Complete coinductive subtyping for abstract compilation of object-oriented languages*

Davide Ancona
DISI, University of Genova
Italy
davide@disi.unige.it

Giovanni Lagorio
DISI, University of Genova
Italy
lagorio@disi.unige.it

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 complete subtyping relation $\leq$ between types built on object and union type constructors: we interpret types as sets of values, and investigate on a definition of subtyping such that $t_1 \leq t_2$ is derivable whenever the interpretation of t_1 is contained in the interpretation of t_2. 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 [5, 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 [14, 13, 1, 18, 17, 11] 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 [4] that a more precise type analysis can be obtained when abstract compilation is performed on programs in Static Single Assignment intermediate form [10].

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 the definition of a suitable subtyping relation on coinductive union and object types is very important to investigate on possible implementations of the inference engine of our framework, based on previous work on coinductive SLD resolution [16, 15] extended with subtyping and subsumption. However, devising a sound definition of subtyping on coinductive types is far from being intuitive, since a suitable notion of *contractive* [7, 8] 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 definition 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 [4], and to add a new typing rule for dealing with a case previously uncovered.

In this paper we show that, despite the previous improvements, we have not succeeded yet in obtaining completeness. To this aim, we propose an extension of the previous definitions of subtyping which turns out to be more expressive. We prove that such a definition is still sound, even though it still fails to be complete. However, we conjecture that such a definition is complete in the most interesting case from the practical point of view: when one restricts the system to regular types and derivations.

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

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 [9, 16, 15]. 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 *has_meth* defines the usual method look-up: $has\_meth(c, m, [this, t_1, \ldots, t_n], t)$ succeeds if look-up of m from class c succeeds and returns a method that, when invoked on target object and arguments $this, t_1, \ldots, t_n$, 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 $obj(C, R)$, then invocation of M with arguments A is correct if look-up of M with first argument $obj(C, R)$, corresponding to *this*, and rest of arguments A succeeds when starting from class C.

We show now that the goal $invoke(nat, twice, [\,], R)$ is derivable for $R = evn$. 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 derivations [16]. Then, since $nat = zer \vee obj(succ, [pred{:}nat])$, by clause 1 for *invoke* the following atoms must be derivable:

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

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

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

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

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

Surprisingly and regrettably, $invoke(evn, twice, [\,], evn)$ is not derivable, since $new(succ, [t_1], t_2)$ 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

[1]We refer to the author's previous work [5, 2, 4] for more details.

$$(\text{int})\frac{}{i \in \mathit{int}} \qquad (\vee\text{L})\frac{v \in t_1}{v \in t_1 \vee t_2} \qquad (\vee\text{R})\frac{v \in t_2}{v \in t_1 \vee t_2} \qquad (\text{obj})\frac{v_1 \in t_1, \ldots, v_n \in t_n}{\mathit{obj}(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots]) \in \mathit{obj}(c, [f_1{:}t_1, \ldots, f_n{:}t_n])}$$

Figure 1: Rules defining membership

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 [16] is generalized by taking into account subtyping constraints between terms, besides the usual unification constraints.

3. TYPE INTERPRETATION AND SUBTYPING

In this section we define object and union coinductive types, and provide an intuitive interpretation of types as sets of values. Then we define subtyping as a syntactic relation between types, and prove that such a relation is complete[2] w.r.t. containment between type interpretations.

3.1 Interpretation of types

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 finite, distinct and that their order is immaterial. Union types $t_1 \vee t_2$ have the standard meaning [6, 12].

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 $i \in \mathbb{Z}$).

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

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).

[2]Soundness can be obtained by a simple generalization of the proof [3] provided for a previous non complete definition of the subtyping relation.

A derivation is a tree where each node is a pair consisting of a judgment of the shape $v \in t$, and of a rule label[3], and where each node, together with its children, corresponds to a valid instantiation of a rule. For instance, the following tree is a derivation 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 derivations are allowed to be infinite. However, not all infinite derivations 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 derivation 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 derivations built applying only rules ($\vee$L) and ($\vee$R), since they allow unsound judgments, as $\mathit{obj}(c, [\,]) \in t$ derived above, since t corresponds to an infinite union of int, and therefore its interpretations should only contain integer 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 [7, 8], we say that rules (int) and (obj) are contractive, whereas ($\vee$L) and ($\vee$R) are not.

DEF. 3.1. *A derivation for $v \in t$ is contractive iff it contains no sub-derivations built only with membership rules ($\vee$R), and ($\vee$L). The membership relation $v \in t$ holds iff there is a contractive derivation 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 *derivation* for contractive derivations, unless explicitly specified.

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 derivations can be built, therefore no judgments of the shape $v \in \bot$ can be derived.

Example 2.

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

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

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

$$\mathcal{C}[\,] \;::=\; \Box \mid obj(c, [f{:}\mathcal{C}[\,], f_1{:}t_1, \ldots, f_n{:}t_n])$$

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

Figure 2: Rules defining the subtyping relation

decisive applications of rule (∨R) and (int):

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

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

3.2 Definition of subtyping

The subtyping relation is coinductively defined by the rules in Figure 2, conceived for a purely functional setting [2]; an extension for dealing with imperative features can be found in another paper [4] by the same authors.

All the previous definitions of the subtyping relation [5, 2, 3], are not complete w.r.t. containment between type interpretations, though the weakest, and also most recent, definition [3] has been proved to be sound.

The difference with the last previous attempt is in rule (split) which is, in fact, a generalization with contexts of rules (∨L) and (distr) defined in the previous system [3].

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

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

The one hole context (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 at least one 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 $t_1 \vee t_2 \leq t$, then the rule can be instantiated only with the empty context.

Rule (∨L) is simply obtained by instantiating (split) with the empty context. Rules (∨R1), (∨R2) and (∨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 [4]. 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 $obj(c, [f{:}\Box, f_1{:}t_1, \ldots, f_n{:}t_n])$, where $\Box$ 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 $u_1 = obj(c, [f{:}t_1]) \vee obj(c, [f{:}t_2])$ and $u_2 = obj(c, [f{:}t_1 \vee t_2])$, then $[\![u_1]\!] = [\![u_2]\!]$, hence if completeness holds, one should be able to derive $u_1 \cong u_2$, that is, $u_1 \leq u_2$ and $u_2 \leq u_1$. The relation $u_1 \leq u_2$ can be derived by applying rules (∨L), (obj), (∨R1) and (∨R2), and by reflexivity[4]. Rule (distr) is necessary for deriving the opposite direction $u_2 \leq u_1$ of the relation, since by applying rules (∨R1), (∨R2) and (obj) we end up with $t_1 \vee t_2 \leq t_1$ or $t_1 \vee t_2 \leq t_2$ which in general do not hold.

For reasons similar to those explained in the previous section, derivations have to be contractive, otherwise unsound judgments can be derived. For instance, if $t = t \vee int$, then we could derive $obj(c, [\,]) \leq t$ by only applying rule (∨R1). However, giving a definition of contractive derivation for subtyping which does not break soundness without compromising completeness (at least when the system is restricted to regular types and derivations) is not trivial. In this case Def. 3.1 alone does not ensure soundness. For instance, if t is the term s.t. $t = t \vee obj(c, [f{:}t])$, then we can build a derivation for the clearly unsound judgment $t \leq int$ by using only rule (split) as shown in Figure 3. We have added to the label of each rule application a natural number corresponding to the depth of the hole of the context used for applying rule (split); the problem with the shown derivation 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. 3.2. *The depth of a context $\mathcal{C}[\,]$ is inductively defined as follows:*

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

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

DEF. 3.3. *A derivation for $t_1 \leq t_2$ is contractive iff it contains no sub-derivations built only with subtyping rules (∨R), and (∨L), and the depth of the applications of rule (split) is bounded.*

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

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

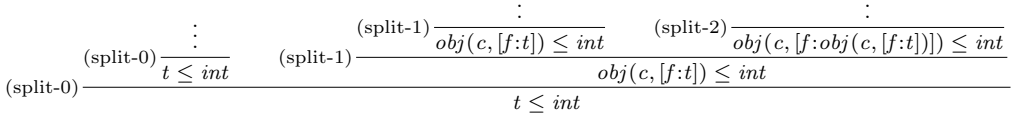

Figure 3: A non contractive derivation for $t \leq int$, with $t = t \vee obj(c, [f{:}t])$.

3.3 Soundness and completeness

We can now state the main claims about subtyping. Soundness holds in the most general case, when types and derivations are allowed to be non regular.

THEOREM 3.1 (SOUNDNESS). *For all t_1, t_2, if $t_1 \leq t_2$ is derivable, then $[\![t_1]\!] \subseteq [\![t_2]\!]$.*

Soundness of subtyping has been already proved [3], however the proof needs to be upgraded since here we are considering a larger relation. However, the proof we have proposed can be adapted, with some efforts, to our new definition of subtyping. The proof, which is not trivial, basically consists in coinductively defining a function which takes two derivations, for $v \in t_1$ and $t_1 \leq t_2$, and returns a derivation for $v \in t_2$. The ability of defining such a function relies on a main lemma which can be generalized here as follows.

LEMMA 3.1. *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 $u \leq t_2$ for a derivation whose first applied rule is not (split).*

We omit here the proof of this lemma, as well as all the other details of the proof of soundness, for space limitation. The main idea of the proof of the lemma is that u can be found by following a certain path in the derivation of $t_1 \leq t_2$ determined by the derivation of $v \in t_1$, until a rule different from (split) is applied. The proof that such a path cannot contain only applications of rule (split) relies on the notion of contractiveness as given in Def. 3.3, and on the definition of a specific Noetherian order which ensures that the number of consecutive applications of rule (split) is always finite.

Completeness does not hold for the idealized system where types and derivations can also be non regular. Consider for instance the terms defined by the following infinite set of equations:

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

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-th element has type t_b, and all preceding elements have type t_a, therefore $[\![u_b]\!] = [\![t_1]\!] \setminus [\![u_a]\!]$. Even though $[\![t_1]\!] = [\![t_2]\!]$, the only derivations 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 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 derivations 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 we conjecture that completeness holds if the system is restricted to regular types and derivations.

We denote with $\leq_r$ the subtyping relation restricted to regular types and derivations, and with $[\![t]\!]_r$ the interpretation of regular types restricted to regular values and membership derivations. Since, by definition, regular derivations can only contain a finite number of sub-derivations, it follows that in a regular derivation the depth of applications of rule (split) is always bounded. Therefore a regular derivation for $t_1 \leq_r t_2$ is contractive iff it contains no sub-derivations built only with subtyping rules ($\vee$R), and ($\vee$L), similarly to what is required in Def. 3.1.

CONJECTURE 3.1 (COMPLETENESS). *For all regular types t_1, t_2, if $[\![t_1]\!]_r \subseteq [\![t_2]\!]_r$, then $t_1 \leq_r t_2$ is derivable.*

PROOF. (Sketch) The proof is in two steps. First we prove that if $[\![t_1]\!]_r \subseteq [\![t_2]\!]_r$, then we can build a derivation for $t_1 \leq_r t_2$, and then we show that the proof is in fact a method for generating only regular and contractive derivations.

Step one is proved by coinduction on the definition of $\leq$ and by case analysis on the top level type constructors of both t_1 and t_2. The Table 3.3 summarizes the rules which are applied for each case, and the auxiliary lemmas which are needed (whose claims are shown below).

Claim of lemmas (recall that all terms are regular, even though most of the lemmas hold also for the most general case):

1. $\mathbb{Z} \subseteq [\![t_1 \vee t_2]\!] \Rightarrow \mathbb{Z} \subseteq [\![t_1]\!]$ or $\mathbb{Z} \subseteq [\![t_2]\!]$.
2. $[\![\mathcal{C}[t_1]]\!] \subseteq [\![\mathcal{C}[t_1 \vee t_2]]\!]$ and $[\![\mathcal{C}[t_2]]\!] \subseteq [\![\mathcal{C}[t_1 \vee t_2]]\!]$.
3. t_1 object type s.t. $[\![t_1]\!] \subseteq [\![u_1 \vee u_2]\!]$, $[\![t_1]\!] \not\subseteq [\![u_1]\!]$, $[\![t_1]\!] \not\subseteq [\![u_2]\!] \Rightarrow t_1$ contains a union type.
4. t_1 object type s.t. $[\![t_1]\!] = \emptyset \Rightarrow \exists \mathcal{C}[\,]$ s.t. $t_1 = \mathcal{C}[\bot]$.
5. Let $t = obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n, \ldots])$.
   Then, $[\![t]\!] \neq \emptyset$ and $[\![t]\!] \subseteq [\![obj(c, [f_1{:}t'_1, \ldots, f_n{:}t'_n])]\!] \Rightarrow$ $\forall i = 1, \ldots, n\ [\![t_i]\!] \subseteq [\![t'_i]\!]$.

The most challenging case is when t_1 is an object type and t_2 is a union type $u_1 \vee u_2$. If $[\![t_1]\!] = \emptyset$, then by lemma 4 we

can apply (split) instead of (∨R1) or (∨R2) (to avoid a non contractive derivation) to the redex $\bot$, and we can conclude by coinductive hypothesis. If $[\![t_1]\!] \neq \emptyset$ and ($[\![t_1]\!] \subseteq [\![u_1]\!]$ or $[\![t_1]\!] \subseteq [\![u_2]\!]$), then we can apply rule[5] (∨R1) or (∨R2) and conclude by coinductive hypothesis. Otherwise, by lemma 3 we know that t_1 contains a union type, therefore we can apply rule (split) by choosing a redex whose depth is minimal, and conclude by lemma 2 and coinductive hypothesis.

The final part of the proof is the most awkward and consists in showing that the derivation obtained in this way is always regular and contractive, and depends on the choices in the proof of step one when more rules are applicable.

4. CONCLUSION

We have shown that the definition of subtyping on coinductive object and union types given in a previous work [3] is not complete even when finite types are considered. To overcome this problem, we have extended our definition of subtyping by introducing a more powerful rule (split) which generalizes the previously devised rules (∨L) and (distr). Such an extension is not trivial since the notion of contractive proof needs to be strengthened to avoid unsoundness. Consequently, the proof of soundness [3] has been generalized.

For what concerns completeness, we have shown that rule (split) does not imply completeness in the more general case when types can be non regular. However, from a more practical point of view, completeness is more interesting when restricting the system to regular types and derivations. In this case we can only conjecture that our definition of subtyping is complete, since although almost finished, there still remain some cases of the proof which need to be checked. Interestingly, if the proof works, then we can easily devise a semicomputable procedure which always returns a contractive and regular derivation 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. We leave open for future work the question whether complete subtyping between regular types is decidable.

5. REFERENCES

[1] O. Agesen. The cartesian product algorithm. In W. Olthoff, editor, *ECOOP'05*, volume 952 of *LNCS*, pages 2–26. Springer, 1995.

[2] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 2–26. Springer, 2009. Best paper prize.

[3] D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. Technical report, DISI, March 2010. Submitted for journal publication.

[4] D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. Technical report, DISI, January 2010. Submitted for journal publication.

[5] 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.

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

[7] 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.

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

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

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[11] 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.

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

[13] N.Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *ECOOP'92*, pages 329–349, 1992.

[14] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA 1991*, pages 146–161, 1991.

[15] 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.

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

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

[18] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01*, volume 2072, pages 99–117. Springer, 2001.

[5] We are deliberately omitting some detail here, since if both inclusions hold, then there are cases where (∨R1) must be preferred over (∨R2), and conversely, depending on the shape of u_1 and u_2.

$t_1 \backslash t_2$	*int*	∨	*obj*
int	(int)	(∨R1) or (∨R2) lemma 1	vacuous
∨	(split) lemma 2	(split) or (∨R1) or (∨R2) lemma 2 and 3	(split) lemma 2
obj	vacuous	(split) or (∨R1) or (∨R2) lemma 2, 3 and 4	(split) or (obj) lemma 4 and 5

Table 1: Applied rules