analysis in the presence of cyclic objects. Section 3 introduces basic definitions and results that are used in the rest of the paper. Section 4 defines semantic subtyping for coinductive record and union types, whereas subtyping rules and proofs of soundness and completeness can be found in Section 5. Finally, Section 6 presents the algorithm derived from the subtyping rules, and its prototype implementation, while Section 7 draws conclusion and proposes directions for future work.

2 A Motivating Example

In this section we present an example which shows how coinductive types allow more precise type analysis in the presence of cyclic objects. Let us consider the Python code in Figure 1 implementing circular linked lists (with dummy header).

Let us focus on the definition of the private method `getNode` of class `Node`, and try to find which type could be assigned to `self` for correctly type checking the body of the method (in Python the first argument of a method, conventionally called `self`, corresponds to `this` in Java).

Let us consider first the following possible candidate types:

$$\tau_1 = \mathit{null} \vee \langle \mathit{elem}{:}\tau_e, \mathit{next}{:}\tau_1 \rangle$$
$$\tau_2 = \langle\,\rangle \vee \langle \mathit{elem}{:}\tau_e, \mathit{next}{:}\tau_2 \rangle$$
$$\tau_3 = \langle \mathit{elem}{:}\tau_e \rangle \vee \langle \mathit{elem}{:}\tau_e, \mathit{next}{:}\tau_3 \rangle$$

Since for this example we are not particularly interested in the specific type of the elements of the lists, we assume that field *elem* has a certain unspecified type τ_e.

Types τ_1, τ_2 and τ_3 only differ for the base case: in τ_1 and τ_2 a sequence of nodes is terminated by null, and by the empty record, respectively, whereas in τ_3 terminal nodes are represented by the record type $\langle \mathit{elem}{:}\tau_e \rangle$.

Independently of their interpretation (either inductive, or coinductive), all types do not allow correct typechecking of the body of `getNode`, because if we assume that `self`, and, hence, the local variable `n`, has one of the tree types defined above, then the statement `n = n.next` is not type correct, because access of field `next` is not defined for the types *null*, $\langle\,\rangle$, and $\langle \mathit{elem}{:}\tau_e \rangle$.

Note that if we consider the analogous code for languages with nominal types like Java, then the body of the method is correctly typechecked since **this** has type `Node`, but in fact the code is not type safe, because in Java reference types always include the null reference, and the type system does not check access to the null reference (hence, well-typed code can throw the `NullPointerException` exception).

Let us now consider the following type τ:

$$\tau = \langle \mathit{elem}{:}\tau_e, \mathit{next}{:}\tau \rangle$$

If `self` (and, hence, `n`) has type τ, then the body of `getNode` typechecks, because now the statement `n = n.next` is type safe.

```
class Node:
        def __init__(self,elem):
                self.elem = elem
                self.next = self

        def getNode(self,index):
                n = self
                for i in range(0,index):
                        n = n.next
                return n

class CircularList:
        def __init__(self):
                self.head = Node(None)
                self.size = 0

        def __checkBounds(self,index,limit):
                if index < 0 or index >= limit:
                        raise IndexError("list index out of range")

        def add(self,index,elem):
                self.__checkBounds(index,self.size+1)
                n = self.head.getNode(index)
                tmp = Node(elem)
                tmp.next = n.next
                n.next = tmp;
                self.size+=1

        def get(self,index):
                self.__checkBounds(index,self.size)
                return self.head.getNode(index+1).elem
```

Fig. 1. Implementation of circular linked lists in Python

This result is independent of the interpretation of τ; however, as already observed in the introduction, if τ is interpreted inductively, then we get $[\![\tau]\!] = \emptyset$; but if the type of `self` is empty, then method `getNode` is useless, since no value can be passed to it. Indeed, since $[\![\tau]\!] = \emptyset$, by semantic subtyping we have that τ is subtype of any type, therefore any well-typed expression that can possibly return a value, cannot have type τ, otherwise the type system would be unsound.

For instance, the return type of method `__init__` of class `Node` (this is similar to a Java constructor) cannot be τ, because, otherwise the expression `Node(elem)` in method `add` would have the empty type τ, and this would not be sound. As a consequence, class `Node` and `CircularList` could not be typed if τ is interpreted inductively. On the contrary, if τ is interpreted coinductively, then $[\![\tau]\!] \neq \emptyset$, and both classes can be successfully typechecked.

We conclude this section by observing that if `self` has type τ_1, τ_2 or τ_3 as defined above, then method `getNode` can typecheck successfully if both the following items are verified:

1. the statement `n = n.next` is guarded by a suitable test; for instance, if `self` has type τ_1, then `n = n.next` should be replaced by the statement **if**`(n != None): n = n.next` (`None` is the equivalent of Java **null**);
2. type analysis has to be flow sensitive, and the type of `n` has to be narrowed in the `then` branch of the **if** statement we introduced.

Item 1 makes the code less efficient by adding a superfluous check that can be avoided if we know that class `Node` is only used by class `CircularList`. Item 2 requires a more sophisticated type analysis; flow sensitive typing and type narrowing are challenging tasks, especially in the presence of aliasing.

If `self` is assigned type τ (interpreted coinductively), then neither of the items above are required.

3 Background

In this section we define record and union coinductive types and present definitions and general results that will be used in the rest of the paper.

3.1 Types and Tree

In the rest of the paper we will deal with finitely branching trees which are allowed to contain infinite paths. A formalization of such infinite trees has been given by Courcelle [11]. In the rest of the paper by term we mean a finitely branching trees which are allowed to contain infinite paths, where nodes correspond to constructors, and the number of children of a node correspond to its arity.

These trees will represent, either types, or proof trees.

The following proposition states a well-known property of regular terms [11,16].

A system of guarded equations is a finite set of syntactic equations of shape $X = e$, where X is a variable, and e may contains variables, such that there exist no subsets of equations having shape $X_0 = X_1, \ldots, X_n = X_0$.

A solution to a set of guarded equations is a substitution to all variables contained in the equations that satisfies all syntactic equations.

Definition 1. *A regular tree is a possibly infinite tree containing a finite set of subtrees. A type is regular if it is a term that corresponds to a regular tree, that is, it has a finite set of subterms.*

Proposition 1. *Every regular tree t can be represented by a system of guarded equations.*

We define types as all regular terms coinductively defined as follows:

$$\tau \ ::= \mathbf{0} \mid \mathit{int} \mid \mathit{null} \mid \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n \rangle \mid \tau_1 \vee \tau_2$$

A record type $\langle f_1{:}\ \tau_1 \ldots f_n{:}\tau_n \rangle$ is a finite map from field names to types, therefore we implicitly assume that field names are distinct and their order is immaterial. If τ is a record type, then $\mathit{dom}(\tau)$ denotes the set of its fields, $\tau(f)$ the type associated with f (if $f \in \mathit{dom}(\tau)$), and $\tau[f{:}\tau']$ the update of record τ with the association of field f to type τ'.

Union type $\tau_1 \vee \tau_2$ intuitively represents the union of the value of τ_1 and τ_2 [8,15]. Type $\mathbf{0}$ is the empty type, and *int* represents the set $\mathbb{Z}$, and *null* denotes the singleton set containing the null reference.

Example 1. The type of all cyclic or non-cyclic integer lists can be defined by the following guarded equation:

$$T = \langle elm{:}int, next{:}T \rangle \vee null$$

The type is regular and has only the following four subterms:

$$T \ \langle elm{:}int, next{:}T \rangle \ int \ null$$

Example 2. Let us consider the terms T_i for all natural numbers i, defined by the following system of infinite guarded equations:

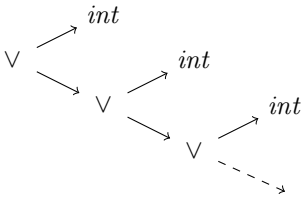
$$\begin{aligned} T_0 &= null \\ T_{i+1} &= \langle pred{:}\ T_i \rangle \ (\text{for all } i \geq 0) \end{aligned}$$

The type $T_0 \vee T_1 \vee \ldots \vee T_n \vee T_{n+1} \ldots$ is not a regular type.

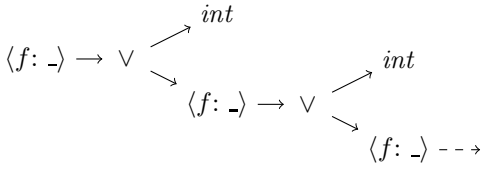
We now introduce the notion of *contractive* type, which allows us to reject all those types whose interpretation is not well-defined (see the example at the end of Section 4 for the details).

Definition 2. *A type is contractive if it does not contain infinite paths whose nodes are all labeled by union types.*

Example 3. The type $T = T \vee int$ is not contractive, because there exists an infinite path whose nodes are all labeled by the union type $T \vee int$.



Example 4. The type $T = \langle f{:}\ T \vee int \rangle$ is contractive because all infinite paths have nodes that are alternatively labeled by a record and a union type.



In the rest of the paper all types are restricted to be regular and contractive.

3.2 Principle of Induction and Coinduction

Let $\mathcal{U}$ denotes a set universe, and $\mathcal{P}(\mathcal{U})$ the powerset of $\mathcal{U}$. Given a set of rules defining a subset of $\mathcal{U}$, the *immediate consequence* operator F is the endofunction on the parts of $\mathcal{U}$, that given a set of premises X, returns the set of consequences immediately derivable from the rules.

Definition 3. *Let X be a set in $\mathcal{P}(\mathcal{U})$. X is F-closed if $F(X) \subseteq X$; X is F-consistent if $X \subseteq F(X)$; X is a fixed point of F if $X = F(X)$.*

Theorem 1 (Tarski-Knaster)
Let F:$\mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ be monotone.

- *The least fixed point (lfp) of F is the intersection of all F-closed sets.*
- *The greatest fixed point (gfp) of F is the union of all F-consistent sets.*

We denote as lfp(F) the least fixed point of F and as gfp(F) the greatest fixed point of F.

From the previous theorem the following induction and coinduction principles can be derived.

Induction principle. Let p and q be two predicates over $\mathcal{P}(\mathcal{U})$, and let p be inductively defined by a set of rules whose immediate consequence is F. If the rules for p are closed w.r.t. predicate q, then $\forall x \in \mathcal{U}\ p(x) \Rightarrow q(x)$ holds. This comes from the fact that by definition a rule is closed w.r.t. q iff the following implication holds: if the premises satisfy q, then the conclusion satisfies q. Indeed, this is equivalent to $F(\{x \mid q(x)\}) \subseteq \{x \mid q(x)\}$, which implies $\{x \mid p(x)\} \subseteq \{x \mid q(x)\}$ for the previous theorem.

Coinduction principle. Let p and q be two predicates over $\mathcal{P}(\mathcal{U})$, and let q be coinductively defined by a set of rules whose immediate consequence is F. Let us assume that the following property holds:
for all $x \in \mathcal{U}$, if $p(x)$ holds, then there exists a rule for q that can be applied to a set of premises satisfying p to derive the consequence x. Then $\forall x \in \mathcal{U}\ p(x) \Rightarrow q(x)$ holds. This comes from the fact that the condition above is equivalent to $\{x \mid p(x)\} \subseteq F(\{x \mid p(x)\})$, which implies $\{x \mid p(x)\} \subseteq \{x \mid q(x)\}$ for the previous theorem.

In the rest of the paper we will use the following convention: rules that have to be interpreted inductively use thin lines, while rules that have to be interpreted coinductively use thick lines.

4 Semantic Subtyping between Coinductive Types

We interpret types as sets of values. Values are all finite and infinite (but regular) terms coinductively defined as follows (where $i \in \mathbb{Z}$):

$$v ::= i \mid \text{null} \mid \langle f_1 \mapsto v_1, \ldots, f_n \mapsto v_n \rangle$$

Analogously to record types, record values are finite maps from field names to values, therefore we implicitly assume that field names are distinct and their order is immaterial. The interpretation of types is coinductively defined by the rules in Figure 2.

Thicker lines indicate that rules are interpreted coinductively, that is, also infinite proof trees are considered; this is equivalent to considering the greatest fixed-point of the function induced by the rules and corresponding to one step of inference [16].

Note that a record value can belong to a record type with fewer fields, the right-hand-side ellipsis in the record value indicates that the value is allowed to contain more fields.

$$(\text{null} \in)\ \frac{}{\text{null} \in \mathit{null}} \qquad (\text{int} \in)\ \frac{}{i \in \mathit{int}}\ i \in \mathbb{Z} \qquad (\text{l-or} \in)\ \frac{v \in \tau_1}{v \in \tau_1 \vee \tau_2} \qquad (\text{r-or} \in)\ \frac{v \in \tau_2}{v \in \tau_1 \vee \tau_2}$$

$$(\text{rec} \in)\ \frac{v_1 \in \tau_1, \ldots, v_n \in \tau_n}{\langle f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots \rangle \in \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n \rangle}$$

Fig. 2. Value membership

For instance, the following tree is a proof for $\langle f \mapsto 1 \rangle \in \mathit{int} \vee \langle f{:}\mathit{int} \rangle$.

$$(\text{r-or} \in)\ \frac{(\text{rec} \in)\ \dfrac{(\text{int} \in)\ \dfrac{}{1 \in \mathit{int}}}{\langle f \mapsto 1 \rangle \in \langle f{:}\mathit{int} \rangle}}{\langle f \mapsto 1 \rangle \in \mathit{int} \vee \langle f{:}\mathit{int} \rangle}$$

The following derivation for a non-contractive type motivates the definition of contractivity introduced in the previous section (see Def. 2); consider the regular type τ s.t. $\tau = \tau \vee \mathit{int}$, and the following infinite proof containing just applications of rules (l-or $\in$):

$$(\text{l-or} \in)\ \frac{(\text{l-or} \in)\ \dfrac{\vdots}{\mathit{null} \in \tau}}{\mathit{null} \in \tau}$$

Here we have a non-sound derivation as $\mathit{null} \in \tau$ derived above: τ corresponds to an infinite union of *int*, and therefore its interpretation cannot contain the *null* type. Non-contractive types can be correctly handled by introducing the notion of contractive proof tree [4]. Since from contractive types only contractive proofs can be derived, and non contractive types do not extend the expressive power[1] of types, it is more convenient to restrict types to contractive ones.

[1] Indeed, for any non contractive type there exists an equivalent contractive one.

Definition 4. *The interpretation of τ, is defined by $[\![\tau]\!] = \{v \mid v \in \tau \text{ holds}\}$.*

Lemma 1. *If $\tau = \mathbf{0}$ then $[\![\tau]\!] = \emptyset$, that is, $v \notin \mathbf{0}$.*

Proof. By definitions of membership rules we can not create a derivation for $v \in \mathbf{0}$ then by Def. 4 $[\![\tau]\!] = \emptyset$.

Lemma 2. *If $\tau = \tau_1 \vee \tau_2$ then $[\![\tau]\!] = \emptyset$ iff $[\![\tau_i]\!] = \emptyset$ $\forall i \in 1..2$, that is, $v \notin \tau_i$ $\forall i \in 1..2$.*

Proof.
$\Rightarrow$ By Def. 4 and by definitions of membership rules ($\nexists v.v \in \tau_i$ holds) $\forall \tau_i \in 1..n$, that is, by Def. 4 $[\![\tau_i]\!] = \emptyset$ $\forall t_i \in \tau$.
$\Leftarrow$ By Def. 4 we have ($\nexists v.v \in \tau_i$ holds) $\forall \tau_i \in \tau$, that is, by definitions of membership rules we can not create a derivation for $v \in \tau$ then by Def. 4 $[\![\tau]\!] = \emptyset$.

Lemma 3. *If $\tau = \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle$ then $[\![\tau]\!] = \emptyset$ iff $\exists i \in 1..n$ $[\![\tau_i]\!] = \emptyset$, that is, $\exists i \in 1..n$ $v_i \notin \tau_i$.*

Proof.
$\Rightarrow$ By Def. 4 and by definitions of membership rules $\exists i \in 1..n$. ($\nexists v.v \in \tau_i$ holds), that is, by Def. 4 $\exists i \in 1..n$ $[\![\tau_i]\!] = \emptyset$.
$\Leftarrow$ By Def. 4 we have $\exists i \in 1..n$. ($\nexists v.v \in \tau_i$ holds), that is, by definitions of membership rules we can not create a derivation for $v \in \tau$ then by Def. 4 $[\![\tau]\!] = \emptyset$.

Given a type τ, and a set of types Ξ, the restriction of τ w.r.t. Ξ, denoted by $\tau_{|\Xi}$, is coinductively defined as follows:

- $\tau_{|\Xi} = \tau$, if $\tau \in \{\mathbf{0}, \mathit{null}, \mathit{int}\}$;
- $(\tau_1 \vee \tau_2)_{|\Xi} = \tau_{1|\Xi} \vee \tau_{2|\Xi}$, if $\tau_1, \tau_2 \notin \Xi$;
- $(\tau_1 \vee \tau_2)_{|\Xi} = \mathit{null}$, if $\tau_1 \in \Xi$ or $\tau_2 \in \Xi$;
- $\langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle_{|\Xi} = \langle f_i{:}\tau_{i|\Xi} \mid 1 \leq i \leq n, \tau_i \notin \Xi\rangle$.

The restriction $\tau_{|\Xi}$ removes from τ all types contained in Ξ; intuitively, if $\tau_{|\Xi}$ returns a type whose interpretation is empty, then it means that the emptiness of τ can be proved without assuming any assumption on the types in Ξ (that is, those types no longer need to be inspected; see Lemma 6). For this reason, if either τ_1 or τ_2 are contained in Ξ, then $\tau_1 \vee \tau_2$ cannot be proved empty, and, therefore, the restriction $(\tau_1 \vee \tau_2)_{|\Xi}$ returns a non empty type (for simplicity, the *null* type is returned, but any other non empty type could be returned as well). A similar reasoning applies to the case of record types.

In the following we show some examples of application of the restriction operator.

For all types τ, $\tau_{|\emptyset} = \tau$.

If τ_1 is the type s.t. $\tau_1 = \langle f{:}\tau_2\rangle$, $\tau_2 = \langle g{:}\tau_1, h{:}\mathbf{0}\rangle$, then $\tau_{1|\{\tau_1\}} = \langle f{:}\langle h{:}\mathbf{0}\rangle\rangle$, $\tau_{2|\{\tau_1\}} = \langle h{:}\mathbf{0}\rangle$, and $\tau_{1|\{\tau_2\}} = \langle\rangle$.

If τ_3 is the type s.t. $\tau_3 = \tau_4 \vee \mathbf{0}, \tau_4 = \langle f{:}\tau_3\rangle$, then $\tau_{3|\{\tau_3\}} = \langle\rangle \vee \mathbf{0}$, and $\tau_{4|\{\tau_3\}} = \langle\rangle$.

Lemma 4. *If* $[\![\tau_{|\Xi}]\!] = \emptyset$, *then* $[\![\tau_{|\Xi\cup\{\tau\}}]\!] = \emptyset$.

Proof. It is sufficient to prove that if $v \in \tau_{|\Xi\cup\{\tau\}}$, then there exists $v' \in \tau_{|\Xi}$. The value v' corresponds to $ext(v, \tau, \Xi, \tau, v)$, where $ext(v, \tau, \Xi, \tau', v')$ is coinductively defined as follows:

- $ext(v, \tau, \Xi, \tau', v') = v$, if $\tau \in \{null, int\}$;
- $ext(v, \tau_1 \vee \tau_2, \Xi, \tau, v') = ext(v, \tau_1, \Xi, \tau, v')$, if $\tau_1, \tau_2 \notin \Xi, v \in \tau_{1|\Xi\cup\{\tau\}}$
- $ext(v, \tau_1 \vee \tau_2, \Xi, \tau, v') = ext(v, \tau_2, \Xi, \tau, v')$, if $\tau_1, \tau_2 \notin \Xi$, not $v \in \tau_{1|\Xi\cup\{\tau\}}$, and $v \in \tau_{2|\Xi\cup\{\tau\}}$
- $ext(v, \tau_1 \vee \tau_2, \Xi, \tau, v') = null$ if $\tau_1 \in \Xi$ or $\tau_2 \in \Xi$
- $ext(v, \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle, \Xi, \tau, v') =$
  $\langle f_i \mapsto ext(v.f_i, \tau_i, \Xi, \tau, v') \mid 1 \le i \le n, \tau_i \notin \Xi \cup \{\tau\}\rangle \cup$
  $\langle f_i \mapsto ext(v', \tau, \Xi, \tau, v') \mid 1 \le i \le n, \tau_i = \tau\rangle$

The proof can be concluded by proving by coinduction on the definition of value membership that if $v \in \tau_{|\Xi\cup\tau'}$ and $v' \in \tau'_{|\Xi\cup\tau'}$, then $ext(v, \tau, \Xi, \tau', v') \in \tau_{|\Xi}$.

5 A Sound and Complete Inference System

In this section we define a system of coinductive subtyping rules and prove that it is sound and complete with respect to the definition of semantic subtyping given in Section 4.

Remark : Unless explicitly stated, in the rest of the section we only consider regular and contractive types.

5.1 Type Normalization

The problem of defining a decision procedure for subtyping becomes simpler if types are first normalized; such a normalization simplifies empty types, and is driven by the following laws:

$$\tau \vee \mathbf{0} = \mathbf{0} \vee \tau = \tau \qquad \langle \ldots f{:}\mathbf{0} \ldots\rangle = \mathbf{0}$$

This normalization needs to be performed only once, before deciding subtyping; the subtyping rules, and the derived subtyping algorithm preserve this type normalization, hence no further normalization steps are required.

We use the notation $\tau_1 \triangleright \tau_2$ to indicate that type τ_1 is normalized to type τ_2; for instance, the judgment $(int \vee \mathbf{0}) \vee (\mathbf{0} \vee int) \triangleright int \vee int$ holds. To see a more involved example, let us consider the regular type defined by $\tau = \mathbf{0} \vee \langle f{:}\tau, g{:}\mathbf{0} \vee \mathbf{0}\rangle$; then, $\tau \triangleright \mathbf{0}$ holds.

Normalization requires a decision procedure for testing emptiness of types; non-emptiness is naturally specified by the coinductive rules in Figure 3.

Clearly, the primitive types *int* and *null* are not empty. A union type $\tau_1 \vee \tau_2$ is not empty if at least one between τ_1 and τ_2 is not empty. A record type is

$$\frac{}{int \not\cong \emptyset} \qquad \frac{}{null \not\cong \emptyset} \qquad \frac{\tau_i \not\cong \emptyset}{\tau_1 \vee \tau_2 \not\cong \emptyset}\ i \in 1..2 \qquad \frac{\tau_1 \not\cong \emptyset, \ldots, \tau_n \not\cong \emptyset}{\langle f{:}\tau_1, \ldots, f_n{:}\tau_n \rangle \not\cong \emptyset}$$

Fig. 3. Non-emptiness of types

not empty if all types of its fields are not empty. Note that the rules must be interpreted coinductively because in some cases infinite proof trees are required. Consider for instance the type defined by $\tau = \langle f{:}\tau \rangle$; if $v = \langle f \mapsto v \rangle$, then $v \in \tau$, therefore $\tau \not\cong \emptyset$ must hold. This can be proved by an infinite proof tree obtained by repeatedly applying the rule for records.

Soundness and Completeness of the Judgment $\tau \not\cong \emptyset$. Before proving that the judgment $\tau \not\cong \emptyset$ is sound and complete w.r.t. the predicate $[\![\tau]\!] \neq \emptyset$, we illustrate the new proof technique we propose and use; this is the same technique that will be adopted for proving soundness and completeness of the subtyping rules.

Soundness and completeness are expressed by the implications $\tau \not\cong \emptyset \Rightarrow [\![\tau]\!] \neq \emptyset$, and $[\![\tau]\!] \neq \emptyset \Rightarrow \tau \not\cong \emptyset$, respectively.

Since $\tau \not\cong \emptyset$ is defined coinductively, completeness can be proved in a standard way by coinduction on the rules defining $\tau \not\cong \emptyset$, as explained in Section 3. Unfortunately, the same technique cannot be adopted for proving soundness (hence, for coinductive systems the difficult direction to prove is soundness, whereas for inductive systems is completeness).

To prove soundness we first consider the equivalent implication ($[\![\tau]\!] = \emptyset \Rightarrow \tau \not\cong \emptyset$ does not hold) corresponding to a proof by contradiction; then we observe that this implication can be proved if we split the implication in the following two:

$$[\![\tau]\!] = \emptyset \Rightarrow \tau \cong \emptyset \Rightarrow (\tau \not\cong \emptyset \text{ does not hold}) \tag{1}$$

where $\tau \cong \emptyset$ is the complement judgment of $\tau \not\cong \emptyset$ corresponding to testing type emptiness. Now it seems we get stuck because if $\tau \cong \emptyset$ is defined inductively, then the implication on the left hand side cannot be proved easily, whereas if $\tau \cong \emptyset$ is defined coinductively, the same consideration applies for the implication on the right hand side.

However, we still can have the cake and eat it too if we are able to define the judgment $\tau \not\cong \emptyset$ with an inference system whose inductive and coinductive interpretation coincide (hence, there exists a unique fixed point which is both the least and the greatest). A sufficient condition for this is that all proof trees of the inference system are finite.

The complement judgment we are looking for is defined in Figure 4. We use thin lines in the rules because it is sufficient to interpret the system inductively to define the judgment, however if we interpret the rules coinductively we get the same definition of emptiness for regular and contractive types.

Note that the only role of the set of types Ξ is to force the inductive and coinductive interpretation of the rules in Figure 4 to coincide, as proved in Lemma 5.

$$\frac{}{\Xi \vdash \mathbf{0} \cong \emptyset} \qquad \frac{\Xi \vdash \tau_1 \cong \emptyset \quad \Xi \vdash \tau_2 \cong \emptyset}{\Xi \vdash \tau_1 \vee \tau_2 \cong \emptyset} \qquad \frac{\Xi \cup \{\tau\} \vdash \tau_i \cong \emptyset}{\Xi \vdash \tau \cong \emptyset} \begin{array}{l} \tau = \langle f{:}\tau_1, \ldots, f_n{:}\tau_n \rangle \\ \tau \notin \Xi \\ i \in \{1, \ldots, n\} \end{array}$$

Fig. 4. Emptiness of types

To distinguish between the two interpretations we use the notations $\Xi \vdash \tau \cong \emptyset$ and $\Xi \Vdash \tau \cong \emptyset$ to indicate judgments corresponding to the inductive and coinductive interpretation of the rules, respectively.

Lemma 5. *$\Xi \Vdash \tau \cong \emptyset$ implies $\Xi \vdash \tau \cong \emptyset$.*

Proof. A direct consequence of the fact that τ is regular (hence Ξ cannot grow indefinitely) and contractive (hence the rule for union can be applied consecutively only a finite number of times).

We can now prove the two implications in (1) on the left and right side, respectively. The following two lemmas with Lemma 5 prove the soundness of $\tau \ncong \emptyset$.

In Lemma 6 two different hypotheses are needed to ensure that the claim holds. For instance, $[\![\langle f{:}\mathbf{0}\rangle]\!] = \emptyset$, but $\{\langle f{:}\mathbf{0}\rangle\} \Vdash \langle f{:}\mathbf{0}\rangle \cong \emptyset$ does not hold because of the side condition of the rule for record types; in this case the hypothesis $\tau \notin \Xi$ is not verified, but $[\![\tau_{|\Xi}]\!] = \emptyset$ holds. As another example, if τ is s.t. $\tau = \langle f{:}\langle g{:}\tau, h{:}\mathbf{0}\rangle\rangle$, then $[\![\tau]\!] = \emptyset$, but $\{\langle g{:}\tau, h{:}\mathbf{0}\rangle\} \Vdash \tau \cong \emptyset$ does not hold (again, because of the side condition of the rule for record types). In this case the hypothesis $\tau \notin \Xi$ is verified, but $[\![\tau_{|\Xi}]\!] = \emptyset$ does not hold.

Lemma 6. *If $\tau \notin \Xi$, and $[\![\tau_{|\Xi}]\!] = \emptyset$, then $\Xi \Vdash \tau \cong \emptyset$.*

Proof. By coinduction on the rules for $\Xi \Vdash \tau \cong \emptyset$. We only show the interesting case for $\tau = \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle$. By Lemma 4 $[\![\tau_{|\Xi}]\!] = \emptyset$ implies $[\![\tau_{|\Xi\cup\{\tau\}}]\!] = \emptyset$.

Furthermore, if $[\![\tau_{|\Xi\cup\{\tau\}}]\!] = \emptyset$, then by *Lemma* 3 and definition of $\tau_{|\Xi\cup\{\tau\}}$ when τ is a record type, there exists $i \in \{1, \ldots, n\}$ s.t. $[\![\tau_{i|\Xi\cup\{\tau\}}]\!] = \emptyset$, and $\tau_i \notin \Xi \cup \{\tau\}$. Since $\tau \notin \Xi$ by hypothesis, we can conclude by coinduction and by using rule for record types.

Lemma 7. *If $\Xi \vdash \tau \cong \emptyset$, then $\tau \ncong \emptyset$ does not hold.*

Proof. Easy induction on the rules defining $\Xi \vdash \tau \cong \emptyset$.

Completeness of $\tau \ncong \emptyset$ can be easily proved by coinduction, as expected.

Lemma 8. *$[\![\tau]\!] \neq \emptyset$ implies $\tau \ncong \emptyset$.*

Proof. By coinduction on the rules for $\tau \ncong \emptyset$.

The following corollary simply derives the equivalence of $[\![\tau]\!] \neq \emptyset$ and $\tau \ncong \emptyset$ from the lemmas above; as a byproduct, we also get the equivalence of $[\![\tau]\!] = \emptyset$ and $\tau \cong \emptyset$.

Corollary 1

1. $\tau \ncong \emptyset$ *if and only if* $[\![\tau]\!] \neq \emptyset$.
2. $\emptyset \vdash \tau \cong \emptyset$ *if and only if* $[\![\tau]\!] = \emptyset$.

Proof

1. soundness: Lemma 6 + Lemma 5 + Lemma 7; completeness: Lemma 8.
2. soundness: Lemma 8 + Lemma 7; completeness: Lemma 6 + Lemma 5 .

We are now ready to define type normalization. This is defined by the coinductive rules in Figure 5.

$$(\text{prim} \triangleright)\ \frac{}{\tau \triangleright \tau}\ \tau \in \{\mathbf{0}, null, int\} \qquad (\text{or} \triangleright)\ \frac{\tau_1 \triangleright \tau_1' \quad \tau_2 \triangleright \tau_2'}{\tau_1 \vee \tau_2 \triangleright \tau_1' \vee \tau_2'}\ \begin{matrix}\tau_1 \ncong \emptyset \\ \tau_2 \ncong \emptyset\end{matrix}$$

$$(\text{r-or} \triangleright)\ \frac{\tau_1 \triangleright \tau_1'}{\tau_1 \vee \tau_2 \triangleright \tau_1'}\ \emptyset \vdash \tau_2 \cong \emptyset \qquad (\text{l-or} \triangleright)\ \frac{\tau_2 \triangleright \tau_2'}{\tau_1 \vee \tau_2 \triangleright \tau_2'}\ \emptyset \vdash \tau_1 \cong \emptyset$$

$$(\text{rec} \triangleright)\ \frac{\tau_1 \triangleright \tau_1', \ldots, \tau_n \triangleright \tau_n'}{\langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle \triangleright \langle f_1{:}\tau_1', \ldots, f_n{:}\tau_n'\rangle}\ \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle \ncong \emptyset$$

$$(\text{e-rec} \triangleright)\ \frac{}{\langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle \triangleright \mathbf{0}}\ \emptyset \vdash \langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle \cong \emptyset$$

Fig. 5. Type normalization

The empty and primitive types normalize to themselves, whereas normalizing a union type corresponds to coinductively normalizing its two subtypes, if they are both non-empty, or just one in case the other is empty. For record types two cases have to be distinguished: if $\langle f_1{:}\tau_1, \ldots, f_n{:}\tau_n\rangle \ncong \emptyset$ holds (that is, $\tau_i \ncong \emptyset$ holds for all $i \in \{1, \ldots, n\}$), then each subtype can be coinductively normalized to get the final type $\langle f_1{:}\tau_1', \ldots, f_n{:}\tau_n'\rangle$. Otherwise the type normalizes to the empty set.

The following claims show that the normalization relation $\triangleright$ is actually a total function, and that it preserves type interpretation.

Lemma 9. *If* $\tau \ncong \emptyset$ *does not hold, then* $\tau \triangleright \mathbf{0}$.

Proof. See the extended version [2].

Theorem 2. *For all* τ *there exists a unique type* τ' *such that* $\tau \triangleright \tau'$.

Proof. The proof uses Lemma 9 and Proposition 1. See the extended version [2].

Lemma 10. *If* $\tau \triangleright \tau'$, *and* $\tau' \in \{\mathbf{0}, null, int\}$, *then* $[\![\tau]\!] = [\![\tau']\!]$.

Proof. The proof uses Corollary 1. See the extended version [2].

Lemma 11. *If $\tau \triangleright \tau'_1 \vee \tau'_2$, and $v \in \tau$, then there exist τ_1, τ_2 s.t. $\tau = \tau_1 \vee \tau_2$, and ($\tau_1 \triangleright \tau'_1$, and $v \in \tau_1$, or $\tau_2 \triangleright \tau'_2$, and $v \in \tau_2$).*

Proof. The proof uses Corollary 1. See the extended version [2].

Theorem 3. *For all τ, τ', if $\tau \triangleright \tau'$, then $[\![\tau]\!] = [\![\tau']\!]$.*

Proof. The proof uses Lemma 10 and Lemma 11. See the extended version [2].

Corollary 2. *$\tau \triangleright \mathbf{0}$ if and only if $[\![\tau]\!] = \emptyset$.*

Proof. $\tau \triangleright \mathbf{0} \Rightarrow [\![\tau]\!] = \emptyset$ can be derived directly from Theorem 3. For the other direction, if $[\![\tau]\!] = \emptyset$, then $\tau \not\cong \emptyset$ does not hold by Corollary 1, therefore we can derive $\tau \triangleright \mathbf{0}$ directly from Lemma 9.

5.2 Subtyping Rules

In this section we define the rules for subtyping. In the rest of the paper we assume that all types are normalized (besides being regular and contractive). Subtyping rules are based on the identity between sets $A \subseteq B \cup C \Leftrightarrow A \setminus B \subseteq C$.

For instance, if one would like to prove that

$$\langle f{:}\mathit{null} \vee \mathit{int}\rangle \leq \langle f{:}\mathit{null}\rangle \vee \langle f{:}\mathit{int}\rangle \vee \mathit{int}$$

holds, then one can prove that $\langle f{:}\mathit{null} \vee \mathit{int}\rangle \setminus \langle f{:}\mathit{null}\rangle \leq \langle f{:}\mathit{int}\rangle \vee \mathit{int}$ holds, which in turn holds if $(\langle f{:}\mathit{null} \vee \mathit{int}\rangle \setminus \langle f{:}\mathit{null}\rangle) \setminus \langle f{:}\mathit{int}\rangle \leq \mathit{int}$ holds.

Now $\langle f{:}\mathit{null} \vee \mathit{int}\rangle \setminus \langle f{:}\mathit{null}\rangle = \langle f{:}(\mathit{null} \vee \mathit{int}) \setminus \mathit{null}\rangle = \langle f{:}\mathit{int}\rangle$, and $\langle f{:}\mathit{int}\rangle \setminus \langle f{:}\mathit{int}\rangle = \mathbf{0}$, hence we can conclude the proof because trivially $\mathbf{0} \leq \mathit{int}$ holds.

Unfortunately, types are not closed w.r.t. complement. Even though this could be formally proved[2], for space reasons we only provides an informal argumentation.

Let us consider the two types τ and τ' introduced in Section 1:

$$\tau = \mathit{null} \vee \langle \mathit{el}{:}\mathit{int}, \mathit{nx}{:}\tau\rangle \qquad \tau' = \langle \mathit{el}{:}\mathit{int}, \mathit{nx}{:}\tau'\rangle$$

Since $[\![\tau]\!]$ contains all values corresponding to either finite or infinite lists, while $[\![\tau']\!]$ contains all values corresponding just to infinite lists, we deduce that $[\![\tau]\!] \setminus [\![\tau']\!]$ is the set of all values corresponding just to finite lists. If we assume that types are closed w.r.t. complement, then there must exist a regular and contractive type τ'' s.t. $[\![\tau'']\!] = [\![\tau]\!] \setminus [\![\tau']\!]$, but no regular contractive type can have a coinductive interpretation corresponding to the set of all values corresponding to finite lists, because such a set is not a complete metric space for the standard metric on infinite trees.

[2] The proof relies on the property that for all regular and contractive types τ, $[\![\tau]\!]$ is a complete metric space for the standard metric on infinite trees.

Given this negative result, we have to compute complement lazily, and extend the syntax of types to introduce the complement[3] type constructor, denoted by $-$. Note that while $-$ is a type constructor, $\setminus$ denotes an operation that given two types τ_1 and τ_2, returns a new type.

For instance $int \setminus int = \mathbf{0}$, and $int \setminus \langle f{:}int \rangle = int$. However, when both types are records the type returned by the complement is in general a extended type containing the type constructor $-$. For instance, if we assume that fields f, g, and h are all distinct, then $\langle f{:}\tau_1, g{:}\tau_2 \rangle \setminus \langle f{:}\tau_3, h{:}\tau_4 \rangle$ returns the extended type

$$\langle f{:}\tau_1 - \tau_3, g{:}\tau_2 \rangle \vee \langle f{:}\tau_1, g{:}\tau_2, h?{:} - \tau_4 \rangle$$

where $h?$ denotes an optional field: record type $\langle h?{:} - \tau_4 \rangle$ contains record values which either do not have field h, or have field h with a value v s.t. $v \notin [\![\tau_4]\!]$.

The reader can verify that

$$[\![\langle f{:}\tau_1, g{:}\tau_2 \rangle]\!] \setminus [\![\langle f{:}\tau_3, h{:}\tau_4 \rangle]\!] = [\![\langle f{:}\tau_1 - \tau_3, g{:}\tau_2 \rangle \vee \langle f{:}\tau_1, g{:}\tau_2, h?{:} - \tau_4 \rangle]\!] .$$

Indeed $v \in [\![\langle f{:}\tau_1, g{:}\tau_2 \rangle]\!]$ and $v \notin [\![\langle f{:}\tau_3, h{:}\tau_4 \rangle]\!]$ if and only if v has the two fields f and g, where g is always associated with a value in $[\![\tau_2]\!]$, whereas f is associated either with a value in $[\![\tau_1]\!]$, but not in $[\![\tau_3]\!]$, or with a value in $[\![\tau_1]\!]$, but then either v does not have field h, or it has field h associated with a value not in $[\![\tau_4]\!]$. The definition of $\setminus$ for record types is the generalization of the following identity between sets:

$$(A_1 \times \ldots \times A_n) \setminus (B_1 \times \ldots \times B_n) =$$
$$(A_1 \setminus B_1) \times A_2 \times \ldots \times A_n \cup \ldots \cup A_1 \times \ldots \times A_{n-1} \times (A_n \setminus B_n).$$

Extended types are defined in Figure 6; note that the two definitions are stratified: first types are defined coinductively, then extended types are inductively defined on top of types.

$$\pi ::= \tau \mid \langle f_1{:}\rho_1, \ldots, f_n{:}\rho_n, f'_1?{:}\varrho_1, \ldots, f_k?{:}\varrho_k \rangle \mid \pi_1 \vee \pi_2$$
$$\varsigma ::= \rho \mid \varrho \qquad \rho ::= \tau \mid \rho - \tau \quad \varrho ::= -\tau \mid \varrho - \tau$$

Fig. 6. Extended types

The meta-variable ρ corresponds to an extended type that can be associated with a non optional field of an extended record type, and has shape $((\tau_0 - \tau_1) - \ldots \tau_k)$, while the meta-variable ϱ corresponds to an extended type that can be associated with an optional field of an extended record type, and has shape $((-\tau_0 - \tau_1) - \ldots \tau_k)$; finally, the meta-variables ς has been introduced just for practical reasons to avoid useless duplication for all cases where the expected type can be either ρ or ϱ.

Interpretation of extended types is defined in Figure 7 by a corresponding extended judgment for membership $v \in_e \pi$ and $v \in_e \varsigma$ (note that values are not

[3] The constructor is overloaded since it denotes both unary absolute complement, and binary relative complement.

extended); as happens for extended types, the definitions of $v \in_e \pi$ and $v \in_e \varsigma$ are stratified over the definition of $v \in \tau$: first $v \in \tau$ is defined coinductively, then $v \in_e \pi$ and $v \in_e \varsigma$ are inductively defined on top of $v \in \tau$.

$$(\text{emb } \in_e)\frac{}{v \in_e \tau}\ v \in \tau \qquad (\text{l-or } \in_e)\frac{v \in_e \pi_1}{v \in_e \pi_1 \vee \pi_2}\ ext(\pi_1 \vee \pi_2) \qquad (\text{r-or } \in_e)\frac{v \in_e \pi_2}{v \in_e \pi_1 \vee \pi_2}\ ext(\pi_1 \vee \pi_2)$$

$$(\text{rec } \in_e)\frac{\begin{array}{c} v(f_i) \in_e \rho_i\ \forall i \in \{1,\ldots,n\} \\ f'_j \in dom(v) \Rightarrow v(f'_j) \in_e \varrho_j\ \forall j \in \{1,\ldots,k\} \end{array}}{v \in_e \pi}\ \begin{array}{l} \pi = \langle f_1{:}\rho_1, \ldots, f_n{:}\rho_n, \\ \quad f'_1?{:}\varrho_1, \ldots, f'_k?{:}\varrho_k \rangle \\ ext(\pi) \\ \{f_1,\ldots,f_n\} \subseteq dom(v) \end{array}$$

$$(\text{comp})\frac{v \in_e \varsigma}{v \in_e \varsigma - \tau}\ v \notin \tau \qquad (\text{a-comp})\frac{}{v \in_e -\tau}\ v \notin \tau$$

Fig. 7. Value membership for extended types

Rules defining $v \in_e \pi$ are straightforward. We use the auxiliary predicate *ext* on extended types s.t. $ext(\pi)$ holds if and only if π is a proper extended type, that is, there is no type τ s.t. $\tau = \pi$. Such a predicate is used to avoid rule (emb $\in_e$) to overlap the other rules.

The complement operator is defined in Figure 8.

$$\tau \setminus \tau = \mathbf{0} \qquad \pi \setminus \mathbf{0} = \pi \qquad \mathbf{0} \setminus \tau = \mathbf{0}$$

$$\tau \setminus \tau' = \tau \text{ if } \tau \neq \tau', \tau \in \{int, null\} \text{ and } \tau' \neq \tau_1 \vee \tau_2$$

$$\pi \setminus \tau = \pi \text{ if } \pi = \langle \ldots \rangle \text{ and } \tau \in \{int, null\}$$

$$\pi \setminus \tau = (\bigvee_{f \in dom(\pi) \cap dom(\tau)} \pi -_f \tau) \bigvee (\bigvee_{f \in dom(\tau) \setminus dom(\pi)} \pi \sim_f \tau) \text{ if } \pi, \tau = \langle \ldots \rangle$$

$$\begin{array}{ll} \text{where} & \pi -_f \tau = \pi[f{:}\rho - \tau'] \text{ if } \pi = \langle \ldots f{:}\rho \ldots \rangle\ \tau = \langle \ldots f{:}\tau' \ldots \rangle \\ & \pi -_f \tau = \pi[f?{:}\varrho - \tau'] \text{ if } \pi = \langle \ldots f?{:}\varrho \ldots \rangle\ \tau = \langle \ldots f{:}\tau' \ldots \rangle \\ & \pi \sim_f \tau = \pi[f?{:} - \tau'] \text{ if } \tau = \langle \ldots f{:}\tau' \ldots \rangle \end{array}$$

Fig. 8. Complement operator

The complement needs to be computed between an extended type π and a type τ; furthermore, both types cannot be union types except for the two corner cases $\tau \setminus \tau$ and $\mathbf{0} \setminus \tau$ (anyway, as we will see, two subtyping rules allow elimination of union types by splitting them, so that the complement operator can eventually be used). All cases are straightforward, except for the last case involving two record types which has been already explained by an example. In this case, the type returned by $\pi \setminus \tau$ is always a union of records, where the number n of records equals the number of fields contained in τ. Note that if $n = 0$, then the returned type is $\mathbf{0}$; for instance, $\langle f{:}int \rangle \setminus \langle\ \rangle = \mathbf{0}$. If $n = 1$, then a single record is returned: for instance $\langle\ \rangle \setminus \langle f{:}int \rangle = \langle f?{:} - int \rangle$, or $\langle f{:}null \rangle \setminus \langle f{:}int \rangle = \langle f{:}null - int \rangle$.

Recall that the notation $\pi[f{:}\rho-\tau']$ (and, equivalently, $\pi[f?{:}\varrho-\tau']$ and $\pi[f?{:}-\tau']$) denotes the record type updated by the association $f{:}\rho-\tau'$ (or $f?{:}\varrho-\tau'$ and $f?{:}-\tau'$, respectively); note that in the sole case of the definition of $\pi \sim_f \tau$, this update is actually an addition since by definition $f \notin dom(\pi)$.

The following lemmas are instrumental to prove the soundness and completeness of the subtyping rules.

Lemma 12. *If $\pi \setminus \tau = \pi'$, then $[\![\pi]\!] \setminus [\![\tau]\!] = [\![\pi']\!]$.*

Proof. Routine verification.

Lemma 13. *π is a record type s.t. $[\![\pi]\!] = \emptyset$ if and only if there exist f, ρ, and τ s.t. π has shape $\langle \ldots f{:}\rho-\tau \ldots\rangle$, and $[\![\rho-\tau]\!] = \emptyset$.*

Proof. It suffices to notice that by definition of the complement operator of Figure 8, all types τ (hence, not extended) occurring in π comes from non-extended record types which have been normalized, hence cannot be empty by Corollary 2; furthermore, a record type π cannot be empty because of an optional field f, since π can always contain all record values that do not have field f.

The subtyping rules are defined in Figure 9.

$$(\text{empty} \le)\ \frac{}{\mathbf{0} \le \Xi} \qquad (\text{left-or} \le)\ \frac{\pi_1 \le \Xi \quad \pi_2 \le \Xi}{\pi_1 \vee \pi_2 \le \Xi} \qquad (\text{r-or} \le)\ \frac{\pi \le \Xi \cup \{\tau_1, \tau_2\}}{\pi \le \Xi \cup \{\tau_1 \vee \tau_2\}}\ \tau_1 \vee \tau_2 \notin \Xi$$

$$(\text{comp} \le)\ \frac{\pi' \le \Xi}{\pi \le \Xi \cup \{\tau\}}\ \begin{array}{l}\tau \notin \Xi \\ \pi \setminus \tau = \pi'\end{array} \qquad (\text{rec} \le)\ \frac{\tau' \le \Xi}{\langle \ldots f{:}\rho - \tau \ldots\rangle \le \emptyset}\ \rho - \tau \rightsquigarrow \tau' - \Xi$$

Fig. 9. Subtyping rules

The subtyping judgment has shape $\pi \le \Xi$, where π is an extended type, and Ξ is a finite set of non-extended types $\{\tau_1, \ldots, \tau_n\}$ corresponding to the union $\tau_1 \vee \ldots \vee \tau_n$ (which collapses to $\mathbf{0}$ when $n = 0$, and to τ_1 when $n = 1$). The set Ξ is required for ensuring termination: union types in Ξ are lazily split and reinserted in Ξ to avoid unbounded growth of union types with duplicate types.

Rules (left-or $\le$) and (r-or $\le$) are applied for splitting and eliminating union types on both sides (this can always achieved with a finite number of applications of the rules by virtue of contractivity); then rule (comp $\le$) removes types from the set Ξ. When finally the set Ξ is empty we get the judgment $\pi \le \emptyset$: if $\pi = \mathbf{0}$, then we can conclude by rule (empty $\le$); if $\pi \in \{null, int\}$, then no rule can be applied and the judgment fails as expected; if π is a record type, then rule (rec $\le$) tries to find a non optional field of type $\rho - \tau$, and to check whether such a type is empty.

The side condition in rule (rec $\le$) is needed for normalizing the types of non optional fields having shape $\rho - \tau$: it transforms the type $(\ldots(\tau - \tau_1)\ldots) - \tau_n)$

in the pair $\tau - (\{\tau_1\} \cup \ldots \{\tau_n\})$ (see the straightforward inductive definition in Figure 10). This is essential for avoiding unbounded growth of union types (and, consequently of types having shape $\rho - \tau$) which may have duplicate types; for instance, this would happen for the judgment $\tau_1 \leq \{\tau_2\}$, where $\tau_1 = \langle f{:}\tau_1, g{:}int\rangle$ and $\tau_2 = \langle f{:}\tau_2 \vee \tau_2\rangle$.

Splitting is performed lazily for two reasons: by running our prototype implementation on numerous tests, we have realized that splitting all union types contained in τ_1 and τ_2 before deciding $\tau_1 \leq \tau_2$ (eager strategy) is less efficient than a lazy strategy; anyway, when the eager strategy is followed, splitting has to be performed repeatedly on the types $\pi \setminus \tau$ generated by rule (comp $\leq$).

$$\frac{\rho - \tau \rightsquigarrow \tau'' - \Xi}{(\rho - \tau) - \tau' \rightsquigarrow \tau'' - (\Xi \cup \{\tau'\})} \qquad \frac{}{\tau - \tau' \rightsquigarrow \tau - \{\tau'\}}$$

Fig. 10. Normalization of $\rho - \tau$

The following two lemmas are instrumental to the proofs of soundness and completeness of the subtyping rules, and can be easily proved by induction on the types $\rho - \tau$.

Lemma 14. *For all ρ, τ, there exist unique τ', Ξ s.t. $\rho - \tau \rightsquigarrow \tau' - \Xi$ holds.*

Lemma 15. *If $\rho - \tau \rightsquigarrow \tau' - \{\tau_1, \ldots, \tau_n\}$, then $[\![\rho - \tau]\!] = [\![\tau' - (\tau_1 \vee \ldots \vee \tau_n)]\!]$.*

Proofs of Soundness and Completeness of the Subtyping Rules. We adopt the same technique used for proving the soundness of the judgment $\tau \not\cong \emptyset$. Therefore first we have to define the complement judgment (see Figure 11).

As for the case of the negation of the $\tau \not\cong \emptyset$ judgment, the standard interpretation of the rules is inductive (thin lines), but Lemma 16 shows that the use of the set Ψ of extended types forces the inductive (judgment $\Psi \vdash \pi \not\leq \Xi$) and coinductive (judgment $\Psi \Vdash \pi \not\leq \Xi$) interpretation of the rules to coincide (when we restrict judgments $\Psi \vdash \tau \not\leq \Xi$ to finite sets Ξ).

Lemma 16. *For all finite sets Ξ, $\Psi \Vdash \tau \not\leq \Xi$ implies $\Psi \vdash \tau \not\leq \Xi$.*

Proof. It suffices to prove that any proof tree for $\Psi \Vdash \pi \not\leq \Xi$ must be finite. To do that, we first observe that, given π and Ξ, the cardinality of Ψ in the judgments of the proof tree for $\Psi \Vdash \pi \not\leq \Xi$ must be bounded. This can be proved by firstly observing that Ψ contains only the record types that appear in the left-hand-side of $\not\leq$ in the judgments, that such record types have fields ranging over a finite set (since we assume that initially τ and all types in Ξ are regular, and the set Ξ is finite), and that for all types of shape $((\tau_0 - \tau_1) - \ldots \tau_k)$ associated with non optional fields and generated by rule (comp $\not\leq$), τ_0 corresponds to a subterm of the initial type π, whereas $\tau_1, \ldots, \tau_k$ correspond to subterms of types contained in the initial set Ξ.

$$(\text{prim } \not\leq)\frac{}{\Psi \vdash \tau \not\leq \emptyset}\ {\scriptstyle \tau \in \{null, int\}} \qquad (\text{l-l-or } \not\leq)\frac{\Psi \vdash \pi_1 \not\leq \Xi}{\Psi \vdash \pi_1 \vee \pi_2 \not\leq \Xi} \qquad (\text{r-l-or } \not\leq)\frac{\Psi \vdash \pi_2 \not\leq \Xi}{\Psi \vdash \pi_1 \vee \pi_2 \not\leq \Xi}$$

$$(\text{comp } \not\leq)\frac{\Psi \vdash \pi' \not\leq \Xi}{\Psi \vdash \pi \not\leq \Xi \cup \{\tau\}}\ {\scriptstyle \tau \notin \Xi \atop \pi \setminus \tau = \pi'} \qquad (\text{r-or } \not\leq)\frac{\Psi \vdash \pi \not\leq \Xi \cup \{\tau_1, \tau_2\}}{\Psi \vdash \pi \not\leq \Xi \cup \{\tau_1 \vee \tau_2\}}\ {\scriptstyle \tau_1 \vee \tau_2 \notin \Xi}$$

$$(\text{rec } \not\leq)\frac{\forall f \in dom(\pi)\ \pi(f) = \rho - \tau \rightsquigarrow \tau' - \Xi \Rightarrow \Psi \cup \{\pi\} \vdash \tau' \not\leq \Xi}{\Psi \vdash \pi \not\leq \emptyset}\ {\scriptstyle \pi = \langle \ldots \rangle \atop \pi \notin \Psi}$$

Fig. 11. Negation of subtyping

To prove that all proof trees are finite, we introduce the following measure on the judgments of shape $\Psi \Vdash \pi \not\leq \Xi$ defined on a Noetherian order, and show that for every rule of Figure 11 the measure of its premises is always strictly less than the measure of its consequence.

If B denotes an upper bound of the size of Ψ, then the measure of the judgment $\Psi \Vdash \pi \not\leq \Xi$ is defined by the quadruple $(B - |\Psi|, max_\vee(\Xi), |\Xi|, max_\vee(\pi))$, where $|\_|$ denotes cardinality, $max_\vee(\pi)$ returns the length of the maximum path from the root of π containing only union type constructors (this is always well-defined by contractivity), and $max_\vee(\Xi) = \sum_{\tau \in \Xi} max_\vee(\tau)$. If we consider the standard lexicographic order (where the leftmost value is the most significant one) on quadruples, then we obtain a Noetherian order, since trivially $max_\vee(\Xi) \geq 0, |\Xi| \geq 0$, $max_\vee(\pi) \geq 0$ and $B - |\Psi| > 0$ by virtue of the boundedness of Ψ sets.

We now prove that the measure of the premises of every rule is always strictly less than the measure of its consequence.

Rule (comp $\not\leq$): let (n_1, n_2, n_3, n_4) be the measure value for the consequence, then the value for the premise is $(n_1, n_2, n_3 - 1, n'_4)$, and $(n_1, n_2, n_3 - 1, n'_4) < (n_1, n_2, n_3, n_4)$;

Rules (l-l-or $\not\leq$) and (r-l-or $\not\leq$): let (n_1, n_2, n_3, n_4) be the measure value for the consequence, then the value for the premise is $(n_1, n_2, n_3, n_4 - 1)$, and $(n_1, n_2, n_3, n_4 - 1) < (n_1, n_2, n_3, n_4)$;

Rule (r-or $\not\leq$): let (n_1, n_2, n_3, n_4) be the measure value for the consequence, then the value for the premise is $(n_1, n_2 - 1, n'_3, n_4)$, and $(n_1, n_2 - 1, n'_3, n_4) < (n_1, n_2, n_3, n_4)$;

Rule (rec $\not\leq$): let (n_1, n_2, n_3, n_4) be the measure value for the consequence, then the value for any premise is $(n_1 - 1, n'_2, n'_3, n'_4)$, and $(n_1 - 1, n'_2, n'_3, n'_4) < (n_1, n_2, n_3, n_4)$.

Soundness is split into two implications, the first proved by coinduction, the second by induction.

Lemma 17. *Let Ψ be a set of extended types s.t. for all $\pi' \in \Psi$, π' is a record type having shape $\langle \ldots f{:}\rho - \tau \ldots \rangle$. Then $\pi \notin \Psi$ and $[\![\pi]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$ imply $\Psi \Vdash \pi \not\leq \{\tau_1, \ldots, \tau_n\}$.*

Proof. By coinduction on the rules of Figure 11 and case analysis on π.

If $\pi = \mathbf{0}$, then $[\![\pi]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$ does not hold, therefore the implication vacuously holds.

If $\pi = \pi_1 \vee \pi_2$, then $[\![\pi]\!] = [\![\pi_1]\!] \cup [\![\pi_2]\!]$, therefore if $[\![\pi]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$, then either $[\![\pi_1]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$ or $[\![\pi_2]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$, hence by coinduction we can apply either rule (l-l-or $\not\leq$) or (r-l-or $\not\leq$) and conclude.

For the remaining cases we distinguish two subcases: either $\Xi \neq \emptyset$ or $\Xi = \emptyset$.

If $\Xi \neq \emptyset$ and $\pi \in \{null, int, \langle \ldots \rangle\}$, then, by coinduction, rule (r-or $\not\leq$) can be applied if $\tau_n = \tau' \vee \tau''$, because $[\![\tau_1 \vee \ldots \vee (\tau' \vee \tau'')]\!] = [\![\tau_1 \vee \ldots \vee \tau' \vee \tau'']\!]$; if τ_n is not a union type, then, by coinduction, rule (comp $\not\leq$) can be applied because there exists τ' s.t. $\pi \setminus \tau_n = \tau'$ (π and τ_n are not union), $[\![\tau']\!] = [\![\pi]\!] \setminus [\![\tau_n]\!]$ by Lemma 12, and $[\![\pi]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$ implies $[\![\pi]\!] \setminus [\![\tau_n]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_{n-1}]\!]$.

If $\Xi = \emptyset$ and $\pi \in \{null, int\}$, then we can easily conclude by coinduction and rule (prim $\not\leq$).

If $\Xi = \emptyset$ and π is a record type s.t. $[\![\pi]\!] \not\subseteq \emptyset$; if there is no f, ρ, and τ, s.t. π has shape $\langle \ldots f{:}\rho - \tau \ldots \rangle$, then we can conclude by coinduction and by applying rule (rec $\not\leq$) with no premises (note that the side condition $\pi \notin \Psi$ holds by hypothesis). Otherwise, by Lemma 13, for all f, ρ, and τ s.t. π has shape $\langle \ldots f{:}\rho - \tau \ldots \rangle$, we know that $[\![\rho - \tau]\!] \neq \emptyset$. Furthermore, by Lemma 14 there exist unique τ', Ξ s.t. $\rho - \tau \rightsquigarrow \tau' - \Xi$ holds, and by Lemma 15, if $\Xi = \{\tau'_1, \ldots, \tau'_k\}$, then $[\![\rho - \tau]\!] = [\![\tau' - (\tau'_1 \vee \ldots \vee \tau'_k)]\!]$, hence $[\![\tau' - (\tau'_1 \vee \ldots \vee \tau'_k)]\!] \neq \emptyset$ which implies $[\![\tau']\!] \not\subseteq [\![\tau'_1 \vee \ldots \vee \tau'_k]\!]$. Finally, if for all $\pi' \in \Psi$, π' is a record type having shape $\langle \ldots f'{:}\rho' - \tau'' \ldots \rangle$, then the same property holds for $\Psi \cup \{\pi\}$, and $\tau' \notin \Psi \cup \{\pi\}$ holds because τ' is not an extended type. Hence we can conclude by coinduction and rule (rec $\not\leq$).

Lemma 18. *If $\Psi \vdash \pi \not\leq \Xi$, then $\pi \leq \Xi$ does not hold.*

Proof. By induction on the rules defining $\Psi \vdash \pi \not\leq \Xi$. We detail the proof only for the most involved rule (rec $\not\leq$). If π is a record, then the only applicable rule for proving $\pi \leq \emptyset$ is (rec $\leq$). If rule (rec $\not\leq$) has no premises, then there is no field having type of shape $\rho - \tau$, hence rule (rec $\leq$) is not applicable. If rule (rec $\not\leq$) has premises, then for all fields of type $\rho - \tau$ we know that by Lemma 14 there exist exist unique τ', Ξ s.t. $\rho - \tau \rightsquigarrow \tau' - \Xi$, therefore by induction we deduce that $\tau' \leq \Xi$ does not hold, therefore rule (rec $\leq$) can never be applied, and, hence, $\pi \leq \emptyset$ does not hold.

Soundness trivially derives from the three previous lemmas.

Corollary 3 (Soundness). *If $\pi \leq \{\tau_1, \ldots, \tau_n\}$, then $[\![\pi]\!] \subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$.*

Proof. It suffices to show that $[\![\pi]\!] \not\subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$ implies that $\pi \leq \{\tau_1, \ldots, \tau_n\}$ does not hold. This can be proved directly by applying Lemma 17, Lemma 16, and Lemma 18.

Completeness throws no surprise and can be proved with a standard proof by coinduction on the subtyping rules.

Theorem 4 (Completeness). *If $[\![\pi]\!] \subseteq [\![\tau_1 \vee \ldots \vee \tau_n]\!]$, then $\pi \leq \{\tau_1, \ldots, \tau_n\}$ holds.*

Proof. The proof uses Lemma 12, Lemma 13, Lemma 14 and Lemma 15. See the extended version [2].

6 A Sound and Complete Algorithm

We have proved that the subtyping rules in Figure 9 are sound and complete w.r.t. the definition of semantic subtyping; however, such rules do not directly specify an algorithm for deciding semantic subtyping between coinductive types. In this section we show how it is possible to define a sound and complete algorithm implementing such rules.

The algorithm is specified by the following recursive function `subtype`, which is assumed to be invoked over normalized types; we omit the normalization function that can be derived from the Figure 5 (the interested reader can refer to the prototype implementation).

In order to decide whether π_1 is a subtype of π_2, function `subtype` must be called with $\Psi = \emptyset$, $\pi = \pi_1$, and $\Xi = \{\tau_2\}$.

```
// pre-condition: π and all types in Ξ are normalized
boolean subtype(Set<Pair<ExtType,Set<Type>>> Ψ, ExtType π,Set<Type> Ξ) {
    // rule (empty ≤)
    if(π==0)
        return true
    // termination condition
    if(∃(π′, Ξ′) ∈ Ψ s.t. π′==π && Ξ′ ⊆ Ξ)
        return true
    // rule (right-or ≤)
    while(∃τ1, τ2 s.t. τ1 ∨ τ2 ∈ Ξ)
        Ξ=(Ξ \ τ1 ∨ τ2) ∪ {τ1, τ2}
    // rule (left-or ≤)
    if(∃π1, π2 s.t. π==π1 ∨ π2)
        return subtype(Ψ, π1, Ξ) && subtype(Ψ, π2, Ξ)
    // rule (comp ≤)
    else if(∃τ ∈ Ξ) {
            π′=π \ τ
            Ξ′=Ξ \ {τ}
            return subtype(Ψ ∪ {(π, {τ})},π′,∅) ||
                   Ξ′!=∅ && subtype(Ψ ∪ {(π, Ξ)},π′,Ξ′)
        }
    // rule (rec ≤)
    else if (π==⟨...⟩){
            foreach f ∈ dom(π)
                if(∃ρ, τ s.t. π(f)==ρ − τ) {
                    ρ − τ ⇝ τ′ − Ξ′
                    if(subtype(Ψ,τ′,Ξ′))
                        return true
                }
            return false
        }
    else
        // int ≤ ∅ and null ≤ ∅ do not hold
        return false
}
```

The algorithm is derived from the rules in Figure 9, but also from the proof of soundness; in particular, Lemma 17, and Lemma 16 show that if $\Xi = \{\tau_1, \ldots, \tau_n\}$, $\tau = \tau_1 \vee \ldots \vee \tau_n$, and $[\![\pi]\!] \not\subseteq [\![\tau]\!]$, then failure of $\pi \leq \Xi$ is always finite (indeed, all proofs for $\emptyset \vdash \pi \not\leq \Xi$ are finite), whereas if $[\![\pi]\!] \subseteq [\![\tau]\!]$ holds, then the proof tree for $\pi \leq \Xi$ could be infinite; however, Lemma 16 shows that such a proof tree is always regular, hence we can use the complement of the side-condition $\pi \notin \Psi$ of rule (rec $\not\leq$) to ensure termination for $\pi \leq \Xi$.

However, the presented algorithm differs from the inference system of Figure 9 for several details:

Order of rule application: as expected, the order in which rules can be applied has been made deterministic. Rule (empty $\leq$) overlaps with (right-or $\leq$), and (comp $\leq$), and is tried first, for obvious efficiency reasons. Rule (right-or $\leq$) overlaps (besides (empty $\leq$)) only with (left-or $\leq$) (recall that $\pi \setminus \tau$ is only defined when both π and τ are not union types, hence rule (comp $\leq$) does not overlap with rules (right-or $\leq$) and (left-or $\leq$)), and it is applied first for efficiency reasons: were rule (left-or $\leq$) be applied first, the applications of rule (right-or $\leq$) would be uselessly duplicated for the two premises of (left-or $\leq$). Rules (left-or $\leq$), (comp $\leq$), and (rec $\leq$) do not overlap, therefore the order in which are considered is immaterial.

Termination condition: the termination condition used by the algorithm is an improvement of that used in rule (right-or $\not\leq$) for the definition of the judgment $\Psi \vdash \pi \not\leq \Xi$ (obviously the termination condition has to be complemented). First, such a termination condition is used for all rules defining $\pi \leq \Xi$ (except (empty $\leq$)), and not just for rule (rec $\leq$). When function `subtype` has to check whether $\pi \leq \Xi$ holds, it first verifies (unless $\pi = \mathbf{0}$) whether the set Ψ already contains a pair (π, Ξ') such that $\Xi' \subseteq \Xi$; this means that the algorithm is already checking whether $\pi \leq \Xi'$ holds (that is, there is a corresponding call to `subtype` on the stack) and if $\pi \leq \Xi'$ holds, then $\pi \leq \Xi$ holds as well; therefore, **true** can be returned. If $\pi \leq \Xi$ does not hold, then $\pi \leq \Xi'$ does not hold as well, therefore the corresponding call to `subtype` will eventually find a counter-example and return **false** as expected.

Finally, new pairs are inserted in Ψ when rule (comp $\leq$) is applied; this is the point where new types can be generated through the computation of $\pi \setminus \tau$ that can contain extended record types with fields having types of shape $\rho - \tau'$; only in this case the application of rule (rec $\leq$) can lead to a potentially infinite loop, as shown by the proof of Lemma 16 (recall that if π is a record that does not contain any field having type of shape $\rho - \tau'$, then rule (rec $\leq$) has no premises). In this way, we give the algorithm more chances to prune the proof tree, and, thus, to avoid combinatorial explosion, but we avoid indiscriminate insertion in Ψ of all pairs corresponding to a call to `subtype`.

Optimization of rule (comp $\leq$): besides all optimizations explained above, we have also implemented a more refined version of rule (comp $\leq$): before checking that $\pi \setminus \tau \leq \Xi \setminus \{\tau\}$ holds, we verify whether $\pi \setminus \tau$ is already empty (thus, $\pi \setminus \tau \leq \emptyset$ holds), to avoid useless applications of rule (comp $\leq$).

7 Conclusion

In this paper we have tackled the problem of defining a practical top-down algorithm for deciding semantic subtyping for coinductively interpreted types in the presence of record and union types.

We have defined a set of coinductive subtyping rules, and proved that such a set is sound and complete w.r.t. semantic subtyping; from such rules an algorithm has been derived and implemented by a prototype written in Prolog.

As a byproduct, we have proposed and used a new proof technique that can be fruitfully used for proving soundness results for coinductively defined judgments (or, dually, for proving completeness results for inductively defined judgments).

We have shown with an example in Python how coinductive types allow more precise type analysis in the presence of cyclic objects; furthermore, a complete procedure for deciding subtyping makes the analysis even more precise. This work can be directly applied to our previous work on abstract compilation for object-oriented languages [4,3,6,7] to perform static global type analysis; the types employed by abstract compilation are essentially the same studied here, with the difference that the previously defined subtyping rules were sound but not complete [5]. Actually, our prototype implementation supports the same types as defined in our first work on coinductive types [4].

There are several directions for further research on this topic. To simplify the technical details, in this paper we have considered non updatable records (that is, record subtyping is covariant in the types of the fields), but for effectively using our result in object-oriented languages, the subtyping algorithm has to be extended to updatable records (that is, record subtyping is invariant in the types of the updatable fields).

Besides updatable records there are other interesting extensions to the type system and to the subtyping algorithm to obtain more precise type analysis; in particular, the addition of polymorphic types would require a non trivial extension of the subtyping algorithm to handle set of subtyping constraints with type variables.

References

1. Amadio, R., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems 15(4), 575–631 (1993)
2. Ancona, D., Corradi, A.: Sound and complete subtyping between coinductive types for object-oriented languages. Technical report, DIBRIS - Università di Genova, Italy (2014), `ftp://ftp.disi.unige.it/person/AnconaD/CompleteCoinductiveSubtyping.pdf`
3. Ancona, D., Corradi, A., Lagorio, G., Damiani, F.: Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 31–45. Springer, Heidelberg (2011)
4. Ancona, D., Lagorio, G.: Coinductive type systems for object-oriented languages. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 2–26. Springer, Heidelberg (2009)
5. Ancona, D., Lagorio, G.: Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In: Montanari, A., Napoli, M., Parente, M. (eds.) Proceedings of GandALF 2010. Electronic Proceedings in Theoretical Computer Science, vol. 25, pp. 214–223 (2010)

6. Ancona, D., Lagorio, G.: Idealized coinductive type systems for imperative object-oriented programs. RAIRO - Theor. Inf. and Applic. 45(1), 3–33 (2011)
7. Ancona, D., Lagorio, G.: Static single information form for abstract compilation. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) TCS 2012. LNCS, vol. 7604, pp. 10–27. Springer, Heidelberg (2012)
8. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Intersection and union types: Syntax and semantics. Information and Computation 119(2), 202–230 (1995)
9. Bonsangue, M., Rot, J., Ancona, D., de Boer, F., Rutten, J.: A coalgebraic foundation for coinductive union types. In: 41st International Colloquium on Automata, Languages and Programming, ICALP 2014 (to appear, 2014)
10. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundam. Inform. 33(4), 309–338 (1998)
11. Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science 25, 95–169 (1983)
12. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM 55(4) (2008)
13. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Trans. Internet Techn. 3(2), 117–148 (2003)
14. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. 27(1), 46–90 (2005)
15. Igarashi, A., Nagira, H.: Union types for object-oriented programming. Journ. of Object Technology 6(2), 47–68 (2007)
16. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207, 284–304 (2009)

A Artifact Description

Authors of the Artifact. Davide Ancona and Andrea Corradi.

Summary. We have developed a prototype implementation of the presented algorithm in SWI Prolog; besides allowing rapid prototyping and conciseness, Prolog has the advantage of offering native support for regular terms and unification, which is very useful for defining coinductively defined functions which returns regular terms (consider for instance the problem of implementing type normalization as defined in Figure 5.

Although the prototype has been developed as a proof of concept, and more optimizations and an implementation in a more efficient programming language should be considered, the numerous tests show that the algorithm is usable in practice.

As an example of the performed tests, let us consider the following two types τ_L, τ_{EL}, and τ_{OL} defined by the following equations:

$$\tau_L = \langle el{:}int, nx{:}\tau_L\rangle \vee null$$
$$\tau_{EL} = \langle el{:}int, nx{:}\langle el{:}int, nx{:}\tau_{EL}\rangle\rangle \vee null$$
$$\tau_{OL} = \langle el{:}int, nx{:}\langle el{:}int, nx{:}\tau_{OL}\rangle\rangle \vee \langle el{:}int, nx{:}null\rangle$$

Type τ_L corresponds to all integer lists, whereas τ_{EL} and τ_{OL} represent all integer lists whose length (when finite) is even and odd, respectively. As expected, the tests $\tau_{EL} \vee \tau_{OL} \leq \tau_L$ and $\tau_L \leq \tau_{EL} \vee \tau_{OL}$ succeed, whereas $\tau_L \leq \tau_{EL}$ and $\tau_L \leq \tau_{OL}$ fail.

Content. The artifact package includes:

- `README.txt`: explanation of how the artifact works and how to use it.
- `results.pdf`: experimental results.
- `src/contractive.pl`: contractivity check.
- `src/normalization.pl`: type normalization as defined in the paper.
- `src/plunit.pl`: unit testing framework.
- `src/subtype.pl`: implementation of the main predicate subtype/2.
- `src/tests.pl`: tests for the subtype predicate and code to run the benchmarks.

Getting the Artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. The latest version of our code is available on `ftp://ftp.disi.unige.it/person/AnconaD/ECOOP14artifact.zip`.

Tested Platforms. The artifact is known to work on any platform running SWI Prolog (`http://swi-prolog.org/`) version 6.6.

License. GPL-2.0
(`https://www.gnu.org/licenses/old-licenses/gpl-2.0.txt`)

MD5 Sum of the Artifact. fac97ebe56df60b35de45fe7a32ebd6f

Size of the Artifact. 162 KB