Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas*

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

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

In recent work we have shown how it is possible to define very precise type systems for object-oriented languages by abstractly compiling a program into a Horn formula f. Then type inference amounts to 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 in the most interesting instantiations both the terms of the coinductive Herbrand universe and goal derivations cannot be finitely represented. However, sound and quite expressive approximations can be implemented by considering only regular terms and derivations. In doing so, it is essential to introduce a proper *subtyping* relation formalizing the notion of approximation between types.

In this paper we study a subtyping relation on coinductive terms built on union and object type constructors. We define an interpretation of types as set of values induced by a quite intuitive relation of membership of values to types, and prove that the definition of subtyping is sound w.r.t. subset inclusion between type interpretations. The proof of soundness has allowed us to simplify the notion of contractive derivation and to discover that the previously given definition of subtyping did not cover all possible representations of the empty type.

1 Introduction

In recent work [4] we have defined a framework which allows precise type analysis of object-oriented programs by means of abstract compilation of the program to be analyzed into a Horn formula (that is, a conjunction of Horn clauses). Then, type inference corresponds to resolving a certain goal (or query) w.r.t. the coinductive (that is, the greatest) Herbrand model of f.

Coinductively defined terms of the Herbrand universe (which correspond to type expressions), in conjunction with the union type constructor, provide an abstract representation for arbitrary sets of values, whereas coinductive SLD resolution [15, 14] allows type inference of recursive method invocation. However, type systems defined in this way are idealized, since, except for the most simple cases where types are just constants, in the most interesting instantiations both terms and goal derivations cannot be finitely represented.

However, sound and quite expressive approximations can be implemented by considering only regular types and derivations, that is, infinite terms and trees, respectively, which can be finitely represented. In doing so, it is essential to introduce a proper *subtyping* relation [2] formalizing the notion of approximation between types, and a corresponding notion of *subsumption* at the level of goal derivation. In this way, regular types, which correspond to usual recursive types, are simply considered as approximations (that is, supertypes) of much finer infinite types which have no finite representation.

This novel approach has several advantages:

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

- It offers a quite general and highly modular framework for type analysis of object-oriented programs, where quite different kinds of analysis can be defined without changing the core inference engine based on coinductive SLD resolution empowered by the notions of subtyping and subsumption. Every instantiation corresponds to a particular choice of the type constructors, the abstract compilation schema, and the definition of the subtyping relation. Our previous papers provide several examples corresponding to different instantiations of the same framework [2, 3]; under this point of view, our proposal is an attempt to provide a common framework for reasoning on type analysis of object-oriented programs. Indeed, the solutions to the problem of type analysis of object-oriented programs which can be found in literature [13, 12, 1, 17, 16, 10] are often rather ad hoc, cannot be easily described in an abstract way, and, for these reasons, cannot be easily compared.
- Several static analysis techniques for compiler optimization can be easily adopted for enhancing type analysis. For instance, we have shown [3] that a more precise type analysis can be obtained when abstract compilation is performed on programs in Static Single Assignment intermediate form [9].
- It promotes a nice integration between theory and practice, since type inference algorithms are just approximations of an idealized type system where its derivable type judgments can be expressed as the limits of chains of approximating judgments derivable by the algorithm, where their precision depends on the space and time resources available to the implementation.

The definition of a suitable subtyping relation is of paramount importance to obtain reasonable approximations of our framework, especially in the presence of union types, which have proved to be quite expressive when coinductive terms are considered.

For this reason, in this paper we study a subtyping relation on coinductive terms built on union and object type constructors. Since types may be infinite, the relation is defined coinductively; however, such a definition is far from being intuitive, because a suitable notion of *contractive* [6, 7] derivation has to be introduced to avoid unsound derivations. The contributions of this paper w.r.t. our previous work are the following:

- We define an interpretation of types as set of values induced by a quite intuitive relation of membership of values to types.
- We prove that the definition of subtyping is sound w.r.t. subset inclusion between type interpretations. The proof of soundness has allowed us to simplify the notion of contractive derivation for subtyping.
- We have discovered that the previously given definition of subtyping did not cover all possible representations of types with an empty interpretation. Consequently, a new subtyping rule has been added, based on a complete characterization of empty types; such a characterization allowed us to define an algorithm for checking empty regular types.

In Section 2 a gentle introduction to the framework is given by means of simple examples. Subtyping and type interpretation are defined in Section 3, whereas Section 4 is devoted to the proof of soundness. Section 5 deals with empty types, and, finally, Section 6 draws some conclusion.

2 Abstract compilation into Horn formulas

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 {
    add(n)  { return n; }
}

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

For simplicity, we just consider method `add`; 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 the instantiation we consider here, types include object types $obj(c,[f_1{:}t_1,\ldots,f_n{:}t_n])$, where c is the class of the object and $f_1,\ldots,f_n$ its fields with their corresponding types $t_1,\ldots,t_n$, union types $t_1 \vee t_2$, and primitive types as *int*. 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 [8, 15, 14]. For instance, the term t s.t. $t = int \vee t$ is regular[2] 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.

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

Type *zer* corresponds to all objects representing zero, while *nat* corresponds to all objects representing natural numbers and, similarly, *pos*, *evn* and *odd* to all objects representing positive, even, and odd natural numbers, respectively. An example of non regular types is given by the infinite sequence $t_1 \vee (t_2 \vee (\ldots \vee t_n \ldots))$, where the term t_i represents the i^{th} prime number.

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,add,[This,N],N).
```

[1] We refer to the author's previous work [4, 2, 3] for more details.

[2] The exact meaning of such a term will be explained in the next section.

```
has_meth(succ,add,[This,N],R) ←
    field_acc(This,pred,P),
    new(succ,[N],S),
    invoke(P,add,[S],R).
```

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)).
```

In this case, since we know[3] that $extends(succ, object)$ and $new(object, [\,], obj(object, [\,]))$ hold, then we can derive $new(succ, [N], obj(succ, [pred : N]))$.

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 $invoke(evn, add, [odd], R)$ is derivable for $R = t$ where t is the regular type s.t. $t = odd \vee t$. If we take for granted that t is equivalent[4] to odd, then not only we can prove that adding an even and an odd number always returns an odd number, but we can also infer the thesis (that is, the result is an odd number), since the query corresponds to just asking which number is returned when adding an even and an odd number.

We recall that, when considering the coinductive Herbrand model, derivations are allowed to be infinite [15]. Then, since $evn = zer \vee obj(succ, [pred{:}obj(succ, [pred{:}evn])])$, by clause 1 for *invoke* we must show that $invoke(zer, add, [odd], odd)$ and $invoke(obj(succ, [pred{:}obj(succ, [pred{:}evn])]), add, [odd], t)$. 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 $invoke(obj(succ, [pred{:}evn]), add, [obj(succ, [pred{:}odd])], t)$. Then, if we re-apply the same clauses once again, we get $invoke(evn, add, [succ^2(odd)], t)$ (where $succ^2(odd)$ is just an abbreviation for $obj(succ, [pred{:}obj(succ, [pred{:}odd])])$) which is equal to the initial goal, except for the argument type which is $succ^2(odd)$ instead of odd. It is now clear that we can get an infinite derivation containing all atoms having shape $invoke(evn, add, [succ^{2n}(odd)], t)$ for all $n \geq 0$, hence $invoke(evn, add, [odd], t)$ is derivable.

There are two main problems with the example of derivation given above: it is not regular, hence it cannot be computed, and we would like to resolve $invoke(evn, add, [odd], R)$ for $R = odd$ rather than for $R = t$. 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 $succ^2(odd) \leq odd$ and $t \leq odd$ should hold.[5] Furthermore, the following subsumption relations are expected to hold: if $succ^2(odd) \leq odd$, then $invoke(evn, add, [odd], t)$

[3] The set of all clauses generated from the two class declarations is available in the Appendix.

[4] The equivalence between the two terms will be clarified in the next section.

[5] More precisely, both directions of the two disequalities hold, since both pairs of terms are equivalent, but here we are only interested in one specific direction.

$$(\text{int})\frac{}{int \leq int} \qquad (\vee\text{R1})\frac{t \leq t_1}{t \leq t_1 \vee t_2} \qquad (\vee\text{R2})\frac{t \leq t_2}{t \leq t_1 \vee t_2} \qquad (\vee\text{L})\frac{t_1 \leq t \quad t_2 \leq t}{t_1 \vee t_2 \leq t}$$

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

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

Figure 1: Rules defining the subtyping relation

subsumes $invoke(evn, add, [succ^2(odd)], t)$, that is, subtyping is contravariant w.r.t. method arguments, as usual, and, therefore, if method *add* returns t when applied to argument *odd*, then it returns t when applied to any subtype of *odd* (in this specific case, $succ^2(odd)$). On the other hand, subtyping is covariant w.r.t. the returned type, therefore if $t \leq odd$ then $invoke(evn, add, [odd], t)$ subsumes $invoke(evn, add, [odd], odd)$, that is, if method *add* returns t when applied to *odd*, then it returns all supertypes of t as well (*odd* in this specific case).

By introducing subtyping and subsumption it is possible to build a regular derivation for $invoke(evn, add, [odd], t)$, by just observing that to prove $invoke(evn, add, [odd], t)$ we need to prove $invoke(evn, add, [succ^2(odd)], t)$ which, in turn, is subsumed by $invoke(evn, add, [odd], t)$, hence we can conclude the proof by coinductive hypothesis. Finally, by applying subsumption once more we can derive $invoke(evn, add, [odd], odd)$ from $invoke(evn, add, [odd], t)$. More in practice, this means that coSLD resolution [15] can be generalized by taking into account subtyping constraints between terms, besides the usual unification constraints.

3 Subtyping and type interpretation

In this section we formally define subtyping as a syntactic relation between types; then we provide an intuitive interpretation of types as sets of values, to define a semantic counterpart of the subtyping relation.

3.1 Definition of subtyping

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

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

An object type $obj(c,[f_1{:}t_1,\ldots,f_n{:}t_n])$ 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 [5, 11].

The subtyping relation is coinductively defined by the rules in Figure 1. Rules are conceived for a purely functional setting [2], an extension for dealing with imperative features can be found in another paper [3] by the same authors.

Rules ($\vee$R1), ($\vee$R2) and ($\vee$L) specify subtyping between union types, and simply state that 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 subtping 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 [3]. Finally, subtyping between object types is allowed only when they refer to the same class name.

Rule (distr) expresses distributivity of object over union types; intuitively, object types correspond to Cartesian product which distributes over union: $A \times (B \cup C) = (A \times B) \cup (A \times C)$. For instance $obj(c,[f{:}t_1 \vee t_2]) \cong obj(c,[f{:}t_1]) \vee obj(c,[f{:}t_2])$, where $u_1 \cong u_2$ holds iff $u_1 \leq u_2$ and $u_2 \leq u_1$. The relation $obj(c,[f{:}t_1]) \vee obj(c,[f{:}t_2]) \leq obj(c,[f{:}t_1 \vee t_2])$ can be derived by applying rules ($\vee$L), (obj), ($\vee$R1) and ($\vee$R2), and by the fact that $t_1 \leq t_1 \vee t_2$ and $t_2 \leq t_1 \vee t_2$ hold by reflexivity, which is ensured by rules (int) and (obj). Rule (distr) is necessary for deriving the opposite direction of the relation, since by applying rules ($\vee$R1), ($\vee$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. Finally, note that rule (distr) is applicable only when the object type on the left-hand side has at least a field associated with a union type; since order of fields is immaterial, in the rule such a field appears always in the first position for readability.

A derivation is a tree where each node is a pair consisting of a judgment of the shape $t_1 \leq t_2$, and the label of a rule[6], and where each node, together with its children, corresponds to a valid instantiation of a rule. For instance, the following tree

$$\begin{array}{ccc} (int \leq int, \text{int}) & & (int \leq int, \text{int}) \\ & \nwarrow \quad \nearrow & \\ & (int \vee int \leq int, \vee\text{L}) & \end{array}$$

is a derivation for $int \vee int \leq int$. However, in the rest of the paper we will use the following equivalent but more intuitive representation for derivations:

$$(\vee\text{L})\dfrac{\overset{(\text{int})}{\overline{int \leq int}} \quad \overset{(\text{int})}{\overline{int \leq int}}}{int \vee int \leq int}$$

Since subtyping is defined over infinite types, 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* [6, 7] (see the definition below). To see why we need such a restriction, consider the regular type u s.t. $u = u \vee u$, and the following infinite derivation containing just applications of rules ($\vee$R1) and ($\vee$R2):

$$\dfrac{\dfrac{\vdots}{int \leq u}}{int \leq u}$$

We reject infinite derivations built applying only rules ($\vee$R1) and ($\vee$R2), since they allow unsound judgments, as $int \leq u$ derived above. As it will be shown in Section 3.2, u corresponds to the empty type, that is, to the bottom element $\bot$ w.r.t. the subtyping relation; indeed, for any type t there exists a contractive derivation for $\bot \leq t$ obtained by applying rule ($\vee$L) infinite times.

[6] This labeling is necessary for the proof of soundness.

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

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

Figure 2: Rules defining membership

Before giving the formal definition of contractive derivation, let us consider another example: if $\bot$ is again the regular type s.t. $\bot = \bot \vee \bot$, then the following infinite derivation, obtained by infinite applications of rule (distr), proves that $obj(c, [f_1{:}\bot, f_2{:}t]) \leq u$ for all u:

$$\frac{\dfrac{\vdots}{obj(c, [f_1{:}\bot, f_2{:}t]) \leq u} \qquad \dfrac{\vdots}{obj(c, [f_1{:}\bot, f_2{:}t]) \leq u}}{obj(c, [f_1{:}\bot, f_2{:}t]) \leq u}$$

Apparently this seems to be an unsound use of rule (distr) as it happens for rules ($\vee$R1) and ($\vee$R2) in the example above; however, this is not the case, as we formally prove in the next section. Since $obj(c, [f_1{:}\bot, f_2{:}t]) \leq u$ and $\bot \leq u$ for all types u, then $\bot \leq obj(c, [f_1{:}\bot, f_2{:}int])$ and $obj(c, [f_1{:}\bot, f_2{:}int]) \leq \bot$ hold, that is, the two types are equivalent and, therefore, both represent the empty type. This result is not so surprising if we interpret the empty type as the empty set of values, and we recall the similarity between records and Cartesian products, and the validity of the equation $\emptyset \times V = \emptyset$.

Def. 3.1 *A derivation for $t_1 \leq t_2$ is contractive iff it contains no sub-derivations built only with rules ($\vee$R1) and ($\vee$R2). The subtyping relation $t_1 \leq t_2$ holds iff there is a contractive derivation for it.*

In the following we use the term *derivation* for contractive ones, unless explicitly specified.

3.2 Interpretation of types

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

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

As 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 2. 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).

An analogous notion of contractive derivation has to be enforced also for membership derivations.

Def. 3.2 *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 $\{v \mid v \in t \text{ holds}\}$.

Before proving the main soundness theorem we show some examples of interpretations.

Example 1 If $\bot$ is the regular type s.t. $\bot = \bot \vee \bot$, then $[\![\bot]\!] = \emptyset$. Indeed, the only applicable rules are ($\vee$L) and ($\vee$R), hence only non contractive derivations can be built.

Example 2 If t is the regular type s.t. $t = int \vee t$, then $[\![t]\!] = [\![int]\!] = \mathbb{Z}$, that is, t and int have the same interpretation. Indeed, all the contractive derivations are obtained by applying n times ($n \geq 0$) rule ($\vee$R) (which is useless in this case), then rule ($\vee$L) followed by (int):

$$\dfrac{\dfrac{\dfrac{}{i \in int}}{i \in int \vee t}}{\vdots}}{i \in int \vee t}$$

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):

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

Let t_2 be the term s.t. $t_2 = obj(zero, [\,]) \vee obj(succ, [pred{:}t_2])$. 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 derivations are finite, hence trivially contractive), whereas the value v_∞ s.t. $v_\infty = obj(succ, [pred \mapsto v_\infty])$ belongs to t_2, but not to t_1. Indeed, the following contractive and regular derivation can be built by alternatively applying rules ($\vee$R) and (obj) infinite times.

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

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

4 Soundness

We now prove that the definition of $\leq$ is sound w.r.t. containment between type interpretations. The proof of soundness is based on the following lemma.

Lemma 4.1 *If t is an object type s.t. $t \leq u$ and $v \in t$, then there exists an object type t' (not necessarily equal to t) s.t. $v \in t'$, and s.t. there exists a derivation for $t' \leq u$ whose first applied rule is ($\vee$R1), ($\vee$R2) or (obj).*

Proof: The proposed proof is constructive, since it shows that the derivation for $t' \leq u$ is just a sub-derivation of the derivation for $t \leq u$, and that the derivation for $v \in t'$ can be easily built from the derivation for $v \in t$.

Let $t = obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])$, by membership rule (obj) $v = obj(c, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n, \ldots])$; furthermore, the corresponding derivation has the following shape:

$$\dfrac{\dfrac{\dfrac{\vdots}{v_1 \in t'_1}}{\dfrac{\vdots\ k_1}{v_1 \in t_1}} \quad \cdots \quad \dfrac{\dfrac{\vdots}{v_n \in t'_n}}{\dfrac{\vdots\ k_n}{v_n \in t_n}}}{v \in obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n])}$$

where $t'_1, \ldots, t'_n$ are not union types, and are obtained after repeatedly applying rules ($\vee$L) or ($\vee$R) $k_1, \ldots, k_n$ times respectively. We know that all k_i are finite, otherwise the derivation would not be contractive. The proof proceeds by induction on $m = \sum_{i \in 1 \ldots n} k_i$.

If $m = 0$, then all $t_1, \ldots, t_n$ are not union types. If $u = int$, then there are no applicable subtyping rules and the claim trivially holds since the hypothesis is not satisfied; if u is either a union or an object type, then the only applicable subtyping rules are ($\vee$R1), ($\vee$R2) or (obj), therefore we easily conclude with $t' = t$. If $m > 0$ and the derivation is obtained by applying rule[7] (distr), then $t_1 = t_a \vee t_b$, that is, $t = obj(c, [f_1{:}t_a \vee t_b, \ldots, f_n{:}t_n])$. Furthermore, in the derivation for $v \in t$, the first applied rule of the sub-derivation for $v_1 \in t_a \vee t_b$ is either ($\vee$L) or ($\vee$R). If ($\vee$L) has been applied (the other case is completely symmetric), then a derivation for $v \in obj(c, [f_1{:}t_a, \ldots, f_n{:}t_n])$ can be obtained from that of $v \in t$, by simply removing the application of rule ($\vee$L) for $v_1 \in t_a \vee t_b$, as depicted in Figure 3. Therefore in such derivation $\sum_{i \in 1 \ldots n} k_i = m - 1$. Finally, since rule (distr) has been applied, we know that $obj(c, [f_1{:}t_a, \ldots, f_n{:}t_n]) \leq u$, hence we can conclude by inductive hypothesis.

As a final remark, note that the construction of t' and of the derivations for $t' \leq u$ and $v \in t'$ are uniquely determined by the derivations for $t \leq u$ and $v \in t$. Therefore, the proof of the lemma shows that there exists a function $\mathscr{F}_L$ s.t. if d_1 and d_2 are derivations for $t \leq u$ and $v \in t$, respectively, with t object type, then $\mathscr{F}_L(d_1, d_2)$ returns (d_3, d_4) s.t. d_3 and d_4 are derivations for $t' \leq u$ and $v \in t'$, respectively, where t' is an object type, d_3 is a sub-derivation of d_1 where the first applied rule is ($\vee$R1), ($\vee$R2) or (obj), and d_4 is obtained by d_2 by replacing some node and removing some applications of rules ($\vee$L) and ($\vee$R). □

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

Proof: The claim can be put in the following equivalent form: for all t_1, t_2, v if $t_1 \leq t_2$, $v \in t_1$ then $v \in t_2$.

The proof is constructive, since it coinductively defines a function $\mathscr{F}$ from derivations for $t_1 \leq t_2$ and $v \in t_1$ to derivations for $v \in t_2$. The definition of $\mathscr{F}$ is given by cases on the first applied subtyping rule of the derivation for $t_1 \leq t_2$.

Rule (int) $\mathscr{F}\left({}^{(\text{int})}\dfrac{}{int \leq int}, {}^{(\text{int})}\dfrac{}{i \in int} \right) = {}^{(\text{int})}\dfrac{}{i \in int}$.

[7] If one between ($\vee$R1), ($\vee$R2), and (obj) has been applied, then the conclusion is straightforward as for $m = 0$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{v_1 \in t_1'}}{\vdots\ {\scriptstyle k_1-1}}}{v_1 \in t_a}\,{\scriptstyle(\vee\mathrm{L})}}{v_1 \in t_a \vee t_b} \quad \cdots \quad \dfrac{\dfrac{\vdots}{v_n \in t_n'}}{\vdots\ {\scriptstyle k_n}} \Big/ v_n \in t_n \;\Big/\; v \in obj(c, [f_1{:}t_a \vee t_b, \ldots, f_n{:}t_n]) \quad \Longrightarrow \quad \dfrac{\dfrac{\vdots}{v_1 \in t_1'}}{\vdots\ {\scriptstyle k_1-1}} \Big/ v_1 \in t_a \quad \cdots \quad \dfrac{\dfrac{\vdots}{v_n \in t_n'}}{\vdots\ {\scriptstyle k_n}} \Big/ v_n \in t_n \;\Big/\; v \in obj(c, [f_1{:}t_a, \ldots, f_n{:}t_n])$$

Figure 3: Transformation of derivations in proof of lemma 4.1

Rule ($\vee$R1) $\mathscr{F}\left(({\scriptstyle\vee\mathrm{R1})}\dfrac{d_1}{t_1 \leq u_1 \vee u_2}, d_2\right) = {\scriptstyle(\vee\mathrm{L})}\dfrac{\mathscr{F}(d_1,d_2)}{v \in u_1 \vee u_2}$, where d_1 is a derivation for $t_1 \leq u_1$, and d_2 is a derivation for $v \in t_1$.

Rule ($\vee$R2) $\mathscr{F}\left(({\scriptstyle\vee\mathrm{R2})}\dfrac{d_1}{t_1 \leq u_1 \vee u_2}, d_2\right) = {\scriptstyle(\vee\mathrm{R})}\dfrac{\mathscr{F}(d_1,d_2)}{v \in u_1 \vee u_2}$, where d_1 is a derivation for $t_1 \leq u_2$, and d_2 is a derivation for $v \in t_1$.

Rule ($\vee$L) There are two sub-cases, depending on the shape of the derivation for $v \in t_2$:

$$\mathscr{F}\left(({\scriptstyle\vee\mathrm{L})}\dfrac{d_1 \quad d_2}{u_1 \vee u_2 \leq t_2},\ {\scriptstyle(\vee\mathrm{L})}\dfrac{d_3}{v \in u_1 \vee u_2}\right) = \mathscr{F}(d_1,d_3)$$

$$\mathscr{F}\left(({\scriptstyle\vee\mathrm{L})}\dfrac{d_1 \quad d_2}{u_1 \vee u_2 \leq t_2},\ {\scriptstyle(\vee\mathrm{R})}\dfrac{d_4}{v \in u_1 \vee u_2}\right) = \mathscr{F}(d_2,d_4)$$

In this case d_1 and d_2 are derivations for $u_1 \leq t_2$ and $u_2 \leq t_2$, respectively, whereas d_3 and d_4 are derivations for $v \in u_1$ and $v \in u_2$, respectively.

Rule (obj)

$$\mathscr{F}\left(\begin{array}{c}({\scriptstyle\mathrm{obj})}\dfrac{d_1,\ldots,d_n}{obj(c,[f_1{:}u_1,\ldots,f_n{:}u_n,\ldots]) \leq obj(c,[f_1{:}u_1',\ldots,f_n{:}u_n'])},\\[2ex] ({\scriptstyle\mathrm{obj})}\dfrac{d_1',\ldots,d_n',\ldots}{obj(c,[f_1 \mapsto v_1,\ldots,f_n \mapsto v_n,\ldots]) \in obj(c,[f_1{:}u_1,\ldots,f_n{:}u_n,\ldots])}\end{array}\right) =$$

$$({\scriptstyle\mathrm{obj})}\dfrac{\mathscr{F}(d_1,d_1'),\ldots,\mathscr{F}(d_n,d_n')}{obj(c,[f_1 \mapsto v_1,\ldots,f_n \mapsto v_n,\ldots]) \in obj(c,[f_1'{:}u_1,\ldots,f_n'{:}u_n])}$$

where $d_1,\ldots,d_n$ are derivations for $u_1 \leq u_1',\ldots,u_n \leq u_n'$, respectively, whereas $d_1',\ldots,d_n'$ are derivations for $v_1 \in u_1,\ldots,v_n \in u_n$, respectively.

The derivation for $obj(c,[f_1 \mapsto v_1,\ldots,f_n \mapsto v_n,\ldots]) \in obj(c,[f_1{:}u_1,\ldots,f_n{:}u_n,\ldots])$ contains ellipses in the right hand side of the sub-derivations $d_1',\ldots,d_n'$ and of the fields of both the value and the type. Their meaning is that there may be other entities in the derivation which, however, can be omitted, since the definition of $\mathscr{F}$ does not depend on them.

Rule (distr) In this case the hypotheses of lemma 4.1 are verified, therefore we can use the function $\mathscr{F}_L$ defined in the proof of the lemma:

$$\mathscr{F}(d_1,d_2) = \mathscr{F}(\mathscr{F}_L(d_1,d_2))$$

where d_1 is a derivation for $t_1 \leq t_2$ whose first applied rule is (distr), hence t_1 is an object type, and d_2 is a derivation for $v \in t_1$. According to the proof of the lemma, $\mathscr{F}_L(d_1,d_2)$ returns (d_3,d_4) s.t. d_3 and d_4 are derivations for $t \leq t_2$ and $v \in t$, t is an object type, and the first applied rule of d_3 is ($\vee$R1), ($\vee$R2), or (obj). Therefore case (distr) is delegated to one of the three cases ($\vee$R1), ($\vee$R2), (obj) specified above.

Now the remaining part of the proof is showing that $\mathscr{F}$ is well-defined. Since $\mathscr{F}$ is defined coinductively, we need to prove that $\mathscr{F}$ is a function, that is, it cannot return two different derivations when applied to the same arguments. To show this, we first prove the following property.

Property (*) If d_1 and d_2 are derivations for $t_1 \leq t_2$ and $v \in t_1$, respectively, and (d_1,d_2) matches cases ($\vee$L) or (distr) of the definition of $\mathscr{F}$, then there always exist d_3 and d_4 s.t. for any derivation d returned by $\mathscr{F}(d_1,d_2)$, the following facts hold: $d = \mathscr{F}(d_3,d_4)$, there exists t s.t. d_3 and d_4 are derivations for $t \leq t_2$ and $v \in t$, respectively, and (d_3,d_4) matches one between (int), ($\vee$R1), ($\vee$R2), and (obj) cases.

Proof of (*): It is immediate to prove that if d_1 and d_2 are derivations for $t_1 \leq t_2$ and $v \in t_1$, respectively, then there always exists one and only one case matching (d_1,d_2) in the definition of $\mathscr{F}$. If (d_1,d_2) matches case (distr), then by lemma 4.1 we know that $\mathscr{F}_L$ is defined on (d_1,d_2), and returns (d_3,d_4) s.t. d_3 and d_4 are derivations for $t \leq t_2$ and $v \in t$, where t is an object type, and the first applied rule of d_3 is ($\vee$R1), ($\vee$R2) or (obj). Now, since (d_1,d_2) cannot match any other case, by definition of $\mathscr{F}$ we can conclude that for any d returned by $\mathscr{F}(d_1,d_2)$, the equality $d = \mathscr{F}(\mathscr{F}_L(d_1,d_2)) = \mathscr{F}(d_3,d_4)$ must hold.

If (d_1,d_2) matches case ($\vee$L), then we proceed by induction on the number n of contiguous applications of membership rules ($\vee$L) and ($\vee$R) with which derivation d_2 starts. We know that such n is finite, otherwise d_2 would not be contractive. The basis if for $n = 1$, since for $n = 0$ the pair (d_1,d_2) would not match case ($\vee$L); for simplicity, let us assume that d_2 starts with the application of rule ($\vee$L), that is, the first sub-case applies (the other sub-case is symmetric). Then we know that d_1 and d_2 have the following shape:

$$d_1 = \ {}_{(\vee\mathrm{L})}\frac{d_3 \quad d_3'}{t \vee t' \leq t_2} \qquad d_2 = \ {}_{(\vee\mathrm{L})}\frac{d_4}{v \in t \vee t'}$$

where d_3 and d_4 are derivations for $t \leq t_2$ and $v \in t$, respectively. Since (d_1,d_2) cannot match any other case, by definition of $\mathscr{F}$ we have that for any d returned by $\mathscr{F}(d_1,d_2)$, the equality $d = \mathscr{F}(d_3,d_4)$ must hold. Finally, (d_3,d_4) must match some case of the definition of $\mathscr{F}$, but such case cannot be ($\vee$L); indeed, $n = 1$ and, therefore, t cannot be a union type. In case (d_3,d_4) matches case (distr), we can apply[8] the result already proved for that case. The inductive step is a direct consequence of the inductive hypothesis and of the fact that if d_2 starts with $n+1$ consecutive applications of rules ($\vee$L) and ($\vee$R), then d_4 starts with n consecutive applications of rules ($\vee$L) and ($\vee$R).

We can now prove the following property.

$\mathscr{F}$ is deterministic: For all d_1,d_2,d,d', if $\mathscr{F}(d_1,d_2) = d$ and $\mathscr{F}(d_1,d_2) = d'$, then $d = d'$.

We prove that $d = d'$ by induction on the height of the finite trees approximating d and d', that is, we show that all paths of d starting from its root are equal to the paths of d' starting from its root, for all

[8] This is possible because proof of case (distr) does not depend on proof of case ($\vee$L).

the lengths[9] of the paths. The basis consists in proving that d and d' have the same root and start with the same rule application (that is, the path length is 0). This comes directly from the definition of $\mathscr{F}$ for the cases (int), ($\vee$R1), ($\vee$R2), and (obj), from the fact that all cases are disjoint, and from property (*) (which deals with the two remaining cases). The inductive step is derived from these same facts, from the inductive hypothesis, and from the standard definition of path length.

$\mathscr{F}$ returns contractive derivations: If d_1 and d_2 are derivations for $t_1 \leq t_2$, $v \in t_1$, respectively, then $\mathscr{F}(d_1,d_2)$ is defined and is a derivation for $v \in t_2$.

First, we recall that the definition of $\mathscr{F}$ covers all possible cases, then $\mathscr{F}$ is always defined on (d_1,d_2). Then we show that the tree returned by $\mathscr{F}$ is always a derivation, and finally we prove that all returned derivations are contractive. To prove that all returned trees are derivations, we first observe that $\mathscr{F}$ always returns a tree having shape $\frac{d}{v \in t_2}$. Again, this comes directly from the definition of $\mathscr{F}$ for the cases (int), ($\vee$R1), ($\vee$R2), and from property (*) (which deals with the two remaining cases). Then the proof proceeds by induction on the height of the finite derivations approximating $\mathscr{F}(d_1,d_2)$. That is, we prove that every node whose distance[10] from the root has length less or equal than n is obtained with a correct rule instantiation, for all n. The basis (for $n = 0$) comes directly from the definition of $\mathscr{F}$ for the cases (int), ($\vee$R1), ($\vee$R2), and from property (*). Let us see case ($\vee$R1) as an example. In this case we know that $\mathscr{F}(d_1,d_2) = (\vee\text{L})\frac{\mathscr{F}(d_3,d_4)}{v \in u_1 \vee u_2}$, where d_3 is a derivation for $t_1 \leq u_1$, and d_3 is a derivation for $v \in t_1$, therefore the root of $\mathscr{F}(d_3,d_4)$ is $v \in u_1$, hence $u_1 \vee u_2$ is obtained with a correct instantiation of rule ($\vee$L). The inductive step is derived from the definition of $\mathscr{F}$ for the cases (int), ($\vee$R1), ($\vee$R2), from property (*), from the inductive hypothesis, and from the standard definition of path length.

We conclude the proof by showing that if d_1 and d_2 are contractive, then $\mathscr{F}(d_1,d_2)$ is contractive as well. By contradiction, let us assume that the returned derivation is not contractive, that is, there exists a sub-derivation containing just applications of memberships rules ($\vee$L) and ($\vee$R). Since ($\vee$R1) and ($\vee$R2) are the only two cases where an application of membership rule ($\vee$L) or ($\vee$R) is added to the returned derivation, and cases ($\vee$L) and (distr) may be defined in terms of cases ($\vee$R1) and ($\vee$R2), then such a sub-derivation can be built by applying only cases ($\vee$R1), ($\vee$R2), ($\vee$L) and (distr) of the definition of $\mathscr{F}$. Now we observe that if case (distr) occurs, then, by definition of $\mathscr{F}_L$ given in lemma 4.1, and by definition of cases ($\vee$R1) and ($\vee$R2), only cases ($\vee$R1) and ($\vee$R2) may occur afterwards; but this means that d_1 contains a sub-derivation built only with rules ($\vee$R1) and ($\vee$R2), that is, d_1 is not contractive, which is in contradiction with the hypothesis. If case (distr) does not occur, and case ($\vee$L) occurs infinite times, then by definition of cases ($\vee$R1), ($\vee$R2), and ($\vee$L), we deduce that d_2 is not contractive, against the hypothesis. The last possibility is when case (distr) does not occur, and case ($\vee$L) occurs only a finite numbers of time; but this necessarily means that at a certain point only cases ($\vee$R1) and ($\vee$R2) may occur, that is, d_1 is not contractive, which is in contradiction with the hypothesis. □

5 A complete characterization of the empty type

We have already shown in Section 3 that $obj(c,[f_1{:}\bot,f_2{:}t]) \leq \bot$, where $\bot$ is the empty type, that is, the type s.t. $\bot = \bot \vee \bot$; therefore, $\bot$ and $obj(c,[f_1{:}\bot,f_2{:}t])$ are equivalent. In fact, besides $obj(c,[f_1{:}\bot,\ldots])$, there are infinitely many other types equivalent to $\bot$, namely, all object types "containing" $\bot$.

For instance, the type $t = obj(c_1,[f{:}obj(c_2,[g{:}\bot])])$ is s.t. $[\![t]\!] = \emptyset$. Unfortunately, $t \leq \bot$ is not derivable from the rules in Figure 1. Indeed, all possible derivations can be built by only applying rules ($\vee$R1)

[9] Recall that the path from the root to a given node is always finite, even when the tree is infinite.

[10] Where the distance is the length of the path from the node to the root.

and ($\vee$R2), and are, therefore, not contractive. To overcome this problem, we introduce a rule explicitly dealing with all types equivalent to the empty type. In order to do that, we would need a predicate $t \downarrow_\perp$ defining all types t equivalent to $\perp$. However, the complementary predicate $t \uparrow_\perp$ turns out to be more convenient, because of its strong similarity with the membership relation; indeed, a type t is not equivalent to the empty type iff there exists a value v s.t. $v \in t$ holds. In this way, it is quite straightforward to prove that the predicate $t \uparrow_\perp$ is sound and complete w.r.t. our type interpretation. Hence, our new subtyping rule is defined as follows.

$$(\text{empty})\frac{}{t_1 \leq t_2}\ t_1 \not\uparrow_\perp$$

The definition of $t \uparrow_\perp$ is quite straightforward.

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

As usual, all derivations have to be contractive, hence they cannot contain sub-derivations obtained by only applying rules ($\uparrow \vee$L) and ($\uparrow \vee$R).

Note that if we restrict ourselves to regular types, then the definition of $\uparrow_\perp$ can be turned into the following algorithm specified in pseudo-Java code.

```
boolean not_empty(type t,stack path) {
  if(t.is_visited())
    return path.is_contractive(t);
  else {
    t.set_visited();
    switch(t) {
        case int: return true;
        case t1∨t2:
          path.push(t);
          if(not_empty(t1,path)) {
            path.pop();
            return true;
          }
          res=not_empty(t2,path);
          path.pop();
          return res;
        case obj(c,[f1:t1,...,fn:tn]):
          path.push(t);
          for i∈1,...,n {
            if (!not_empty(ti,path)) {
              path.pop();
              return false;
            }
          }
          path.pop();
          return true;
      }
  }
}
```

The argument t is the type to be inspected, whereas `path` contains the stack of visited nodes, which must be initially empty. Such a stack is used for checking that the found derivation is contractive. Methods

`is_visited` and `set_visited` are used to keep track of visited terms, which correspond to nodes in a graph. If we end up with an already visited type, then we have an infinite regular path that, however, has to be contractive, otherwise the corresponding derivation is not valid: method `is_contractive` checks whether there is an object type in the sub-path of `path` from t to the top of the stack. The time complexity of the algorithm is linear in the number of edges of the graph representing the term, providing that `is_contractive` has a constant time[11] complexity.

We can now prove that the definition of $\uparrow_\perp$ is sound and complete w.r.t. the interpretation of types.

Theorem 5.1 (Soundness of $t\uparrow_\perp$) *If $t\uparrow_\perp$, then $[\![t]\!] \neq \emptyset$.*

Proof: Similarly to the proof of Theorem 4.1, we coinductively define a function $\mathscr{F}$ mapping derivations for $t\uparrow_\perp$ to derivations for $v \in t$, for a fixed value v:

$$\mathscr{F}\left((\text{int})\frac{}{int\uparrow_\perp}\right) = (\text{int})\frac{}{0 \in int} \qquad \mathscr{F}\left((\vee\text{L})\frac{d}{t_1 \vee t_2\uparrow_\perp}\right) = (\vee\text{L})\frac{\mathscr{F}(d)}{v \in t_1 \vee t_2}$$
$$\mathscr{F}\left((\vee\text{R})\frac{d}{t_1 \vee t_2\uparrow_\perp}\right) = (\vee\text{R})\frac{\mathscr{F}(d)}{v \in t_1 \vee t_2}$$
$$\mathscr{F}\left((\text{obj})\frac{d_1,\ldots,d_n}{obj(c,[f_1{:}t_1,\ldots,f_n{:}t_n])\uparrow_\perp}\right) = (\text{obj})\frac{\mathscr{F}(d_1),\ldots,\mathscr{F}(d_n)}{obj(c,[f_1\mapsto v_1,\ldots,f_n\mapsto v_n]) \in obj(c,[f_1{:}t_1,\ldots,f_n{:}t_n])}$$

Not that $\mathscr{F}$ fully preserves the shape of derivations, in the sense that only the derived judgments change. Using a similar, but simpler, proof scheme as adopted for Theorem 4.1, it is possible to prove that the above definition corresponds to a function $\mathscr{F}$ s.t. for all derivations d for $t\uparrow_\perp$, $\mathscr{F}(d)$ is a derivation for $v \in t$, for a certain v. □

Theorem 5.2 (Completeness of $t\uparrow_\perp$) *If $[\![t]\!] \neq \emptyset$, then $t\uparrow_\perp$.*

Proof: The proof is similar to that for soundness, except that here the function definition is even simpler, since it basically forgets the value v in the membership judgment.

$$\mathscr{F}\left((\text{int})\frac{}{v \in int}\right) = (\text{int})\frac{}{int\uparrow_\perp} \qquad \mathscr{F}\left((\vee\text{L})\frac{d}{v \in t_1 \vee t_2}\right) = (\vee\text{L})\frac{\mathscr{F}(d)}{t_1 \vee t_2\uparrow_\perp}$$
$$\mathscr{F}\left((\vee\text{R})\frac{d}{v \in t_1 \vee t_2}\right) = (\vee\text{R})\frac{\mathscr{F}(d)}{t_1 \vee t_2\uparrow_\perp}$$
$$\mathscr{F}\left((\text{obj})\frac{d_1,\ldots,d_n}{obj(c,[f_1\mapsto v_1,\ldots,f_n\mapsto v_n]) \in obj(c,[f_1{:}t_1,\ldots,f_n{:}t_n])}\right) = (\text{obj})\frac{\mathscr{F}(d_1),\ldots,\mathscr{F}(d_n)}{obj(c,[f_1{:}t_1,\ldots,f_n{:}t_n])\uparrow_\perp}$$

□

This final result allows us to fully reuse the proof of Theorem 4.1 to show that subtyping remains sound w.r.t. containment between type interpretations, if rule (empty) is added.

Corollary 5.1 *The subtyping relation coinductively defined by rules in Figure 1, and by rule (empty) is sound w.r.t. containment between type interpretations.*

[11] This can be achieved by associating a position with each node in the path, and by recording the minimum position p s.t. all paths starting from a node whose position is greater than p are non contractive.

Proof: It suffices considering the same function $\mathscr{F}$ defined in proof of Theorem 4.1, since the new case (empty) cannot occur; indeed, there exist no derivations d_1 and d_2 for $t_1 \leq t_2$ and $v \in t_2$, respectively, s.t. the first applied rule of d_1 is (empty), because, by the side condition of rule (empty), $t_1 \not\downarrow_\perp$, and, hence, by Theorem 5.2, $[\![t_1]\!] = \emptyset$. □

6 Conclusion

We have studied a subtyping relation on coinductive terms built on object and union types constructors, by providing a quite natural interpretation based on a membership relation of values to types, and proved that such a relation is sound w.r.t. containment between type interpretations.

This study has allowed us to improve the original definition of subtyping [2] in two different directions:

- Contractiveness was too restrictive, since no derivations built only with ($\vee$R1), ($\vee$R2), and (distr) rules were allowed, whereas the type interpretation and the corresponding proof of soundness given here have shown that no restrictions on rule (distr) is ever needed. Consequently, the subtyping relation can be implemented more directly, since, rules ($\vee$R1) and ($\vee$R2) have only one premise, in contrast with (distr), and, therefore, checking contractiveness of derivations is simpler.
- The definition did not consider all possible representations of the empty type. Consequently a corresponding new rule has been added, and a sound and complete characterization of all representations of the empty type has been provided; when restricted to regular types, such a characterization directly provides an algorithm for checking whether the interpretation of a type is empty. The time complexity of the algorithm is linear in the number of edges of the graph representing the term.

References

[1] O. Agesen (1995): *The Cartesian Product Algorithm*. In: W. Olthoff, editor: *ECOOP'05 - Object-Oriented Programming*, *Lecture Notes in Computer Science* 952, Springer, pp. 2–26.

[2] D. Ancona & G. Lagorio (2009): *Coinductive type systems for object-oriented languages*. In: S. Drossopoulou, editor: *ECOOP 2009 - Object-Oriented Programming*, *Lecture Notes in Computer Science* 5653, Springer, pp. 2–26.

[3] D. Ancona & G. Lagorio (2010): *Idealized coinductive type systems for imperative object-oriented programs*. Technical Report, DISI. Submitted for journal publication.

[4] D. Ancona, G. Lagorio & E. Zucca (2009): *Type Inference by Coinductive Logic Programming*. In: *Post-Proceedings of TYPES'08*, number 5497 in Lecture Notes in Computer Science, Springer.

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

[6] Michael Brandt & Fritz Henglein (1997): *Coinductive Axiomatization of Recursive Type Equality and Subtyping*. In: *TLCA '97 - Typed Lambda Calculi and Applications*, pp. 63–81.

[7] Michael Brandt & Fritz Henglein (1998): *Coinductive Axiomatization of Recursive Type Equality and Subtyping*. *Fundam. Inform.* 33(4), pp. 309–338.

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

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

[10] M. Furr, J. An, J. S. Foster & M. Hicks (2009): *Static Type Inference for Ruby*. In: *SAC '09: Proceedings of the 2009 ACM symposium on Applied computing*, ACM Press.

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

[12] N.Oxhøj, J. Palsberg & M. I. Schwartzbach (1992): *Making Type Inference Practical*. In: *ECOOP'92 - European Conference on Object-Oriented Programming*, pp. 329–349.

[13] J. Palsberg & M. I. Schwartzbach (1991): *Object-Oriented Type Inference*. In: *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pp. 146–161.

[14] L. Simon, A. Bansal, A. Mallya & G. Gupta (2007): *Co-Logic Programming: Extending Logic Programming with Coinduction*. In: *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, pp. 472–483.

[15] L. Simon, A. Mallya, A. Bansal & G. Gupta (2006): *Coinductive Logic Programming*. In: *Logic Programming, 22nd International Conference, ICLP 2006*, pp. 330–345.

[16] T. Wang & S. Smith (2008): *Polymorphic Constraint-Based Type Inference for Objects*. Technical Report, The Johns Hopkins University. Submitted for publication.

[17] Tiejun Wang & Scott F. Smith (2001): *Precise Constraint-Based Type Inference for Java*. In: *ECOOP'01 - European Conference on Object-Oriented Programming*, 2072, Springer, pp. 99–117.

A Appendix: Horn clauses generated by the code examples in Section 2

The last clauses of `has_field` and `has_meth` are essential for correctly dealing with inherited fields and methods, respectively, even though they could be safely omitted here, since classes `Zero` and `Succ` do not inherit any field or method. Note that we have used negation just for brevity, but it can always be omitted by defining the trivial predicates `not_dec_field` and `not_dec_meth`, since `dec_field` and `dec_meth` are simply defined by a collection of ground facts.

Finally, note that the definition of predicate `field_acc` (for field access) depends on the predicate `rec_acc` (for record access) which is defined by a single clause containing just a singleton record; this is correct thanks to subsumption and subtyping on record types. For instance, since the goal `rec_acc([f1:int],f1,int)` is derivable, and `[f1:int,f2:obj(c,[])]` is a subtype of `[f1:int]`, then `rec_acc([f1:int,f2:obj(c,[])],f1,int)` is derivable as well, by subsumption.

```
class(object).
class(zero).
class(succ).
extends(zero,object).
extends(succ,object).
subclass(X,X) ← class(X).
subclass(X,object) ← class(X).
subclass(X,Y) ← extends(X,Z),subclass(Z,Y).
field_acc(obj(C,R),F,T) ← has_field(C,F),rec_acc(R,F,T).
field_acc(T1∨T2,F,FT1∨FT2) ← field_acc(T1,F,FT1),field_acc(T1,F,FT1).
rec_acc([F:T],F,T).
invoke(obj(C,R),M,A,RT) ← has_meth(C,M,[obj(C,R)|A],RT).
invoke(T1∨T2,M,A,RT1∨RT2) ← invoke(T1,M,A,RT1),invoke(T2,M,A,RT2).
new(object,[],obj(object,[])).
new(zero,[],obj(zero,R)) ← extends(zero,P),new(P,[],obj(P,R)).
new(succ,[N],obj(succ,[pred:N|R])) ← extends(succ,P),new(P,[],obj(P,R)).
dec_field(succ,pred).
```

```
has_field(C,F) ← dec_field(C,F).
has_field(C,F) ← extends(C,P),has_field(P,F),¬dec_field(C,F).
dec_meth(zero,add).
dec_meth(succ,add).
has_meth(zero,add,[This,N],N).
has_meth(succ,add,[This,N],R) ← field_acc(This,pred,P),new(succ,[N],S),
                                invoke(P,add,[S],R).
has_meth(C,M,A,R) ← extends(C,P),has_meth(P,M,A,R),¬dec_meth(C,M).
```