Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering, © JOT 2010
Online at http://www.jot.fm.

On sound and complete axiomatization of coinductive subtyping for object-oriented languages

Davide Ancona[a] Giovanni Lagorio[a]

a. DISI, University of Genova, Italy

Acknowledgments This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

Abstract Coinductive abstract compilation is a novel technique, which has been recently introduced for defining precise type systems for object-oriented languages. In this approach, type inference consists in translating the program to be analyzed into a Horn formula f, and in resolving a certain goal w.r.t. the coinductive (that is, the greatest) Herbrand model of f.

Type systems defined in this way are idealized, since types and, consequently, goal derivations, are not finitely representable. Hence, sound implementable approximations have to rely on the notions of *regular* types and derivations, and of *subtyping* and *subsumption* between types and atoms, respectively.

In this paper we address the problem of defining a sound and complete axiomatization of a subtyping relation between coinductive object and union types, defined as set inclusion between type interpretations. Besides being an important theoretical result, completeness is useful for reasoning about possible implementations of the subtyping relation, when restricted to regular types.

1 Introduction

Coinductive abstract compilation [6, 2] is a novel technique, which has been recently introduced, for defining precise type systems for object-oriented languages. In this approach, type inference consists in translating the program to be analyzed into a Horn formula f, and in resolving a certain goal w.r.t. the coinductive (that is, the greatest) Herbrand model of f. Furthermore, union types, besides coinduction and object types, play an important role for guaranteeing the high expressive power of the type language.

This novel approach is quite general and modular, since it can be used for defining quite different type systems for the same language, by simply devising different kinds of translations into Horn formulas, without changing the core inference engine. In contrast with this, most of the solutions to the problem of type analysis of object-oriented programs which can be found in literature [16, 15, 1, 21, 20, 13] have their own inference engine, and cannot be easily compared and specified.

Furthermore, defining type systems in terms of compilation into a logical language eases reuse of all those static analysis techniques proposed for compiler optimization which can be fruitfully adopted for enhancing precision of type analysis. For instance, we have shown that a more precise type analysis can be obtained when abstract compilation is performed on programs in Static Single Assignment intermediate form [5].

Type systems defined by coinductive abstract compilation are idealized, since types and, consequently, goal derivations, are not finitely representable. Hence, sound implementable approximations have to rely on the notions of *regular* types and derivations, and of *subtyping* and *subsumption* between types and atoms, respectively. For this reason, studying subtyping between coinductive union and object types is very important to investigate on possible implementations of the inference engine based on coinductive constraint logic programming [7, 18, 17]. Such approach is similar to the work by Sulzmann and Stuckey [19] who have shown that the generalized Hindley/Milner type inference problem HM(X) can be mapped to inductive CLP(X); the main difference is that we consider coinductive (rather than inductive) Herbrand models and types, and that we are focused on type inference for Java-like (rather than functional) languages.

However, devising a sound definition of subtyping between coinductive types is far from being intuitive, since a suitable notion of *contractive* [9, 10] derivation has to be introduced to avoid unsound derivations. Furthermore, without an appropriate interpretation of types and subtyping, one cannot easily reason on the completeness of subtyping. In a previous work [3] we have proposed to interpret coinductive types as sets of values defined by a quite intuitive coinductive relation of membership, and proved that the proposed axiomatization of subtyping is sound w.r.t. subset inclusion between type interpretations. This result allowed us to relax the condition of contractiveness as previously defined [5], and to add a new typing rule for dealing with a case previously uncovered.

In a more recent paper [4] we discovered that the previously proposed axiomatization [3] was not complete and proposed to add a new rule (split) (which is actually a generalization of previous rules). Despite we have shown that such an axiomatization still fails to be complete in the idealized system where types and proofs can be non regular, we have conjectured that it is complete in the more interesting case (for obvious practical reasons) when one considers only regular types and proof trees.

This paper is an extended version of the above mentioned work [4], where the main new contributions are the following:

- We show by means of a counter-example that the previously defined axiomatization [4] is not complete, even when we restrict ourselves to regular types and proofs, thus disproving the corresponding conjecture.
- Correspondingly, we add a new rule (merge) to overcome the problem found with the counter-example; we prove that the new axiomatization is still sound

and is complete when one considers only regular types and proofs. Proof of completeness relies on a conjecture which is weaker then the previously proposed one.

- All the proof details have been added.

2 An introductory example

Let us consider the standard encoding of natural numbers with objects, written in Java-like code where, however, all type annotations have been omitted.

```
class Zero { twice() { return this; } }
class Succ {
   pred;
   Succ(n) { this.pred=n; }
   twice() {
     return new Succ(new Succ(pred.twice()));
   } }
```

For simplicity, we just consider method `twice`; class `Succ` represents all natural numbers greater than zero, that is, all numbers which are successors of a given natural number, stored in the field `pred`.

In the abstract compilation approach a program, as the one shown above, is translated into a Horn formula where predicates encode the constructs of the language. For instance, the predicate *invoke* corresponds to method invocation, and has four arguments: the target object, the method name, the argument list, and the returned result. Terms represent either types (that is, set of values) or names (of classes, methods and fields). In this example we use type *int*, union types $t_1 \vee t_2$, object types $obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])$, where c is the class of the object and $f_1, \ldots, f_n$ and $t_1, \ldots, t_n$ its fields with their types. In the idealized abstract compilation framework, terms can be also infinite and non regular[1]; a regular term is a term which can be infinite, but can only contain a finite number of subterms or, equivalently, can be represented as the solution of a unification problem, that is, a finite set of syntactic equations of the form $X_i = t_i$, where all variables X_i are distinct and terms t_i may only contain variables X_i [11, 18, 17]. For instance, the term t s.t. $t = int \vee t$ is regular since it has only two subterms, namely, *int* and itself.

Let us see some examples of regular types, that is, regular terms representing set of values.

$$\begin{array}{lcl} zer & = & obj(zero, [\,]) \\ nat & = & zer \vee obj(succ, [pred{:}nat]) \\ evn & = & zer \vee obj(succ, [pred{:}obj(succ, [pred{:}evn])]) \end{array}$$

Type *zer* corresponds to all objects representing zero, while *nat* corresponds to all objects representing natural numbers and, similarly, *evn* to all objects representing even natural numbers. An example of non regular types is given by the infinite sequence $t_0 \vee (t_1 \vee (\ldots \vee t_n \ldots))$, where the term t_i represents the natural number 2^i.

Each method declaration is compiled into a single clause, defining a different case for the predicate *has_meth*, that takes four arguments: the class where the method is declared, its name, the types of its arguments, including the special argument *this* corresponding to the target object, and the type of the returned value. Predicate

[1]We refer to the author's previous work [6, 2, 5] for more details.

has_meth defines the usual method look-up: $\mathit{has\_meth}(c, m, [\mathit{this}, t_1, \ldots, t_n], t)$ succeeds if look-up of m from class c succeeds and returns a method that, when invoked on target object and arguments $\mathit{this}, t_1, \ldots, t_n$, returns values of type t.

For instance, the method declarations of the two classes defined above are compiled as follows:

```
has_meth(zero,twice,[This],This).

has_meth(succ,twice,[This],R4) ←
    field_acc(This,pred,R1),
    invoke(R1,twice,[],R2),
    new(succ,[R2],R3),
    new(succ,[R3],R4).
```

Predicates *field_acc*, *new* and *invoke* correspond to field access, constructor invocation and method invocation, respectively. Similarly to what happens for methods, each constructor declaration is also compiled into a clause. For instance, the following clause is generated from the constructor of class `Succ`:

```
new(succ,[N],obj(succ,[pred:N|R])) ←
           extends(succ,P),new(P,[],obj(P,R)).
```

Other generated clauses are common to all programs and depend on the semantics of the language or on the meaning of types.

```
invoke(T1∨T2,M,A,R1∨R2) ←
         invoke(T1,M,A,R1), invoke(T2,M,A,R2).
invoke(obj(C,R),M,A,Res) ←
      has_meth(C,M,[obj(C,R)|A],Res).
```

The first clause specifies the behavior of invoke with union types. The invocation must be correct for both target types T_1 and T_2 and the returned type is the union of the returned types R_1 and R_2. When the target is an object type $\mathit{obj}(C, R)$, then invocation of M with arguments A is correct if look-up of M with first argument $\mathit{obj}(C, R)$, corresponding to *this*, and rest of arguments A succeeds when starting from class C.

We show now that the goal $\mathit{invoke}(\mathit{nat}, \mathit{twice}, [\,], R)$ is derivable for $R = \mathit{evn}$. This means that, not only we can prove that by doubling any natural number we get an even number, but we can also infer the thesis (that is, the result is an even number), since the query corresponds to just asking which number is returned when doubling any natural number.

We recall that the coinductive Herbrand model is obtained by considering also infinite proof trees (a.k.a. derivations) [18]. Then, since $\mathit{nat} = \mathit{zer} \vee \mathit{obj}(\mathit{succ}, [\mathit{pred}{:}\mathit{nat}])$, by clause 1 for *invoke* the following atoms must be derivable:

$$\begin{array}{l} \mathit{invoke}(\mathit{zer}, \mathit{twice}, [\,], \mathit{zer}) \\ \mathit{invoke}(\mathit{succ}(\mathit{nat}), \mathit{twice}, [\,], \mathit{succ}^2(\mathit{evn})) \end{array}$$

where $\mathit{succ}(t)$ is an abbreviation for $\mathit{obj}(\mathit{succ}, [\mathit{pred}{:}t])$, and $\mathit{succ}^2(t)$ is an abbreviation for $\mathit{succ}(\mathit{succ}(t))$.

The first atom can be derived by applying clause 2 for *invoke*, and then the clause for *has_meth* generated from class `Zero`. For the second atom we apply clause 2 for

$$(\text{int})\frac{}{i \in \mathit{int}} \qquad (\vee\text{L})\frac{v \in t_1}{v \in t_1 \vee t_2} \qquad (\vee\text{R})\frac{v \in t_2}{v \in t_1 \vee t_2}$$

$$(\text{obj})\frac{v_1 \in t_1, \ldots, v_n \in t_n}{\mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots]) \in \mathit{obj}(c, [f_1{:}t_1, \ldots, f_n{:}t_n])}$$

Figure 1 – Rules defining membership

invoke, and then the clause for *has_meth* generated from class `Succ` and get

$$\begin{array}{l}\mathit{field\_acc}(\mathit{succ}(\mathit{nat}), \mathit{pred}, \mathit{nat}),\\ \mathit{invoke}(\mathit{nat}, \mathit{twice}, [\ ], \mathit{evn}), \quad \mathit{new}(\mathit{succ}, [\mathit{evn}], \mathit{succ}(\mathit{evn})),\\ \mathit{new}(\mathit{succ}, [\mathit{succ}(\mathit{evn})], \mathit{succ}^2(\mathit{evn})).\end{array}$$

All atoms are derivable, in particular $\mathit{invoke}(\mathit{nat}, \mathit{twice}, [\ ], \mathit{evn})$ corresponds to our initial goal, hence the derivation we obtain is infinite even though regular. Similarly, it is possible to derive $\mathit{invoke}(\mathit{evn}, \mathit{twice}, [\ ], \mathit{four})$, where $\mathit{four} = \mathit{zer} \vee \mathit{succ}^4(\mathit{four})$, that is, by doubling an even number we get a multiple of four.

Surprisingly and regrettably, $\mathit{invoke}(\mathit{evn}, \mathit{twice}, [\ ], \mathit{evn})$ is not derivable, since $\mathit{new}(\mathit{succ}, [t_1], t_2)$ is never derivable when t_2 is a union type. To overcome these problems, a subtyping relation has to be introduced together with a notion of subsumption between atoms. The definition of the subtyping relation is postponed to the next section, however the intuition suggests that $\mathit{four} \leq \mathit{evn}$ should hold, and since subtyping is covariant w.r.t. the type returned by a method invocation, we expect that $\mathit{invoke}(\mathit{evn}, \mathit{twice}, [\ ], \mathit{evn})$ can be subsumed from $\mathit{invoke}(\mathit{evn}, \mathit{twice}, [\ ], \mathit{four})$.

By introducing subtyping and subsumption it is possible to find regular derivations for goals that would otherwise have infinite but not regular derivations (an example can be found in another paper by the same authors [3]); in other words, subtyping and subsumption are essential for implementing reasonable approximations of idealized coinductive type systems defined by abstract compilation. To do that, coSLD resolution [18] is generalized by taking into account subtyping constraints between terms, besides the usual unification constraints.

3 Definition and Axiomatization of Subtyping

In this section we define object and union coinductive types, and provide an intuitive interpretation of types as sets of values, and define subtyping as set inclusion between type interpretations. Then we provide an axiomatization of subtyping based on coinductive subtyping rules.

3.1 Interpretation of Types and Definition of Subtyping

The types we consider are all infinite terms coinductively defined as follows:

$$t \quad ::= \quad \mathit{int} \mid \mathit{obj}(c, [f_1{:}t_1, \ldots, f_n{:}t_n]) \mid t_1 \vee t_2$$

An object type $\mathit{obj}(c, [f_1{:}t_1, \ldots, f_n{:}t_n])$ specifies the class c to which the object belongs, together with the set of available fields with their corresponding types. The class name is needed for typing method invocations. We assume that fields in an object type are

$$(\vee\text{R1})\frac{t \le t_1}{t \le t_1 \vee t_2} \qquad (\vee\text{R2})\frac{t \le t_2}{t \le t_1 \vee t_2} \qquad (\text{split})\frac{\mathcal{C}[t_1] \le t \quad \mathcal{C}[t_2] \le t}{\mathcal{C}[t_1 \vee t_2] \le t}$$

$$(\text{int})\frac{}{\mathit{int} \le \mathit{int}} \qquad (\text{obj})\frac{t_1 \le t'_1, \dots, t_n \le t'_n}{\mathit{obj}(c, [f_1{:}t_1, \dots, f_n{:}t_n, \dots]) \le \mathit{obj}(c, [f_1{:}t'_1, \dots, f_n{:}t'_n])}$$

$$\mathcal{C}[\,] \quad ::= \quad \square \mid \mathit{obj}(c, [f{:}\mathcal{C}[\,], f_1{:}t_1, \dots, f_n{:}t_n])$$

Figure 2 – Coinductive subtyping rules of system $\mathcal{S}_n$

finite, distinct and that their order is immaterial. Union types $t_1 \vee t_2$ have the standard meaning [8, 14].

We interpret types in a quite intuitive way, that is, as sets of values. Values are all infinite terms coinductively defined by the following syntactic rules (where $i \in \mathbb{Z}$).

$$v \quad ::= \quad i \mid \mathit{obj}(c, [f_1 \mapsto v_1, \dots, f_n \mapsto v_n])$$

As it happens for object types, fields in object values are finite and distinct, and their order is immaterial. Regular values correspond to finite, but cyclic, objects.

Membership of values to (the interpretation of) types is coinductively defined by the rules of Figure 1.

All rules are intuitive. Note that an object value is allowed to belong to an object type having less fields; this is expressed by the ellipsis at the end of the values in the membership rule (obj).

A proof tree (or, simply, a proof) is a tree where each node is a pair consisting of a judgment of the shape $v \in t$, and of a rule label[2], and where each node, together with its children, corresponds to a valid instantiation of a rule. For instance, the following tree is a proof for $\mathit{obj}(c, [f \mapsto 1]) \in \mathit{int} \vee \mathit{obj}(c, [f{:}\mathit{int}])$.

$$(\vee\text{R})\frac{(\text{obj})\dfrac{(\text{int})\dfrac{}{1 \in \mathit{int}}}{\mathit{obj}(c, [f \mapsto 1]) \in \mathit{obj}(c, f{:}\mathit{int})}}{\mathit{obj}(c, [f \mapsto 1]) \in \mathit{int} \vee \mathit{obj}(c, f{:}\mathit{int})}$$

Since values and types can be infinite, all rules must be interpreted coinductively, therefore proofs are allowed to be infinite. However, not all infinite proofs can be considered valid, but only those *contractive* (see the definition below). To see why we need such a restriction, consider the regular type t s.t. $t = t \vee \mathit{int}$, and the following infinite proof containing just applications of rules ($\vee$L):

$$(\vee\text{L})\frac{(\vee\text{L})\dfrac{\vdots}{\mathit{obj}(c, [\,]) \in t}}{\mathit{obj}(c, [\,]) \in t}$$

We reject proofs built applying only rules ($\vee$L) and ($\vee$R), since they generate false judgments, as $\mathit{obj}(c, [\,]) \in t$ derived above: t corresponds to an infinite union of *int*, and therefore its interpretation cannot contain object values. Intuitively, the problem is due to the fact that rules ($\vee$L) and ($\vee$R) allow membership checking only on one part of the type, therefore we may end up with an infinite proof which in fact does not check anything. Following the terminology used by Brandt and Henglein [9, 10], we say that rules (int) and (obj) are contractive, whereas ($\vee$L) and ($\vee$R) are not.

[2] This labeling is necessary for the notion of contractive proof, see below.

Def. 3.1 *A proof for $v \in t$ is contractive iff it contains no sub-proofs built only with membership rules ($\vee R$), and ($\vee L$). The membership relation $v \in t$ holds iff there is a contractive proof for it.*

The interpretation of type t is denoted by $[\![t]\!]$ and defined by $\{v \mid v \in t \text{ holds}\}$.

In the following we use the term *proof* for contractive proofs, unless explicitly specified.

Def. 3.2 *Type t_1 is a subtype of t_2 iff $[\![t_1]\!] \subseteq [\![t_2]\!]$.*

Example 1 If $\bot$ is the regular type s.t. $\bot = \bot \vee \bot$, then $[\![\bot]\!] = \emptyset$. Indeed, the only applicable rules for $v \in \bot$ are ($\vee$L) and ($\vee$R), hence only non contractive proofs can be built, therefore no judgments of the shape $v \in \bot$ can be proved, and $[\![\bot]\!] = \emptyset$. A type t s.t. $[\![t]\!] = \emptyset$ is called an empty type.

Example 2 If t is s.t. $t = t \vee \mathit{int}$, then $[\![t]\!] = [\![\mathit{int}]\!] = \mathbb{Z}$. Indeed, all the contractive proofs for $v \in t$ are obtained by uselessly applying n times ($n \geq 0$) rule ($\vee$L), before the decisive applications of rule ($\vee$R) and (int):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{i \in \mathit{int}}\,(\text{int})}{i \in \mathit{int} \vee t}\,(\vee\text{R})}{\vdots}\,(\vee\text{L})}{i \in \mathit{int} \vee t}\,(\vee\text{L})$$

Other interesting examples of types and their interpretations can be found in another paper [3].

Example 3 Let us consider the infinite (but not regular) type t_1 defined by the following infinite set of equations (where t_1 corresponds to X_0):

$$\begin{array}{rcl}
X_0 & = & Y_0 \vee X_1 \\
 & \cdots & \\
X_n & = & Y_n \vee X_{n+1} \\
 & \cdots & \\
Y_0 & = & \mathit{obj}(\mathit{zero}, [\,]) \\
Y_1 & = & \mathit{obj}(\mathit{succ}, [\mathit{pred}{:}\,Y_0]) \\
 & \cdots & \\
Y_{n+1} & = & \mathit{obj}(\mathit{succ}, [\mathit{pred}{:}\,Y_n]) \\
 & \cdots &
\end{array}$$

Let t_2 be the term s.t. $t_2 = \mathit{obj}(\mathit{zero}, [\,]) \vee \mathit{obj}(\mathit{succ}, [\mathit{pred}{:}t_2])$. Then $[\![t_1]\!] \subsetneq [\![t_2]\!]$; indeed, it is easy to show that $[\![t_1]\!]$ is the set of all objects representing natural numbers, and that such values belong to $[\![t_2]\!]$ as well (all proofs are finite, hence trivially contractive), whereas the value v_∞ s.t. $v_\infty = \mathit{obj}(\mathit{succ}, [\mathit{pred} \mapsto v_\infty])$ belongs to t_2, but not to t_1. Indeed, the following contractive and regular proof can be built by alternatively applying rules ($\vee$R) and (obj) infinite times.

$$\dfrac{\dfrac{\dfrac{\vdots}{v_\infty \in t_2}}{v_\infty \in \mathit{obj}(\mathit{succ}, [\mathit{pred}{:}t_2])}}{v_\infty \in t_2}$$

Finally, it is not difficult to prove that the only proof for $v_\infty \in t_1$ is not contractive, since it can be obtained by infinitely applying rule ($\vee$R); therefore $v_\infty \notin t_1$.

3.2 Subtyping rules

A possible axiomatization of the subtyping relation as defined in Section 3.1 is given by the system $\mathcal{S}_n$ of coinductive rules in Figure 2. Such rules are conceived for a purely functional setting [2]; an extension for dealing with imperative features can be found in another paper [5] by the same authors.

All the previously proposed axiomatizations of the subtyping relation [6, 2, 3], are not complete, although the most recent proposal $\mathcal{S}_1$ [3] has been proved sound.

In comparison to system $\mathcal{S}_n$ defined in Figure 2, system $\mathcal{S}_1$ defines rules ($\vee$L) and (distr) instead of rule (split), which is, in fact, a generalization of the former rules.

$$(\vee\text{L})\,\frac{t_1 \le t \quad t_2 \le t}{t_1 \vee t_2 \le t} \qquad (\text{distr})\,\frac{\begin{array}{c} \mathit{obj}(c,[f{:}u_1,f_1{:}t_1,\ldots,f_n{:}t_n]) \le t \\ \mathit{obj}(c,[f{:}u_2,f_1{:}t_1,\ldots,f_n{:}t_n]) \le t \end{array}}{\mathit{obj}(c,[f{:}u_1 \vee u_2,f_1{:}t_1,\ldots,f_n{:}t_n]) \le t}$$

Such a generalization has been devised to overcome the problem that rules ($\vee$L) and (distr) do not ensure completeness of subtyping even when the system is restricted to regular types and proofs; e.g., $\mathit{obj}(c,[f{:}\mathit{obj}(c',[g{:}t_1 \vee t_2])]) \le \mathit{obj}(c,[f{:}\mathit{obj}(c',[g{:}t_1])]) \vee \mathit{obj}(c,[f{:}\mathit{obj}(c',[g{:}t_2])])$ cannot be proved without the more general rule (split).

The one hole context $\mathcal{C}$ (inductively defined on the depth of the hole; see the same figure) is used for applying the rule when the type on the left-hand side of the relation contains a union type; a context is either the empty one (consisting of just the hole), or is an object type with a context inside (since the order of field is immaterial, the hole can be contained in any field type). Note that there are no contexts which are union types, since rule (split) is applied only to the outer union types; for instance, if the judgement has shape $t_1 \vee t_2 \le t$, then the rule can be instantiated only with the empty context.

Rule ($\vee$L) is simply obtained by instantiating (split) with the empty context. Rules ($\vee$R1), ($\vee$R2) and ($\vee$L) specify subtyping in the presence of union types: the union type constructor is the join operator w.r.t. subtyping. Note also the strong analogy with the left and right logical rules of the classical Gentzen sequent calculus for the disjunction, when the subtyping relation is replaced with the provability relation.

Rule (obj) corresponds to standard width and depth subtyping between object types: the type on the left-hand side may have more fields (represented by the ellipsis at the end), while subtyping is covariant w.r.t. the fields belonging to both types. Note that depth subtyping is allowed since we are considering a purely functional setting [5]. Finally, subtyping between object types is allowed only when they refer to the same class name.

Rule (distr) is obtained by instantiating (split) with contexts of shape $\mathit{obj}(c,[f{:}\square,f_1{:}t_1,\ldots,f_n{:}t_n])$, where $\square$ is the empty context. The rule states that object types distribute over union types, in the same way Cartesian product distributes over union.

For instance, if $u_1 = \mathit{obj}(c,[f{:}t_1]) \vee \mathit{obj}(c,[f{:}t_2])$ and $u_2 = \mathit{obj}(c,[f{:}t_1 \vee t_2])$, then $[\![u_1]\!] = [\![u_2]\!]$, hence if completeness holds, one should be able to derive $u_1 \cong u_2$, that is, $u_1 \le u_2$ and $u_2 \le u_1$. The relation $u_1 \le u_2$ can be derived by applying rules ($\vee$L), (obj), ($\vee$R1) and ($\vee$R2), and by reflexivity[3]. Rule (distr) is necessary for deriving the opposite direction $u_2 \le u_1$ of the relation, since by applying rules ($\vee$R1), ($\vee$R2) and (obj) we end up with $t_1 \vee t_2 \le t_1$ or $t_1 \vee t_2 \le t_2$ which in general do not hold.

[3]It is not difficult to prove that reflexivity of subtyping holds.

$$(\text{split-0})\dfrac{(\text{split-0})\dfrac{\vdots}{t \le \mathit{int}} \qquad (\text{split-1})\dfrac{(\text{split-1})\dfrac{\vdots}{\mathit{obj}(c,[f{:}t]) \le \mathit{int}} \qquad (\text{split-2})\dfrac{\vdots}{\mathit{obj}(c,[f{:}\mathit{obj}(c,[f{:}t])]) \le \mathit{int}}}{\mathit{obj}(c,[f{:}t]) \le \mathit{int}}}{t \le \mathit{int}}$$

Figure 3 – A non contractive derivation for $t \le \mathit{int}$, with $t = t \vee \mathit{obj}(c, [f{:}t])$.

For reasons similar to those explained in the previous section, proofs have to be contractive, otherwise unsound judgments can be derived. For instance, if $t = t \vee \mathit{int}$, then we could derive $\mathit{obj}(c, [\,]) \le t$ by only applying rule ($\vee$R1). However, giving a definition of contractive proof for subtyping which does not break soundness without compromising completeness (at least when the system is restricted to regular types and proofs) is not trivial. In this case Def. 3.1 alone does not ensure soundness. For instance, if t is the term s.t. $t = t \vee \mathit{obj}(c, [f{:}t])$, then we can build a proof for the clearly unsound judgment $t \le \mathit{int}$ by using only rule (split) as shown in Figure 3. We have decorated the label of each rule application with a natural number corresponding to the depth of the hole of the context used for applying rule (split); the problem with the shown proof is that the hole depth of contexts used for applying rule (split) is unbounded, hence types are never "consumed" as it happens in rules (int) and (obj).

Def. 3.3 *The depth of a context $\mathcal{C}[\,]$ is inductively defined as follows:*

$$\begin{aligned}&\mathit{depth}(\Box) = 0\\ &\mathit{depth}(\mathit{obj}(c, [f{:}\mathcal{C}[\,], f_1{:}t_1, \ldots, f_n{:}t_n])) = \mathit{depth}(\mathcal{C}[\,]) + 1\end{aligned}$$

An application of rule (split) has depth n iff the rule is applied with a context of depth n.

Def. 3.4 *A proof for $t_1 \le t_2$ is contractive iff it contains no sub-proofs built only with subtyping rules ($\vee$R), and ($\vee$L), and the depth of the applications of rule (split) is bounded.*

The subtyping relation $t_1 \le t_2$ holds iff there is a contractive proof for it.

Given the definitions above, we have that applications of rules ($\vee$L) and (distr) are in fact applications of rule (split) having depth 0 and 1, respectively, hence any subtyping proof in $\mathcal{S}_1$ corresponds to a subtyping proof in $\mathcal{S}_n$ where the depth of the applications of rule (split) is bounded by 1. This is the meaning of the subscript 1 in $\mathcal{S}_1$, whereas $\mathcal{S}_n$ means that applications of rule (split) can be bounded by any natural number n. For this reason, the last condition on contractive definitions in Def. 3.4 is always verified for proofs in system $\mathcal{S}_1$.

4 Soundness

Soundness holds in the most general case, when types and derivations are allowed to be non regular. The proof is a non trivial adaptation of that given by the same authors [3] for the less general system with rules ($\vee$L) and (distr) instead of (split).

Def. 4.1 *Let ∇ be a proof of $v \in \mathcal{C}[t]$. Then, $rank(\nabla, \mathcal{C}[\ ])$ is inductively defined over the depth of $\mathcal{C}[\ ]$ as follows:*

- *If $\mathcal{C}[\ ] = \Box$, then $rank(\nabla, \mathcal{C}[\ ])$ is the number of consecutive applications of rules ($\vee$L) or ($\vee$R) starting from the root of ∇. Such a number is always defined since ∇ is contractive (Def. 3.1).*
- *If $\mathcal{C}[\ ] = obj(c, [f{:}\mathcal{C}'[\ ], f_1{:}t_1, \ldots, f_n{:}t_n])$, then ∇ has shape $\frac{\nabla_f, \nabla_{f_1}, \ldots, \nabla_{f_n}}{v \in \mathcal{C}[t]}(obj)$, where ∇_f is the proof of $v_f \in \mathcal{C}'[t]$ for an opportune v_f, and $rank(\nabla, \mathcal{C}[\ ]) = rank(\nabla_f, \mathcal{C}'[\ ])$.*

Lemma 4.1 *$v \in \mathcal{C}[u_1 \vee u_2]$ iff $v \in \mathcal{C}[u_1]$ or $v \in \mathcal{C}[u_2]$.*

Proof: By induction over the depth of $\mathcal{C}[\ ]$.

If the context has depth 0 (therefore $\mathcal{C}[\ ] = \Box$), then the only applicable rules are ($\vee$L) and ($\vee$R), therefore $v \in u_1 \vee u_2$ iff $v \in u_1$ or $v \in u_2$.

If the context has shape $obj(c, [f{:}\mathcal{C}'[\ ], f_1{:}t_1, \ldots, f_n{:}t_n])$, then the only applicable rule is (obj), therefore $v \in obj(c, [f{:}\mathcal{C}'[u_1 \vee u_2], f_1{:}t_1, \ldots, f_n{:}t_n])$ iff $v = obj(c, [f \mapsto v_f, f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots])$, and $v_f \in \mathcal{C}'[u_1 \vee u_2]$, $v_1 \in t_1, \ldots, v_n \in t_n$. By inductive hypothesis $v_f \in \mathcal{C}'[u_1 \vee u_2]$ iff $v_f \in \mathcal{C}'[u_1]$ or $v_f \in \mathcal{C}'[u_2]$, hence, since (obj) is the only applicable rule, we conclude that $v \in obj(c, [f{:}\mathcal{C}'[u_1 \vee u_2], f_1{:}t_1, \ldots, f_n{:}t_n])$ iff $v \in obj(c, [f{:}\mathcal{C}'[u_1], f_1{:}t_1, \ldots, f_n{:}t_n])$ or $v \in obj(c, [f{:}\mathcal{C}'[u_2], f_1{:}t_1, \ldots, f_n{:}t_n])$.

As a final remark, note that the proof ∇' of $v \in \mathcal{C}[u_1]$ or $v \in \mathcal{C}[u_2]$ can be obtained from the proof ∇ of $v \in \mathcal{C}[u_1 \vee u_2]$ by just removing an application of rule ($\vee$L) or ($\vee$R), respectively, corresponding to the node $u_1 \vee u_2$ which fills in the hole of the context $\mathcal{C}[\ ]$. That is, $rank(\nabla', \mathcal{C}[\ ]) = rank(\nabla, \mathcal{C}[\ ]) - 1$. $\Box$

Corollary 4.1 *The following fact*

$v \in t_1$, and $t_1 \leq t_2$ with a proof where the first applied rule is (split) with a context $\mathcal{C}[\ ]$ s.t. $t_1 = \mathcal{C}[u_1 \vee u_2]$

is equivalent to

($v \in \mathcal{C}[u_1]$ or $v \in \mathcal{C}[u_2]$) and $\mathcal{C}[u_1] \leq t_2, \mathcal{C}[u_2] \leq t_2$.

Proof: A direct consequence of Lemma 4.1, rule (split) and the hypotheses. $\Box$

Lemma 4.2 *If $v \in t_0$ and $t_0 \leq u$, then any proof ∇ of $t_0 \leq u$ cannot contain an infinite sequence of terms $t_0 \leq u, \ldots, t_n \leq u, \ldots$ starting from the root of ∇ and obtained by applying only rule (split), where $v \in t_i$ for all i.*

Proof: By contradiction, let us assume that there exists such a sequence where for all $i \geq 0$, t_{i+1} is obtained from t_i by applying rule (split) with a certain context $\mathcal{C}_i[\ ]$ s.t. $t_i = \mathcal{C}_i[t_a \vee t_b]$ (for suitable t_a and t_b), $t_{i+1} = \mathcal{C}_i[t]$ with $t = t_a$ or $t = t_b$, $\mathcal{C}_i[t] \leq u$, and $v \in t_i$ for all i.

By Lemma 4.1, for all i we can associate with t_i a proof ∇_i for $v \in t_i$, where $\nabla_0 = \nabla$, and for all $i \geq 0$, ∇_{i+1} is derived from ∇_i and $rank(\nabla_{i+1}, \mathcal{C}_i[\ ]) = rank(\nabla_i, \mathcal{C}_i[\ ]) - 1$, as noted in Lemma 4.1.

To obtain a contradiction, we associate with each type t_i in the sequence an element $|t_i|$ in a Noetherian poset and show that $|t_{i+1}| < |t_i|$ for all i. Since ∇ is contractive (Def.3.4), the applications of (split) in ∇ are bounded by a natural number k. Therefore we can associate with each t_i a tuple $(n_0, \ldots, n_{k+1})$ of length

$k+2$ defined as follows: for all $j \in 0,\ldots,k$, $n_j = \mid \mathcal{C}^{\vee}_{t_i,j} \mid$, where $\mathcal{C}^{\vee}_{t_i,j} = \{\mathcal{C}[\,] \mid \mathit{depth}(\mathcal{C}[\,]) = j, \exists u', u''\ \mathcal{C}[u' \vee u''] = t_i\}$. Hence, j is the number of splittable terms at depth j contained in t_i, whereas $n_{k+1} = \sum_{\mathcal{C}[\,] \in \mathcal{C}^{\leq}_{t_i,k}} (\mathit{rank}(\nabla_i, \mathcal{C}[\,]))$, where $\mathcal{C}^{\leq}_{t_i,j} = \{\mathcal{C}[\,] \mid \mathit{depth}(\mathcal{C}[\,]) \leq j, \exists t\ \mathcal{C}[t] = t_i\}$.

The Noetherian poset we consider is the set $\mathbb{N}^{k+2}$ equipped with lexicographical order:

$$(n_0,\ldots,n_{k+1}) < (n'_0,\ldots,n'_{k+1}) \Leftrightarrow \exists j \in 0,\ldots,k{+}1\, \forall m \in 0,\ldots,j{-}1\ n_m = n'_m \wedge n_j < n'_j$$

We prove now that $|t_{i+1}| = (n'_0,\ldots,n'_{k+1}) < (n_0,\ldots,n_{k+1}) = |t_i|$ for all i. There are two distinct cases.

Case 1 $t_{i+1} = \mathcal{C}_i[u']$, where u' is a union type. Hence, for all $j \in 0,\ldots,k$, $n'_j = n_j$, since $\mathcal{C}^{\vee}_{t_{i+1},j} = \mathcal{C}^{\vee}_{t_i,j}$. Furthermore, since ∇_{i+1} is obtained from ∇_i by just removing an application of rule ($\vee$L) or ($\vee$R) (see Lemma 4.1), then $\mathit{rank}(\nabla_{i+1}, \mathcal{C}_i[\,]) = \mathit{rank}(\nabla_i, \mathcal{C}_i[\,]) - 1$, while for all $\mathcal{C}[\,] \neq \mathcal{C}_i[\,]$ in $\mathcal{C}^{\leq}_{t_i,k} = \mathcal{C}^{\leq}_{t_{i+1},k}$, $\mathit{rank}(\nabla_{i+1}, \mathcal{C}[\,]) = \mathit{rank}(\nabla_i, \mathcal{C}[\,])$. Therefore $n'_{i+1} = n'_i - 1$, and $|t_{i+1}| < |t_i|$.

Case 2 $t_{i+1} = \mathcal{C}_i[u']$, where u' is not a union type. Let $d = \mathit{depth}(\mathcal{C}_i[\,])$, then for all $j \in 0,\ldots,d-1$, $n'_j = n_j$, whereas $n'_d = n_d - 1$, since the term in the hole of $\mathcal{C}_i[\,]$ is no longer splittable, therefore $|t_{i+1}| < |t_i|$.

We finally note that if the sequence $t_0 \leq u,\ldots,t_n \leq u,\ldots$ is infinite, then we obtain the contradiction that there exists a infinite decreasing chain $\ldots,|t_{n+1}| < |t_n|,\ldots,|t_1| < |t_0|$ in a Noetherian poset. □

Corollary 4.2 *If $v \in t_0$ and $t_0 \leq u$, then any proof ∇ of $t_0 \leq u$ cannot contain an infinite sequence of terms $t_0 \leq u_0,\ldots,t_n \leq u_n,\ldots$ starting from the root of ∇ and obtained by applying only rules ($\vee$R1), ($\vee$R2) and (split), where $v \in t_i$ for all i.*

Proof: Since the proof is contractive, it cannot contain an infinite subsequence built only with subtyping rules ($\vee$R) and ($\vee$L), hence the whole sequence must contain infinite (though not necessarily contiguous) applications of rule (split). This is not possible, otherwise we would have an infinite chain $|t_0| \leq |t_1| \leq \ldots \leq |t_i| \leq \ldots$ where $|t_i|$ are elements of the Noetherian poset as defined in Lemma 4.2. Lemma 4.2 can be reused since applications of rules ($\vee$R) and ($\vee$L) leave unchanged the terms on the LHS of the subtyping relation. □

Lemma 4.3 *If $v \in t_1$ and $t_1 \leq t_2$, then there exists a type u (not necessarily equal to t_1) s.t. $v \in u$ and $u \leq t_2$ with a proof whose first applied rule is not (split).*

Proof: If the first applied rule of the proof of $t_1 \leq t_2$ is not (split), then the Lemma trivially holds for $u = t_1$. Otherwise by corollary 4.1 we can follow the path up of one level to get the proofs of $\mathcal{C}[u'] \leq t_2$ and $v \in \mathcal{C}[u']$ for suitable u', and iterate this process until we end up with a proof whose first applied rule is not (split). The termination of this process is guaranteed by Lemma 4.2, since there cannot exist an infinite path where only rule (split) is applied in the proof of $t_1 \leq t_2$. Again, as noted in Lemma 4.1 and 4.2, from any proof of $v \in t_1$ we can deterministically build a proof of $v \in u$. Furthermore, the proof of $u \leq t_2$ can be obtained by deterministically selecting a subproof from the proof of $t_1 \leq t_2$. □

To prove soundness we first define a total function $\mathcal{F}$ associating a pair of proof trees for $t_1 \leq t_2$ and $v \in t_1$, respectively, with a proof for $v \in t_2$.

The definition coinductive, and by cases on the first applied subtyping rule of such a proof.

Rule (int) $\mathcal{F}\left(\text{(int)}\frac{}{\mathit{int} \leq \mathit{int}}, \text{(int)}\frac{}{i \in \mathit{int}} \right) = \text{(int)}\frac{}{i \in \mathit{int}}$.

Rule ($\vee$R1) $\mathcal{F}\left((\vee\text{R1})\frac{\nabla_1}{t_1 \leq u_1 \vee u_2}, \nabla_2 \right) = (\vee\text{L})\frac{\mathcal{F}(\nabla_1, \nabla_2)}{v \in u_1 \vee u_2}$, where ∇_1 is a proof for $t_1 \leq u_1$, and ∇_2 is a proof for $v \in t_1$.

Rule ($\vee$R2) $\mathcal{F}\left((\vee\text{R2})\frac{\nabla_1}{t_1 \leq u_1 \vee u_2}, \nabla_2 \right) = (\vee\text{R})\frac{\mathcal{F}(\nabla_1, \nabla_2)}{v \in u_1 \vee u_2}$, where ∇_1 is a proof for $t_1 \leq u_2$, and ∇_2 is a proof for $v \in t_1$.

Rule (obj)

$$\mathcal{F}\left(\begin{array}{l} \text{(obj)}\dfrac{\nabla_1, \ldots, \nabla_n}{\mathit{obj}(c, [f_1{:}u_1, \ldots, f_n{:}u_n, \ldots]) \leq \mathit{obj}(c, [f_1{:}u'_1, \ldots, f_n{:}u'_n])}, \\ \text{(obj)}\dfrac{\nabla'_1, \ldots, \nabla'_n, \ldots}{\mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots]) \in \mathit{obj}(c, [f_1{:}u_1, \ldots, f_n{:}u_n, \ldots])} \end{array} \right) =$$

$$\text{(obj)}\frac{\mathcal{F}(\nabla_1, \nabla'_1), \ldots, \mathcal{F}(\nabla_n, \nabla'_n)}{\mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots]) \in \mathit{obj}(c, [f_1{:}u'_1, \ldots, f_n{:}u'_n])}$$

where $\nabla_1, \ldots, \nabla_n$ are proofs for $u_1 \leq u'_1, \ldots, u_n \leq u'_n$, respectively, whereas $\nabla'_1, \ldots, \nabla'_n$ are proofs for $v_1 \in u_1, \ldots, v_n \in u_n$, respectively.

The proof for $\mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots]) \in \mathit{obj}(c, [f_1{:}u_1, \ldots, f_n{:}u_n, \ldots])$ contains ellipses in the right hand side of the subproofs $\nabla'_1, \ldots, \nabla'_n$ and of the fields of both the value and the type, to mean that there are parts of the proof that may be omitted since the definition of $\mathcal{F}$ is independent from them.

Rule (split) $\mathcal{F}(\nabla_1, \nabla_2) = \mathcal{F}(\nabla'_1, \nabla'_2)$, where $\nabla_1 = \text{(split)}\frac{\nabla_a \ \nabla_b}{\mathcal{C}[u_a \vee u_b] \leq t_2}$, ∇_a, ∇_b are proofs of $\mathcal{C}[u_a] \leq t_2$ and $\mathcal{C}[u_b] \leq t_2$, respectively, ∇_2 is a proof of $v \in \mathcal{C}[u_a \vee u_b]$, and by Lemma 4.3 ∇'_1 and ∇'_2 are the proofs of $u \leq t_2$ and $v \in u$ (for a suitable u) deterministically obtained from ∇_1 and ∇_2, where the first applied rule of ∇'_1 is not (split). Therefore the definition of $\mathcal{F}$ for case (split) is delegated to one of the other cases.

Lemma 4.4 *The function $\mathcal{F}$ is well-defined and for any contractive proofs ∇_1 and ∇_2 of $t_1 \leq t_2$ and $v \in t_1$, respectively, $\mathcal{F}(\nabla_1, \nabla_2)$ returns a contractive proof of $v \in t_2$.*

Proof: We first need to prove that $\mathcal{F}$ is well-defined. Since $\mathcal{F}$ is defined coinductively, we have to prove that $\mathcal{F}$ is *deterministic*, that is, $\mathcal{F}(\nabla_1, \nabla_2) = \nabla$, $\mathcal{F}(\nabla_1, \nabla_2) = \nabla'$ implies $\nabla = \nabla'$. This can be proved by coinduction over the definition of $\nabla = \nabla'$, and by noting that all cases defining $\mathcal{F}$ are disjoint and that case (split) can always be reduced to one of the other cases, by virtue of Lemma 4.3.

We now prove that if ∇_1 and ∇_2 are proofs of $t_1 \leq t_2$ and $v \in t_1$, respectively, then $\mathcal{F}(\nabla_1, \nabla_2)$ is defined and returns a proof of $v \in t_2$ (for the moment we do not assume that proofs are contractive).

The proof that $\mathcal{F}(\nabla_1, \nabla_2)$ is defined is by coinduction over the definition of $\mathcal{F}$, and is based on the facts that the cases defining $\mathcal{F}$ are exhaustive, and that case (split) can always be reduced to one of the other cases, by virtue of Lemma 4.3.

The proof that $\mathcal{F}(\nabla_1, \nabla_2)$ returns a proof of $v \in t_2$ is by coinduction over the definition of proof, and is based on the definition of $\mathcal{F}$ and on the fact that case (split) can always be reduced to one of the other cases, by virtue of Lemma 4.3.

Finally, we prove that if ∇_1 and ∇_2 are contractive, then $\mathcal{F}(\nabla_1, \nabla_2)$ is contractive as well. By contradiction, if $\mathcal{F}(\nabla_1, \nabla_2)$ is not contractive, then it contains a subproof built only with membership rules ($\vee$R), and ($\vee$L). By construction, this can only be obtained if at a certain point only cases ($\vee$R1), ($\vee$R2), and (split) of the definition of $\mathcal{F}$ are applied. By Lemmas 4.3 and 4.1, if ∇_1 and ∇_2 are contractive and case (split) is applied (therefore $\mathcal{F}(\nabla_1, \nabla_2) = \mathcal{F}(\nabla'_1, \nabla'_2)$), then the proofs ∇'_1 and ∇'_2 are obtained from ∇_1 and ∇_2, respectively, by removing some rule applications, and, therefore, ∇'_1 and ∇'_2 are contractive as well. Finally, we obtain a contradiction by applying Corollary 4.2, since in case (split) ∇'_1 is always a subproof of ∇_1. □

Theorem 4.1 (Soundness) *For all t_1, t_2, if $t_1 \leq t_2$, then $[\![t_1]\!] \subseteq [\![t_2]\!]$.*

Proof: This theorem is a corollary of Lemma 4.4, since the claim of the theorem is equivalent to $t_1 \leq t_2$, $v \in t_1$ implies $v \in t_2$, and a proof of $v \in t_2$ is given by $\mathcal{F}(\nabla_1, \nabla_2)$, where ∇_1 and ∇_2 are proofs of $t_1 \leq t_2$ and $v \in t_1$, respectively. □

5 Towards completeness

Although we discovered pretty soon that the axiomatization defined in Figure 2 is not complete w.r.t. the idealized model where non regular types and proofs are allowed, the case where one restrict the model to regular types and proofs was not so clear, and initially we conjectured that the rules were complete. Unfortunately, completeness does not hold also for the restricted model.

We first provide a counter-example involving non regular types. Consider the terms defined by the following infinite set of equations:

$$\begin{array}{l}
t_1 = obj(c, [f{:}t_a \vee t_b, g{:}t_1]) \\
t_2 = u_a \vee u_b \\
u_a = obj(c, [f{:}t_a, g{:}u_a]) \\
u_b = u_0 \vee \ldots \vee u_n \vee \ldots \\
u_0 = obj(c, [f{:}t_b, g{:}t_1]) \\
\ldots \\
u_{n+1} = obj(c, [f{:}t_a, g{:}u_n]) \\
\ldots
\end{array}$$

We assume that t_a and t_b are two non empty incomparable types ($[\![t_a]\!] \neq \emptyset$, $[\![t_b]\!] \neq \emptyset$, $[\![t_a]\!] \cap [\![t_b]\!] = \emptyset$). Types t_1 and t_2 correspond to all infinite lists where each element has type t_a or t_b. Each u_n corresponds to the infinite lists where the $n+1$-th element has type t_b, all preceding n elements have type t_a, and all remaining elements have type $t_a \vee t_b$. Let v be a value s.t. $v \in [\![t_1]\!]$. If all fields f in v are associated with values in $[\![t_a]\!]$, then $v \in [\![u_a]\!]$, but $v \notin [\![u_n]\!]$ for all n (hence $v \notin [\![u_b]\!]$), since each value in $[\![u_n]\!]$ must contain a subvalue in $[\![t_b]\!]$ associated with the field f at position $n+1$. On the other hand, if v contain at least a subvalue in $[\![t_b]\!]$ associated with a field f, then $v \in [\![u_i]\!]$, where $i+1$ is the lowest position of a field f associated with a value in $[\![t_b]\!]$. Therefore $[\![u_b]\!] = [\![t_1]\!] \setminus [\![u_a]\!]$. Even though $[\![t_1]\!] = [\![t_2]\!]$, the only proofs for $t_1 \leq t_2$ are not contractive since the depth of the applications of rule (split) is necessarily unbounded. Indeed, by applying n (for $n > 0$ arbitrary) times rule (split) at increasing depths we obtain 2^n terms among which all are provably subtypes of u_b except for the one (let us call it t') which represents the lists where the first n elements have type t_a. Since $[\![t']\!] \not\subseteq [\![u_a]\!]$ and $[\![t']\!] \not\subseteq [\![u_b]\!]$ $t' \leq u_a$ and $t' \leq u_b$ cannot be derived by soundness. We conclude that all possible proofs are those which try to split indefinitely t_1, thus having an infinite path containing just applications of rule (split) with an unbounded depth.

The counter-example shown above relies on the fact that t_2 is not a regular term, however it can be adapted as follows to work also with regular types.

$$\begin{aligned}
&t_1 = obj(c, [f{:}t_a \vee t_b]) \vee obj(c, [f{:}t_a \vee t_b, g{:}t_1])\\
&t_2 = u_a \vee u_b\\
&u_a = obj(c, [f{:}t_a]) \vee obj(c, [f{:}t_a \vee t_b, g{:}u_a])\\
&u_b = obj(c, [f{:}t_b]) \vee obj(c, [f{:}t_a \vee t_b, g{:}u_b])
\end{aligned}$$

Type t_1 corresponds to the set of all finite (but not empty) and infinite lists with elements of type $t_a \vee t_b$ (where, again, $[\![t_a]\!] \neq \emptyset$, $[\![t_b]\!] \neq \emptyset$, $[\![t_a]\!] \cap [\![t_b]\!] = \emptyset$). Type u_a and u_b correspond to the sets of all finite (but not empty) and infinite lists with elements of type $t_a \vee t_b$ where, however, the last element (if the list is finite) has type t_a and t_b, respectively. Therefore $[\![t_1]\!] = [\![t_2]\!]$, but $t_1 \leq t_2$ is not provable. Indeed, since $[\![t_1]\!] \not\subseteq [\![u_a]\!]$ and $[\![t_1]\!] \not\subseteq [\![u_b]\!]$, by soundness we have that $t_1 \leq u_a$ and $t_1 \leq u_b$ cannot be proved, therefore a proof for $t_1 \leq t_2$ can only be obtained by first applying rule (split). Independently of the number of consecutive applications of rule (split), we have always to prove $t_1' \leq t_2$ for t_1' obtained from t_1 s.t. $[\![t_1']\!] \not\subseteq [\![u_a]\!]$ and $[\![t_1']\!] \not\subseteq [\![u_b]\!]$, hence the only possible prove we obtain is not contractive, since the depth of the applications of rule (split) is unbounded.

We overcome this problem by adding a new (merge) rule allowing to merge object types in the LHS types of the subtyping relation.

5.1 Rule (merge)

Before defining rule (merge), we provide the coinductive definition of the function $\wedge(t_1, t_2)$ which returns the type corresponding to the intersection of t_1 and t_2. Note that we are not introducing a new type constructior, but simply defining an auxiliary function used in rule (merge). Except for the intersection of object types of the same class, all other cases are trivial. The intersection of two object types, of the same class c, corresponds to an object type containing the union of all the fields, whose type is the intersection of the source types, for fields that are shared between the two object types, and the (single) source type for the others.

$$\wedge(int, int) = int \qquad \wedge(int, obj(\_,\_)) = \bot \qquad \wedge(obj(\_,\_), int) = \bot$$

$$\wedge(obj(c,\_), obj(c',\_)) = \bot \ \ (\text{if } c \neq c')$$

$$\wedge(t_1, t_2 \vee t_3) = \wedge(t_1, t_2) \vee \wedge(t_1, t_3) \qquad \wedge(t_1 \vee t_2, t_3) = \wedge(t_1, t_3) \vee \wedge(t_2, t_3)$$

$$\begin{array}{c}\wedge(obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_m{:}u_m]), obj(c, [f_1{:}t'_1, \ldots, f_n{:}t'_n, h_1{:}u'_1, \ldots, h_k{:}u'_k])) = \\ obj(c, [f_1{:} \wedge (t_1, t'_1), \ldots, f_n{:} \wedge (t_n, t'_n), g_1{:}u_1, \ldots, g_m{:}u_m, h_1{:}u'_1, \ldots, h_k{:}u'_k]) \\ \text{if } \{g_1, \ldots, g_m\} \cap \{h_1, \ldots, h_k\} = \emptyset\end{array}$$

Lemma 5.1 *For all types t_1, t_2, $[\![\wedge(t_1, t_2)]\!] \subseteq [\![t_1]\!]$ and $[\![\wedge(t_1, t_2)]\!] \subseteq [\![t_2]\!]$.*

Proof: (Sketch) The claim is equivalent to the following one: for all types t_1, t_2, and value v, if $v \in \wedge(t_1, t_2)$ then $v \in t_1$ and $v \in t_2$. The proof uses the same technique adopted in Theorem 4.1, even though in a much simpler way: first we coinductively define a function which takes a proof for $v \in \wedge(t_1, t_2)$ and returns two proofs for $v \in t_1$ and $v \in t_2$, respectively. Then, we prove that that such a function is well-defined and maps contractive proofs into contractive proofs. $\square$

Rule (merge) allows merging two object types in the LHS type t_2 having shape $u_1 \vee \ldots \vee t_1 \vee \ldots \vee t_2 \vee \ldots \vee u_n$ (see the definition of the rule and of the two and one holes contexts $\mathcal{M}[\ ,\ ]$ and $\mathcal{M}[\ ]$ in Figure 4), where $t_1 = obj(c, f{:}t_a, f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_m{:}u_m)$ and $t_2 = obj(c, f{:}t_b, f_1{:}t'_1, \ldots, f_n{:}t'_n, h_1{:}u'_1, \ldots, h_k{:}u'_k)$, with $\{g_1, \ldots, g_m\} \cap \{h_1, \ldots, h_k\} = \emptyset$. Of all the fields shared between these two object types, $f\, f_1 \ldots f_n$, the field f (recall

$$\begin{array}{rcl}\mathcal{M}[\ ] & ::= & \square \mid \mathcal{M}[\ ] \vee t \mid t \vee \mathcal{M}[\ ] \\ \mathcal{M}[\ ,\ ] & ::= & \mathcal{M}[\ ] \vee \mathcal{M}[\ ]\end{array}$$

$$(\text{merge})\ \frac{\begin{array}{c}t \leq \mathcal{M}[obj(c, [f{:}t_a \vee t_b, f_1{:} \wedge (t_1, t'_1), \ldots, f_n{:} \wedge (t_n, t'_n), \\ g_1{:}u_1, \ldots, g_m{:}u_m, h_1{:}u'_1, \ldots, h_k{:}u'_k]), \bot]\end{array}}{\begin{array}{c}t \leq \mathcal{M}[\, obj(c, f{:}t_a, f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_m{:}u_m), \\ obj(c, f{:}t_b, f_1{:}t'_1, \ldots, f_n{:}t'_n, h_1{:}u'_1, \ldots, h_k{:}u'_k)]\end{array}}\ \{g_1,\ldots,g_m\}\cap\{h_1,\ldots,h_k\}=\emptyset$$

Figure 4 – Definition of rule (merge) and contexts $\mathcal{M}[\ ,\ ]$ and $\mathcal{M}[\ ]$

that the order of fields is immaterial) is kept with a type that is the union of the source types, t_a and t_b, while the other source types are pairwise intersected by using the auxiliary function $\wedge$.

Note that for making rule (merge) more readable, in the premise we keep both holes of the context, even though the one replaced with the empty type $\bot$ would be deleted in practice; note also that replacing both holes with the same "merged" type would not be equivalent, since in this case the rule could be applied infinitely many times without "consuming" any type, hence, would not be contractive.

The following lemma is important for proving that the axiomatization remains sound when adding the new rule (merge).

Lemma 5.2 *The following inclusion holds:*

$$\begin{array}{c}[\![\mathcal{M}[obj(c, [f{:}t_a \vee t_b, f_1{:} \wedge (t_1, t'_1), \ldots, f_n{:} \wedge (t_n, t'_n), g_1{:}u_1, \ldots, g_m{:}u_m, h_1{:}u'_1, \ldots, h_k{:}u'_k]), \bot]]\!] \\ \subseteq \\ [\![\mathcal{M}[obj(c, f{:}t_a, f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_m{:}u_m), obj(c, f{:}t_b, f_1{:}t'_1, \ldots, f_n{:}t'_n, h_1{:}u'_1, \ldots, h_k{:}u'_k)]]\!]\end{array}$$

Proof: (Sketch) The claim is equivalent to the following one:
if $v \in \mathcal{M}[obj(c, [f{:}t_a \vee t_b, f_1{:}\wedge(t_1, t_1'), \ldots, f_n{:}\wedge(t_n, t_n'), g_1{:}u_1, \ldots, g_m{:}u_m, h_1{:}u_1', \ldots, h_k{:}u_k']), \bot]$
then $v \in \mathcal{M}[obj(c, f{:}t_a, f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_m{:}u_m), obj(c, f{:}t_b, f_1{:}t_1', \ldots, f_n{:}t_n', h_1{:}u_1', \ldots, h_k{:}u_k')]$.

The proof is by induction on the sum of the depth of the two holes in the context. The most interesting case is the proof of the basis. Obviously, v is an object of the shape

$$obj(c, [f \mapsto v_f, f_1 \mapsto v_1^f, \ldots, f_n \mapsto v_n^f, g_1 \mapsto v_1^g, \ldots, g_m \mapsto v_m^g, h_1 \mapsto v_1^h, \ldots, h_k \mapsto v_k^h])$$

By Lemma 5.1 we know that $\forall v_i^f \in [\![\wedge(t_i, t_i')]\!]$, with $i \in \{1, \ldots, n\}$, $v_i^f \in [\![t_i]\!] \cap [\![t_i']\!]$. Also, $v_f \in [\![t_a]\!]$ or $v_f \in [\![t_b]\!]$. Let us assume the former (the other case is symmetric), then $v \in [\![obj(c, f{:}t_a, f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_m{:}u_m)]\!]$ by applying rule (obj), ignoring fields h_i, with $i \in \{1, \ldots, k\}$. □

5.2 Completeness

In the rest of the paper we restrict the subtyping relation to regular types and proofs, and the interpretation of regular types to regular values and membership proofs.

Before investigating the completeness of our axiomatization of subtyping, we first show that there is a sound and complete axiomatization for the notion of non empty type (that is, t s.t. $[\![t]\!] \neq \emptyset$). Such an axiomatization is given by the following coinductive rules:

$$(\uparrow \vee\text{L})\frac{t_1 \uparrow_\bot}{t_1 \vee t_2 \uparrow_\bot} \qquad (\uparrow \vee\text{R})\frac{t_2 \uparrow_\bot}{t_1 \vee t_2 \uparrow_\bot} \qquad (\uparrow \text{int})\frac{}{int \uparrow_\bot} \qquad (\uparrow \text{obj})\frac{t_1 \uparrow_\bot, \ldots, t_n \uparrow_\bot}{obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n]) \uparrow_\bot}$$

We say that $t \uparrow_\bot$ holds iff there exists a contractive proof for $t \uparrow_\bot$ build by instantiating the rules above. A proof is contractive if it does not contain a subproof obtained by only applying rules ($\uparrow \vee$L) and ($\uparrow \vee$R).

Recall from Example 1 of Section 3 that $\bot$ is an abbreviation for the type t s.t. $t = t \vee t$, and that $[\![\bot]\!] = \emptyset$.

Proposition 5.1 *For all types t, $t \uparrow_\bot$ iff $[\![t]\!] \neq \emptyset$.*

Proof: The proof can be found in a companion paper (see Theorems 5.1 and 5.3 [3]). □

The following lemmas are instrumental to prove completeness.

Lemma 5.3 $[\![\mathcal{C}[t_1]]\!] \subseteq [\![\mathcal{C}[t_1 \vee t_2]]\!]$, $[\![\mathcal{C}[t_2]]\!] \subseteq [\![\mathcal{C}[t_1 \vee t_2]]\!]$.

Proof: By Def. 3.1 and Lemma 4.1, $v \in \mathcal{C}[t_1]$ implies ($v \in \mathcal{C}[t_1]$ or $v \in \mathcal{C}[t_2]$) which implies $v \in \mathcal{C}[t_1 \vee t_2]$ (and analogously for $[\![\mathcal{C}[t_2]]\!] \subseteq [\![\mathcal{C}[t_1 \vee t_2]]\!]$). □

Lemma 5.4 *If $[\![t]\!] = \emptyset$, then t contains $\bot$ as a subterm.*

Proof: By virtue of Proposition 5.1, the claim is equivalent to the following: if $t \not\uparrow_\bot$, then t contains $\bot$ as a subterm. For any type t, by direct coinduction, there always exists a (not necessarily contractive) proof of $t \uparrow_\bot$, therefore $t \not\uparrow_\bot$ implies that all proofs of $t \uparrow_\bot$ are not contractive, that is, contain a subproof obtained by only applying rules ($\uparrow \vee$L) and ($\uparrow \vee$R). By induction on the depth of the node we have that if $t' \uparrow_\bot$ is a node of the proof of $t \uparrow_\bot$, then t' is a subterm of t; therefore all non contractive proofs of $t \uparrow_\bot$ contains a node $t' \uparrow_\bot$ where t' is infinite and contains only $\vee$ nodes, hence $t' = \bot$, and t' is a subterm of t. □

Lemma 5.5 *If* $t = obj(c, [f_1{:}t_1, \ldots, f_n{:}t_{n+h}])$, $[\![t]\!] \subseteq [\![obj(c, [f_1{:}t'_1, \ldots, f_n{:}t'_n])]\!]$, *and* $[\![t]\!] \neq \emptyset$, *then* $[\![t_i]\!] \subseteq [\![t'_i]\!]$ *for all* $i = 1, \ldots, n$.

Proof: By Def. 3.1, if $[\![t]\!] \neq \emptyset$, then $[\![t_i]\!] \neq \emptyset$ for all $i = 1, \ldots, n{+}h$, hence there exists v_i s.t. $v_i \in [\![t_i]\!]$ for all $i = 1, \ldots, n{+}h$. Then, for any $k = 1, \ldots, n$, if $v \in [\![t_k]\!]$, then the value $v' = obj(c, [f_1 \mapsto v_1, \ldots, f_{k-1} \mapsto v_{k-1}, f_k \mapsto v, f_{k+1} \mapsto v_{k+1}, \ldots, f_{n+h} \mapsto v_{n+h}])$ is s.t. $v' \in [\![t]\!]$, therefore, by hypothesis, $v' \in [\![obj(c, [f_1{:}t'_1, \ldots, f_n{:}t'_n])]\!]$ and, hence, $v \in [\![t'_k]\!]$. □

For proving the main lemma 5.9 (given below) we first need some auxiliary definitions and lemmas. The following coinductive rules define an equivalence relation $\sim$ on values (which corresponds to bi-similarity between values w.r.t types):

$$(\text{int})\frac{}{i_1 \sim i_2} \qquad (\text{obj})\frac{v_1 \sim v'_1, \ldots, v_n \sim v'_n}{\begin{array}{c}obj(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]) \sim \\ obj(c, [f_1 \mapsto v'_1, \ldots, f_n \mapsto v'_n])\end{array}}$$

Lemma 5.6 *If* $v_1 \sim v_2$ *and* $v_1 \in t$ *then* $v_2 \in t$.

Proof: By case analysis on t and coinduction on the membership rules. □

Field paths (or simply paths, when no ambiguity may arise) are finite and possibly empty sequences of fields, inductively defined by the following productions:

$$p \quad ::= \quad \epsilon \mid f.p$$

The prefix $prefix(f.p)$ of a non-empty path $f.p$ is inductively defined as follows:

$$prefix(f.p) = \begin{cases} \epsilon & \text{if } p = \epsilon \\ f.prefix(p) & \text{otherwise} \end{cases}$$

If v is a value, then the path selection $v.p$ may either be undefined or denote a particular subvalue v' of v; in this case we write $v.p \rightsquigarrow v'$. The relation $v.p \rightsquigarrow v'$ is inductively defined in the standard way:

$$(\text{empty})\frac{}{v.\epsilon \rightsquigarrow v} \qquad (\text{not-empty})\frac{v_i.p \rightsquigarrow v'}{obj(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]).f_i.p \rightsquigarrow v'}\ 1{\leq}i{\leq}n$$

Each node $v' \in t'$ in a contractive proof ∇ for $v \in t$ can be uniquely associated with a field path p s.t. $v.p \rightsquigarrow v'$. Intuitively, such a field path is determined by the applications of rule (obj) in the path of the proof from its root $v \in t$ up to the node $v' \in t'$. Note that, conversely, a field path can be associated with several nodes; this happens when t contains union types. If t does not contain union types, then there is a bijection between the field paths and the nodes of the proof. The set of all field paths defined for a contractive proof ∇ is denoted by $fp(\nabla)$, and defined by $\{p \mid p \in \nabla\}$, where $p \in \nabla$ is coinductively (or inductively) defined as follows:

$$(\text{empty})\frac{}{\epsilon \in \nabla} \qquad (\vee)\frac{p \in \nabla}{p \in (\text{r})\frac{\nabla}{v \in t}}\ \text{r=}\vee\text{L or r=}\vee\text{R}$$

$$(\text{obj})\frac{p \in \nabla_i}{f_i.p \in (\text{obj})\frac{\nabla_1, \ldots, \nabla_n}{v \in obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])}}\ 1{\leq}i{\leq}n, n{>}0$$

Rules (empty) and (obj) are contractive, whereas ($\vee$) is not, however if ∇ is contractive, then there may exist only contractive proofs for $p \in \nabla$. Note that, since field paths are finite, in this case all proofs for $p \in \nabla$ are finite, hence the rules above can be indifferently interpreted coinductively or inductively.

Analogously, if v is a value, then $\mathit{fp}(v)$ is the set of all field paths corresponding to the subvalues of v (included itself), inductively defined by the following rules.

$$(\text{empty})\frac{}{\epsilon \in v} \qquad (\text{obj})\frac{p \in v_i}{f_i.p \in \mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n])}\ 1 \le i \le n, n > 0$$

Lemma 5.7

1. *The sets $\mathit{fp}(\nabla)$ and $\mathit{fp}(v)$ are prefix-closed.*
2. *If ∇ is a proof for $v \in t$, then $\mathit{fp}(\nabla) \subseteq \mathit{fp}(v)$.*

Proof:

1. The proof that $\mathit{fp}(v)$ is prefix-closed is by induction on the length of the path. If $f.\epsilon \in v$, then by definition $\epsilon \in v$. If $f.p \in v$, with $p \neq \epsilon$, then necessarily $v = \mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n])$, $f = f_i$ for $1 \le i \le n$, and $p \in v_i$. By inductive hypothesis, $\mathit{prefix}(p) \in v_i$, hence by rule(obj), $f.\mathit{prefix}(p) \in v$, but $f.\mathit{prefix}(p) = \mathit{prefix}(f.p)$.

   The proof that $\mathit{fp}(\nabla)$ is prefix-closed is by induction on the rules defining $p \in \nabla$, and case analysis on the first instantiated rule. The first instantiated rule cannot be (empty), since $p \neq \emptyset$ (*prefix* is defined only for non empty paths). If the first instantiated rule is ($\vee$), then we can directly conclude by inductive hypothesis. If the first instantiated rule is (obj), then $f = f_i$ for $1 \le i \le n$, and we distinguish two subcases: if $p = f.\epsilon$, then we can conclude by virtue of rule (empty); otherwise, $p = f.p'$ with $p' \neq \epsilon$, and by inductive hypothesis, $\mathit{prefix}(p') \in \nabla_i$, hence by rule(obj), $f.\mathit{prefix}(p') \in \nabla$, but $f.\mathit{prefix}(p') = \mathit{prefix}(f.p')$.

2. The proof is by induction on the rules defining $p \in \nabla$, and case analysis on the first instantiated rule. If the first instantiated rule is (empty), then $p = \emptyset$ and we conclude by instantiating the corresponding rule to obtain $\epsilon \in v$. If the first instantiated rule is ($\vee$), then by inductive hypothesis $p \in v$. Finally, if the first instantiated rule is (obj), then $p = f_i.p'$ for $1 \le i \le n$, $v = \mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots])$ and $t = \mathit{obj}(c, [f_1{:}t_1, \ldots, f_n{:}t_n])$; by inductive hypothesis, $p' \in v_i$, hence we conclude by instantiating the corresponding rule to obtain $f_i.p' \in v$.

$\square$

Lemma 5.8 *If t does not contain union types and $v_1, v_2 \in t$, then:*

1. *there exists a unique proof ∇_1 for $v_1 \in t$ and ∇_2 for $v_2 \in t$, and $\mathit{fp}(\nabla_1) = \mathit{fp}(\nabla_2)$;*
2. *if $\mathit{fp}(v_1) = \mathit{fp}(\nabla_1)$ and $\mathit{fp}(v_2) = \mathit{fp}(\nabla_2)$, then $v_1 \sim v_2$.*

Proof:

1. This is a direct consequence of the fact that membership rules (int) and (obj) are disjointly applicable.

2. The proof is by case analysis on t and coinduction on the rules defining $v_1 \sim v_2$. If $t = int$, then $v_1, v_2 \in \mathbb{Z}$ and we conclude $v_1 \sim v_2$ by rule (int). Otherwise $t = obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])$, and by the hypothesis $fp(v_1) = fp(\nabla_1)$ and $fp(v_2) = fp(\nabla_2)$ we deduce that v_1 and v_2 are object values containing exactly the same fields as those in t, hence, $v_1 = obj(c, [f_1 \mapsto v_1^1, \ldots, f_n \mapsto v_n^1])$, $v_2 = obj(c, [f_1 \mapsto v_1^2, \ldots, f_n \mapsto v_n^2])$. Since $fp(v_1) = fp(\nabla_1)$ and $fp(v_2) = fp(\nabla_2)$, then $fp(v_i^1) = fp(\nabla_i^1)$ and $fp(v_i^2) = fp(\nabla_i^2)$ for all $i = 1, \ldots, n$; indeed, by contradiction, if there exists $p \in v_i^k$ s.t. $p \notin \nabla_i^k$ (for any $i = 1, \ldots, n$, $k = 1, 2$), then $f_i.p \in v_k$, but $f_i.p \notin \nabla_k$, and hence $fp(v_k) \neq fp(\nabla_k)$. Therefore we can conclude by applying the coinductive hypothesis to rule (obj) to obtain $v_1 \sim v_2$.

□

If ∇ is a proof of $v \in t$ and $fp(\nabla) = fp(v)$, then v is minimal w.r.t. t: any subvalue of v is essential for proving $v \in t$, if we remove any of them we obtain a value v' s.t. $v' \in t$ is no longer provable.

If P is a (possibly empty) set of non empty paths, then $P \downarrow f$ denotes the set $\{p \mid f.p \in P\}$. If v is a value, and $P \subseteq fp(v)$, then the restriction $v \setminus P$ is the value coinductively defined as follows:

$$
\begin{aligned}
&i \setminus P = i \text{ if } i \in \mathbb{Z}\\
&v \setminus P = obj(c, [f_{k_1} \mapsto v_{k_1} \setminus (P \downarrow f_{k_1}), \ldots, f_{k_m} \mapsto v_{k_m} \setminus (P \downarrow f_{k_m})]) \text{ if}\\
&\qquad v = obj(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]) \text{ and } \{f_1.\epsilon, \ldots, f_n.\epsilon\} \setminus P = \{f_{k_1}.\epsilon, \ldots, f_{k_m}.\epsilon\}
\end{aligned}
$$

Intuitively, $v \setminus P$ is the value v' obtained by removing from v all subvalues determined by the non empty paths in P.

Lemma 5.9 *$[\![t_1]\!] \subseteq [\![u_1 \vee u_2]\!]$, $[\![t_1]\!] \not\subseteq [\![u_1]\!]$, $[\![t_1]\!] \not\subseteq [\![u_2]\!] \Rightarrow t_1$ contains a union type.*

Proof: We arrange the claim in the following equivalent form: if $[\![t_1]\!] \subseteq [\![u_1 \vee u_2]\!]$, and t_1 does not contain union types, then $[\![t_1]\!] \subseteq [\![u_1]\!]$ or $[\![t_1]\!] \subseteq [\![u_2]\!]$.

We first observe that by Lemma 5.4, $[\![t_1]\!] \neq \emptyset$, and that by definition of $[\![\ ]\!]$ and of membership, $[\![u_1 \vee u_2]\!] = [\![u_1]\!] \cup [\![u_2]\!]$. If $[\![u_1]\!] = \emptyset$ or $[\![u_2]\!] = \emptyset$, then the claim trivially holds. The proof is less simple when $[\![u_1]\!] \neq \emptyset$ and $[\![u_2]\!] \neq \emptyset$. Let us assume by contradiction that $[\![t_1]\!] \not\subseteq [\![u_1]\!]$ and $[\![t_1]\!] \not\subseteq [\![u_2]\!]$, that is, there exist $v_1, v_2 \in [\![t_1]\!]$ s.t. $v_1 \in [\![u_1]\!] \setminus [\![u_2]\!]$, $v_2 \in [\![u_2]\!] \setminus [\![u_1]\!]$.

Since t_1 does not contain union types, by Lemma 5.8(1) there exists a unique proof ∇_1 for $v_1 \in t_1$ and ∇_2 for $v_2 \in t_1$, and $fp(\nabla_1) = fp(\nabla_2) = P$; however, in general $fp(v_1) \neq P \neq fp(v_2)$, but rather $P \subseteq fp(v_1), fp(v_2)$. Nevertheless, it is possible to find v_1', v_2' s.t. $v_1', v_2' \in [\![t_1]\!]$, $v_1' \in [\![u_1]\!] \setminus [\![u_2]\!]$, $v_2' \in [\![u_2]\!] \setminus [\![u_1]\!]$, $fp(v_1') = fp(v_2') = P$. Indeed, v_1' (and analogously v_2') can be obtained by removing from v_1 all subvalues determined by the paths $p \in fp(v_1) \setminus P$ s.t. $prefix(p) \in P$. More precisely, $v_1' = v_1 \setminus \{p \in fp(v_1) \setminus P \mid prefix(p) \in P\}$ and $v_2' = v_2 \setminus \{p \in fp(v_2) \setminus P \mid prefix(p) \in P\}$. Since removed fields are not checked in the proof ∇_1 for $v_1 \in t_1$, then there exists a valid proof for $v_1' \in t_1$. Furthermore, since $v_1 \notin [\![u_2]\!]$, then $v_1' \notin [\![u_2]\!]$, therefore $v_1' \in [\![u_1]\!]$, since $[\![t_1]\!] \subseteq [\![u_1]\!] \cup [\![u_2]\!]$. Similar considerations apply for v_2' as well. Hence, by Lemma 5.8(2) $v_1' \sim v_2'$ and by Lemma 5.6 we obtain the contradiction $v_1' \in [\![u_2]\!]$ and $v_2' \in [\![u_1]\!]$. □

Theorem 5.1 (Completeness, step one) *For all regular types t_1, t_2, if $[\![t_1]\!] \subseteq [\![t_2]\!]$, then there exists a (not necessarily contractive) proof of $t_1 \leq t_2$.*

$t_1 \backslash t_2$	*int*	$\vee$	*obj*
int	(int)	($\vee$R1) or ($\vee$R2) Lemma 5.9	vacuous
$\vee$	(split) Lemma 5.3	(split), ($\vee$R1), ($\vee$R2) or (merge) Lemma 5.3	(split) Lemma 5.3
obj	(split) Lemma 5.4	(split), ($\vee$R1), ($\vee$R2) or (merge) Lemma 5.3, 5.9 and 5.4	(split) or (obj) Lemma 5.4 and 5.5

Table 1 – Applied rules

Proof: The proof is by coinduction on the definition of $\leq$ and by case analysis on the top level type constructors of both t_1 and t_2. Table 1 summarizes the rules which can be applied for each case, and the needed auxiliary lemmas.

1. $t_1 = int$, $t_2 = int$: one can conclude by trivially applying rule (int).

2. $t_1 = int$, $t_2 = u_1 \vee u_2$: since t_1 does not contain union types, by Lemma 5.9 we deduce that either $[\![t_1]\!] \subseteq [\![u_1]\!]$ or $[\![t_1]\!] \subseteq [\![u_2]\!]$. If only $[\![t_1]\!] \subseteq [\![u_1]\!]$ holds, then we conclude by coinductive hypothesis and rule ($\vee$R1); conversely, if only $[\![t_1]\!] \subseteq [\![u_2]\!]$ holds, then we conclude by coinductive hypothesis and rule ($\vee$R2). If both $[\![t_1]\!] \subseteq [\![u_1]\!]$ and $[\![t_1]\!] \subseteq [\![u_2]\!]$ hold, then not always rules ($\vee$R1) and ($\vee$R2) are applicable interchangeably. Consider for instance the case where $t_2 = t_2 \vee int$; if rule ($\vee$R1) were always be chosen when both rules are applicable, then we would end up with a non contractive proof obtained by only applying rule ($\vee$R1). To solve this problem, we need to avoid applying rule ($\vee$R1) or ($\vee$R2) if u_1 or u_2, respectively, have the following shape: all paths either contain only constructor $\vee$ or lead to t_2 through only constructor $\vee$. More precisely, we need to avoid applying rule ($\vee$R1) or ($\vee$R2) if $cyclic(u_1)$ or $cyclic(u_2)$ holds, respectively, where *cyclic* is defined by the following pseudo-code (we assume that initially all the nodes of the term are not marked):

```
cyclic(u) {
   if (marked(u) || u==t₂)
      return true;
   else if (u==u′ ∨ u″) {
      mark(u);
      return cyclic(u′) && cyclic(u″);
   }
   else
      return false;
}
```

   Note that since $[\![int]\!] \neq \emptyset$, and $cyclic(u_1)$ and $cyclic(u_2)$ implies $u_1 \vee u_2 = \bot$, then it can never happen that *cyclic* holds on both u_1 and u_2.

3. $t_1 = int$, $t_2 = obj(c, [f_1 \mapsto u_1, \ldots, f_n \mapsto u_n])$: this case is vacuous since $[\![t_1]\!] \not\subseteq [\![t_2]\!]$.

4. $t_1 = u_1 \vee u_2$, $t_2 = int$: by Lemma 5.3 (applied when $\mathcal{C}[\,] = \Box$) we have $[\![u_1]\!] \subseteq [\![t_1]\!]$, $[\![u_2]\!] \subseteq [\![t_1]\!]$, hence, by transitivity, $[\![u_1]\!] \subseteq [\![t_2]\!]$, $[\![u_2]\!] \subseteq [\![t_2]\!]$, therefore we conclude by coinductive hypothesis and by rule (split) (instantiated with $\mathcal{C}[\,] = \Box$).

$$
\text{(merge-0)}\,\dfrac{\text{(merge-1)}\,\dfrac{\text{(split-0)}\,\dfrac{\text{(}\vee\text{ R1)}\,\dfrac{\begin{array}{c} \mathit{obj}(c,[f{:}t_a \vee t_b]) \leq \\ \mathit{obj}(c,[f{:}t_a \vee t_b]) \end{array}}{\mathit{obj}(c,[f{:}t_a \vee t_b]) \leq t_3} \qquad \text{(}\vee\text{ R2)}\,\dfrac{\text{(obj)}\,\dfrac{t_a \vee t_b \leq t_a \vee t_b \quad t_1 \leq t_2}{\begin{array}{c} \mathit{obj}(c,[f{:}t_a \vee t_b, g{:}t_1]) \leq \\ \mathit{obj}(c,[f{:}t_a \vee t_b, g{:}t_2]) \end{array}}}{\mathit{obj}(c,[f{:}t_a \vee t_b, g{:}t_1]) \leq t_3}}{t_1 \leq \mathit{obj}(c,[f{:}t_a \vee t_b]) \vee \mathit{obj}(c,[f{:}t_a \vee t_b, g{:}t_2]) \equiv t_3}}{t_1 \leq \mathit{obj}(c,[f{:}t_a \vee t_b]) \vee \mathit{obj}(c,[f{:}t_a \vee t_b, g{:}u_a]) \vee \mathit{obj}(c,[f{:}t_a \vee t_b, g{:}u_b])}}{t_1 \leq t_2}
$$

Figure 5 – Regular proof tree for the counter-example using (merge)

5. $t_1 = u_1 \vee u_2$, $t_2 = u'_1 \vee u'_2$: if $[\![t_1]\!] = \emptyset$, then we proceed as for case 4 by applying rule (split). If $[\![t_1]\!] \neq \emptyset$ and $[\![t_1]\!] \subseteq [\![u'_1]\!]$ or $[\![t_1]\!] \subseteq [\![u'_2]\!]$, then we can proceed as for case 2 by applying rules ($\vee$R1) or ($\vee$R2). If $[\![t_1]\!] \neq \emptyset$, $[\![t_1]\!] \not\subseteq [\![u'_1]\!]$, and $[\![t_1]\!] \not\subseteq [\![u'_2]\!]$, then there are two possible cases:
   (a) if rule (merge) is applicable with premise $t_1 \leq t'_2$ (where t'_2 is the term obtained from t_2 by applying (merge)) with t'_2 s.t. $[\![t_1]\!] \subseteq [\![t'_2]\!]$, then we conclude by coinductive hypothesis.
   (b) if the previous condition is not satisfied, then we proceed as for case 4 by applying rule (split).
6. $t_1 = u_1 \vee u_2$, $t_2 = \mathit{obj}(c, [f_1 \mapsto u'_1, \ldots, f_n \mapsto u'_n])$: we proceed as for case 4 by applying rule (split).
7. $t_1 = \mathit{obj}(c, [f_1 \mapsto u_1, \ldots, f_n \mapsto u_n])$, $t_2 = \mathit{int}$: in this case $[\![t_1]\!] \subseteq [\![t_2]\!]$ only if $[\![t_1]\!] = \emptyset$. By Lemma 5.4 t_1 contains $\perp$ as a subterm, hence rule (split) is applicable and we proceed as in the subcase of 5 when $[\![t_1]\!] = \emptyset$.
8. $t_1 = \mathit{obj}(c, [f_1 \mapsto u_1, \ldots, f_n \mapsto u_n])$, $t_2 = u'_1 \vee u'_2$: this case is almost the same as 5, except for the subcase (b), for which one has to prove that rule (split) is applicable; this is the case, by virtue of Lemma 5.9.
9. $t_1 = \mathit{obj}(c, [f_1 \mapsto u_1, \ldots, f_n \mapsto u_n])$, $t_2 = \mathit{obj}(c', [f'_1 \mapsto u'_1, \ldots, f'_k \mapsto u'_k])$: if $[\![t_1]\!] = \emptyset$, then we proceed as in case 7; otherwise, by definition of $[\![t]\!]$ for object types, we have that $c = c'$, and $\{f'_1, \ldots, f'_k\} \subseteq \{f_1, \ldots, f_n\}$; finally, by Lemma 5.5 we can conclude by coinductive hypothesis and rule (obj).

The second step of the proof is the most awkward and consists in showing that the proof built in Theorem 5.1 is always contractive, The proof relies on the following conjecture.

Conjecture 5.1 (Final decomposition) *If $u_1 \vee u_2$ is a regular type, $[\![t]\!] \subseteq [\![u_1 \vee u_2]\!]$, $[\![t]\!] \not\subseteq [\![u_1]\!]$, $[\![t]\!] \not\subseteq [\![u_2]\!]$, no merge can be performed on $u_1 \vee u_2$ to obtain a type u s.t. $[\![t]\!] \subseteq [\![u]\!]$, then it is possible to decompose t, by iterative splits, into $t_1, \ldots, t_n$ s.t. $[\![t]\!] = \bigcup_{i=1,\ldots,n} [\![t_i]\!]$ and for all $i = 1, \ldots, n$ $[\![t_i]\!] \subseteq [\![u_1]\!]$ or $[\![t_i]\!] \subseteq [\![u_2]\!]$.*

Theorem 5.2 (Completeness, step two) *For all regular types t_1, t_2, if $[\![t_1]\!] \subseteq [\![t_2]\!]$, then the proof of $t_1 \leq t_2$ built as in Theorem 5.1 is contractive.*

Figure 5 shows that our second counter-example, explained on page 14, ceases to be a counter-example if we add rule (merge), described in Section 5.1, to our system.

In the example, in trying to derive $t_1 \leq t_2$, we can apply rule (merge) only twice, because if we applied (merge) three times, that is, if we merged the two operands of the outer union of t_3, we would obtain the type $t_4 \equiv obj(c, [f{:}t_a \vee t_b, g{:}t_2])$ that is not a supertype of t_1; indeed, if we consider $v = obj(c, [f \mapsto v_a])$, with $v_a \in [\![t_a]\!]$, it is easy to see that $v \in [\![t_1]\!]$ but $v \notin [\![t_4]\!]$, so $t_1 \leq t_4$ cannot be deduced because it would not be sound.

After having applied (merge) twice we are in the situation described in Conjecture 5.1, in fact, we can iteratively split t_1 and conclude by using rules ($\vee$R1), ($\vee$R2), (obj) and the reflexivity of the subtype relation (that can be trivially proved).

6 Conclusion

We have addressed the problem of defining a sound and complete axiomatization of a subtyping relation between coinductive object and union types, defined as set inclusion between type interpretations.

We have shown that the axiomatization of subtyping proposed in our previous work [4] is not complete even when regular types and proofs are considered. To overcome this problem, we have extended the axiomatization by adding a new rule (merge), we have proved that the new axiomatization is still sound and is complete when one considers only regular types and proofs. Proof of completeness relies on a conjecture which is weaker then the one previously proposed [4] and disproven in this paper. Interestingly enough, if such a conjecture is correct, then one can easily devise a semi-computable procedure which returns a contractive and regular proof tree for $t_1 \leq t_2$ whenever $[\![t_1]\!] \subseteq [\![t_2]\!]$. However when $[\![t_1]\!] \not\subseteq [\![t_2]\!]$, the procedure may terminate with a correct failure, but may also diverge. Besides trying to prove such a conjecture, we leave open for future work the question whether complete subtyping between regular types is decidable. An interesting approach to investigate could be the one proposed by Damm [12] for proving decidability of recursive types (with intersection, union, and function types, but without record types), based on inclusion between regular tree expressions.

References

[1] O. Agesen. The cartesian product algorithm. In W. Olthoff, editor, *ECOOP'05 - Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995.

[2] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP'09 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2009.

[3] D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In Montanari A., Napoli M., and Parente M., editors, *Proceedings of GandALF 2010*, volume 25 of *Electronic Proceedings in Theoretical Computer Science*, pages 214–223, 2010.

[4] D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. In *Formal Techniques for Java-like Programs (FTfJP10)*, ACM Digital Library, 2010.

[5] D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 2010. To appear.

[6] D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *Post-Proceedings of TYPES'08*, number 5497 in Lecture Notes in Computer Science. Springer, 2009.

[7] Davide Ancona, Andrea Corradi, Giovanni Lagorio, and Ferruccio Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, Lecture Notes in Computer Science. Springer, 2010. Selected paper to appear in the post-proceedings.

[8] F. Barbanera, M. Dezani-Cincaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[9] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *TLCA '97 - Typed Lambda Calculi and Applications*, pages 63–81, 1997.

[10] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inform.*, 33(4):309–338, 1998.

[11] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[12] Flemming Damm. Subtyping with union types, intersection types and recursive types. In *Theoretical Aspects of Computer Software (TACS'94)*, pages 687–706. Springer-Verlag, 1994.

[13] M. Furr, J. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied computing.* ACM Press, 2009.

[14] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *Journ. of Object Technology*, 6(2):47–68, 2007.

[15] N.Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *ECOOP'92 - European Conference on Object-Oriented Programming*, pages 329–349, 1992.

[16] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 146–161. ACM Press, 1991.

[17] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, pages 472–483, 2007.

[18] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *Logic Programming, 22nd International Conference, ICLP 2006*, pages 330–345, 2006.

[19] M. Sulzmann and P. J. Stuckey. HM(X) type inference is CLP(X) solving. *Journ. of Functional Programming*, 18(2):251–283, 2008.

[20] T. Wang and S. Smith. Polymorphic constraint-based type inference for objects. Technical report, The Johns Hopkins University, 2008. Submitted for publication.

[21] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01 - European Conference on Object-Oriented Programming*, volume 2072, pages 99–117. Springer, 2001.