Semantic Subtyping for Imperative Object-Oriented Languages

Davide Ancona Andrea Corradi

DIBRIS - Università di Genova
via Dodecaneso, 35 - 16146 Genova, Italy
davide.ancona@unige.it andrea.corradi@dibris.unige.it

Abstract

Semantic subtyping is an approach for defining sound and complete procedures to decide subtyping for expressive types, including union and intersection types; although it has been exploited especially in functional languages for XML based programming, recently it has been partially investigated in the context of object-oriented languages, and a sound and complete subtyping algorithm has been proposed for record types, but restricted to immutable fields, with union and recursive types interpreted coinductively to support cyclic objects.

In this work we address the problem of studying semantic subtyping for imperative object-oriented languages, where fields can be mutable; in particular, we add read/write field annotations to record types, and, besides union, we consider intersection types as well, while maintaining coinductive interpretation of recursive types. In this way, we get a richer notion of type with a flexible subtyping relation, able to express a variety of type invariants useful for enforcing static guarantees for mutable objects.

The addition of these features radically changes the definition of subtyping, and, hence, the corresponding decision procedure, and surprisingly invalidates some subtyping laws that hold in the functional setting.

We propose an intuitive model where mutable record values contain type information to specify the values that can be correctly stored in fields. Such a model, and the corresponding subtyping rules, require particular care to avoid circularity between coinductive judgments and their negations which, by duality, have to be interpreted inductively.

A sound and complete subtyping algorithm is provided, together with a prototype implementation.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.1 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords Structural Types for Objects, Semantic Subtyping, Read/Write Field Annotations

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2983992

1. Introduction

Subtyping and structural types are essential notions for precise type analysis. They can be employed for typing dynamic languages as JavaScript [7, 19, 20, 24, 25], or to integrate nominal type systems for more flexible and accurate type analysis [16, 18, 23, 27, 30].

In simple type systems the subtyping relation can be defined axiomatically by a set of inference rules; however, when more complex types are considered, the axiomatic approach does not provide a model for formal reasoning, and, thus, may fail to convey the right intuition behind subtyping, and, in general, it is not simple to prove that the subtyping rules are complete.

Semantic subtyping has been proposed as a possible solution to these problems, especially in the context of functional languages for XML based programming, as XDuce [21], and $\mathbb{C}$Duce [17]. Semantic subtyping has been investigated also for the π-calculus [11], for coinductive object types with unions [4], and for ML-like languages with polymorphic variants [14]. More recently, in the context of $\mathbb{C}$Duce, semantic subtyping has been extended with parametric polymorphism [12, 13].

In semantic subtyping types are interpreted as sets of values and the subtyping relation corresponds to set inclusion between type interpretations. In this way, the definition of subtyping is more intuitive, and several properties can be easily deduced (for instance, transitivity always holds trivially). Semantic subtyping naturally supports boolean type constructors; for instance, the interpretation of union and intersection types coincide with set-theoretic union and intersection, respectively. Furthermore, semantic subtyping helps reasoning on recursive types. Let us consider, for instance, the following recursive record type: $\tau = \langle next{:}\tau \rangle$. In the semantic subtyping approach, the syntactic equation above is turned into a semantic equation, which, in general, can be interpreted either inductively or coinductively. In functional languages as $\mathbb{C}$Duce where values are inductive,

types are interpreted inductively, therefore the least solution of the semantic equation $[\![\tau]\!] = [\![\langle next{:}\tau\rangle]\!]$ is considered, and the type τ denotes the empty set.

In object-oriented languages objects are allowed to contain cycles, therefore recursive types are interpreted coinductively [4]; the interpretation of τ corresponds to the greatest solution of $[\![\tau]\!] = [\![\langle next{:}\tau\rangle]\!]$, hence τ denotes a non-empty set of cyclic objects.

Recently, semantic subtyping for record and union types has been studied with recursive types interpreted coinductively, and a practical sound and complete top-down algorithm for deciding it has been provided [4, 9]. However, this result is limited to immutable records, where record subtyping is allowed to be covariant in field types, whereas it is well-known that record subtyping with mutable fields must be invariant to avoid unsoundness. Although invariant subtyping is sound, it severely restricts the flexibility of the type system; to avoid this problem, read/write field annotations [1] can be introduced, to allow subtyping to be covariant for read-only fields, and contravariant for write-only fields.

This paper provides two main contributions.

- A coinductive semantic model is defined for record types with read/write field annotations, supporting union, intersection, and recursive types; as it is customary in the semantic subtyping approach, such a semantic model naturally induces a subtyping relation.

  However, the semantic model used to interpret records with mutable fields substantially deviates from previous models adopted for immutable records [4, 9], and requires more complex definitions and challenging proofs of consistency.

- A set of sound and complete rules for such a semantic subtyping relation is defined, and a corresponding algorithm is devised to decide subtyping. The algorithm has been implemented in SWI Prolog.

  The difference between the semantic model used to interpret mutable records and that adopted for immutable records is reflected in the sets of inequations that can be derived: laws that hold for immutable records [4, 9] are no longer valid for mutable records. Furthermore, in our work new laws have to be introduced because a richer type language is considered: fields are annotated, and intersection types are introduced. Consequently, the algorithm presented here to decide subtyping significantly departs from previously proposed algorithms for immutable records.

The proposed model is language independent, in the sense that it supports reasoning on subtyping in the presence of field annotations, without requiring values of the underlying language to support read/write-only fields. In other words, field annotation is a static notion that must not necessarily be reflected at runtime.

In comparison with semantic subtyping for immutable records, the main challenge consists in field annotation, and record mutability. While the definition of a semantic model in a purely functional setting is rather straightforward, here particular care is required to ensure that the model is well-defined, since fields of record values are associated with types to specify which values can be stored in them.

As a consequence, a circularity is introduced since types are interpreted in terms of themselves, and this needs to be properly managed; in particular, the definitions of the coinductive judgment for typing values, and of its negation (which, by duality, is inductive), are mutually recursive. These issues propagate to the definition of the subtyping rules which involves four mutually recursive judgments: the two coinductive judgments for subtyping, and type emptiness, and their corresponding (inductive) negations.

To our knowledge, no standard approach can be found in literature to properly manage coinductive definitions involving negation, or, equivalently, mutually recursive coinductive and inductive definitions. Our simple but effective solution to break circularity consists in equipping judgments with sets of assumptions that have to be verified, and that, in practice, are abducted by the subtyping algorithm we have implemented.

A previous attempt to investigate semantic subtyping for mutable records can be found in literature [5]; however, the considered model is different, and neither a proof of soundness and completeness is provided, nor an algorithm to decide subtyping is presented.

The paper is organized as follows: Section 2 introduces and motivates semantic subtyping, read/write-only field annotations, and union and intersection types, in the context of mutable records, and, hence, of object-oriented programming. After some preliminary definitions given in Section 3, the novel contributions of this paper span the next three sections: Section 4 tackles the problem of providing a consistent model for types supporting mutable records; Section 5 defines a set of subtyping rules which are sound and complete w.r.t. the model presented in the previous section; Section 6 presents an algorithm driven by the specification provided by the subtyping rules, and shows its implementation in SWI Prolog. Conclusion and directions for future work are discussed in Section 7.

2. Motivating Examples

In this section we provide some examples introducing and motivating the main features of the structural types that will be used throughout the paper: record types with read/write field annotations, union, and intersection types, recursion and coinductive interpretation to deal with circular objects.

2.1 Why Semantic Subtyping for Mutable Records?

A sound and complete procedure to decide semantic subtyping for recursive record and union types has been recently proposed [4, 9]; while this result has paved the way to the investigation of semantic subtyping relations in the context of object-oriented languages by interpreting types coinduc-

tively, its usefulness for static type analysis of object-oriented languages is limited, since the definition of subtyping and the corresponding soundness result strongly rely on the assumption that record fields are immutable.

Such an assumption turns out to be unrealistic for mainstream object-oriented languages where objects are allowed to be mutable.

According to the definition provided in the above mentioned papers, semantic subtyping between record types is covariant in the type of their fields; for instance, semantic subtyping holds for the following pair of types: $\langle f{:}int\rangle \leq \langle f{:}int \vee bool\rangle$; intuitively, this means that if a record has field f of type int, then it has also field f of type int or $bool$. This is sound as long as field f cannot be modified, but if f is in fact the field of a modifiable object, then the judgment $\langle f{:}int\rangle \leq \langle f{:}int \vee bool\rangle$ is no longer sound, as shown by the following example:

```
⟨f:int⟩ o1 = ...;
⟨f:int∨bool⟩ o2 = o1;
o2.f = true;
int i = o1.f;
```

If $\langle f{:}int\rangle \leq \langle f{:}int \vee bool\rangle$ holds, then the assignment `o2=o1` should be considered type safe, however if object assignments are by reference, as usually happens in object-oriented languages, then the line `int i = o1.f;` is unsound.

A simple solution to this problem consists in restricting subtyping between record types to make it invariant in the type of fields: $\langle f{:}\tau\rangle \leq \langle f{:}\tau'\rangle$ iff $\tau \equiv \tau'$, hence $\langle f{:}int\rangle \not\leq \langle f{:}int \vee bool\rangle$.

Even though such a solution is technically sound, it also severely restricts subtyping and, hence, makes static type analysis less precise. The examples that follow in the next subsections show how semantic subtyping allows much more precise static typing of object-oriented languages when record types are considered with read and write field annotations; furthermore, together with union types [4, 9], intersection types are introduced as well. As union types, intersection types naturally arise in the semantic subtyping approach, and allow a further increase of the expressive power of the underlying type system.

2.2 Record Types with Read/Write Annotations

We have shown that record subtyping with mutable fields must be invariant in field types to avoid unsoundness, but this severely restricts the flexibility of the type system; to avoid this problem, two different directions can be followed.

- Covariant subtyping is adopted, despite its unsoundness; this happens in the Java, and C# rule for array type subtyping: $T_1\texttt{[]} \leq T_2\texttt{[]}$ iff $T_1 \leq T_2 \leq$ `Object`. Let us consider the following simplified utility method for copying arrays:

```
static void copyarray(Object[] src, Object[] dst) {
    // omitted checks
    int i = 0;
    for (Object el : src) dst[i++] = el;
}
```

  Thanks to the covariant rule for array type subtyping, this method can work with arrays of any reference type, but the Java type system cannot prevent[1] invocation of `copyarray` to throw `ArrayStoreException` as it would happen for `copyarray(new String[]{ "one", "two"},new Integer[2])`.

- A more flexible, but sound subtyping relation is introduced, based on the idea that covariant/contravariant subtyping is sound when restricted to contexts which limit the operations available on objects. Let us consider the following example involving Java generic types.

```
class Ref<T> {
    T cont;
    Ref(T cont) { super(); this.cont = cont; }
}
static <T> void copyref(Ref<T> src, Ref<T> dst) {
    dst.cont = src.cont;
}
```

  Because subtyping for generic types is invariant, the following method invocations fail to be compiled.

```
Ref<Double> src = new Ref<>(1.0);
Ref<Integer> dst = new Ref<>(1);
Ref<Number> dst2 = new Ref<>(1);
copyref(src,dst); // type error
copyref(src,dst2); // type error
```

  Unfortunately, invariant subtyping is too rigid; for instance, `copyref(src,dst2)` does not compile, although this case causes no harm.

  To overcome this problem one can note that in the body of `copyref`, field `cont` of `src` is read, but not updated, whereas field `cont` of `dst` is updated, but not read. Therefore, in the context of method `copyref`, field `cont` of `src` can be considered read-only, although outside it might be updatable as well, while field `cont` of `dst` can be considered write-only, although outside it might be readable as well. Therefore, it is type safe to consider covariant subtyping for `src`, and contravariant subtyping for `dst`.

  In Java this can be achieved with wildcards [10, 29].

```
static <T>
void copyref2(Ref<? extends T> src, Ref<? super T> dst)
{ dst.cont = src.cont; }
copyref(src,dst); // type error
copyref(src,dst2); // statically correct
```

In Java the wildcard `Ref<? extends T>` supports covariant subtyping, whereas `Ref<? super T>` supports contravariant subtyping, therefore

```
Ref<Double>≤Ref<? extends Double>≤Ref<? extends Number>
Ref<Number>≤Ref<? super Number>≤Ref<? super Double>.
```

A similar example can be reproduced in C#, although more involved, because Java generics support call site variance annotations with wildcards, whereas C# generic interfaces and delegates support declaration site variance annotations [3].

With record types and read/write field annotations [1] the example above can be recast as follows: parameter `src` has

[1] Analogous considerations apply to C#.

type $\langle cont^+{:}T\rangle$, that is, a record with a read-only field `cont` (hence, allowed to be covariant) of type T: field $cont$ can be always accessed, with a result of type T; parameter `dst` has type $\langle cont^-{:}T\rangle$, that is, a record with write-only field `cont` (hence, allowed to be contravariant) of type T: field $cont$ can be always updated with a value of type T. We stress that the fact that `src` has type $\langle cont^+{:}T\rangle$ means that field `cont` of the object denoted by `src` can be read, but not updated inside the method, which is different from assuming that field `cont` of the object denoted by `src` must be constant; it could be updatable, but only outside the method. A dual consideration applies to parameter `dst` as well.

The subtyping rules for read-only and write-only record types are pretty intuitive: $\langle f^+{:}T_1\rangle \leq \langle f^+{:}T_2\rangle$ iff $T_1 \leq T_2$, and $\langle f^-{:}T_1\rangle \leq \langle f^-{:}T_2\rangle$ iff $T_2 \leq T_1$.

2.3 Read/Write Fields and Monotonic Initialization

Besides read-only and write-only annotations, read-write annotations are usually introduced [1] for dealing with fields that must be both readable and writable in a certain context; however, with intersection types this third kind of annotation is redundant. The record type with read-write field f of type T is represented by $\langle f^+{:}T\rangle \wedge \langle f^-{:}T\rangle$: all record where field f can be accessed, with a result of type T, **and** can be updated with a value of type T. The model defined in Section 4 ensures that subtyping is invariant as expected: $\langle f^+{:}T_1\rangle \wedge \langle f^-{:}T_1\rangle \leq \langle f^+{:}T_2\rangle \wedge \langle f^-{:}T_2\rangle$ iff $T_1 \leq T_2$ and $T_2 \leq T_1$, that is, $T_1 \equiv T_2$.

Intersection types, not only make read-write annotations superfluous, but also increase the expressive power of types because the two types assigned to f^+ and f^- need not to be the same; in terms of Java wildcards that would roughly correspond in allowing parameterized types as `Ref<? super Double extends Number>`, not supported by the current Java type system, where both a lower and an upper bound for a wildcard can be declared. By combining field annotations, and intersection and union types, it is possible to enforce *monotonic initialization* [16]: objects monotonically evolve from an uninitialized to a full initialized state; for object values this guarantees that a field, once initialized with a non-null value, never becomes uninitialized again with `null`.

Let $null$ be the singleton type denoting the null value, and T be a non-null type (for instance, any record type is non-null); then the type $\langle f^+{:}null \vee T\rangle \wedge \langle f^-{:}T\rangle$ forces monotonic initialization for field f: access to f may evaluate to null **or** to a value of type T, but only non-null values of type T can be assigned to f. A more detailed example of monotonic initialization in presence of circular objects can be found below.

2.4 Multiple Fields and Recursive Types

With intersection types, record types need not contain multiple fields as it is usually required for types in structural type systems for object-oriented languages; all we need is singleton record types, because a type as $\langle elem^+{:}int, next^+{:}null\rangle$ simply reads as an abbreviation for $\langle elem^+{:}int\rangle \wedge \langle next^+{:}null\rangle$.

We can now consider a more complex example involving recursive types specifying node objects in linked lists. In the Java standard API a unique interface is used for defining three different kinds of lists:

1. *unmodifiable* lists: their elements cannot be replaced nor their size can be modified;
2. *structurally unmodifiable* lists: their elements can be replaced, but their size cannot be modified;
3. *structurally modifiable* lists: their elements can be replaced, and their size can be modified.

Accordingly, the node objects composing an unmodifiable/structurally unmodifiable/modifiable linked list of elements of type T will have the following types, respectively.

1. $\tau_1 = null \vee (\langle elem^+{:}T\rangle \wedge \langle next^+{:}\tau_1\rangle)$;
2. $\tau_2 = null \vee (\langle elem^+{:}T\rangle \wedge \langle elem^-{:}T\rangle \wedge \langle next^+{:}\tau_2\rangle)$;
3. $\tau_3 = null \vee (\langle elem^+{:}T\rangle \wedge \langle elem^-{:}T\rangle \wedge \langle next^+{:}\tau_3\rangle \wedge \langle next^-{:}\tau_3\rangle)$.

All types are recursively defined by means of syntactic equations, hence they correspond to *regular trees* (a.k.a. as rational trees or cyclic terms, see Section 3 for further details); there are two main advantages in managing recursion with regular trees, instead of introducing a μ recursive binder:

- subtyping rules are simpler, since they can be presented at a more abstract level;
- cyclic terms are naturally supported by SWI Prolog, hence, there is also a practical advantage; for instance, the unification[2] `T=rp(f:T)` succeeds, and the logical variable `T` is instantiated with the unique cyclic term satisfying the equation above. As expected, the following SWI Prolog query (where `==` denotes syntactic equality) succeeds: `T1=rp(f:T1),T2=rp(f:rp(f:T2)),T1==T2.`

Let us comment on type τ_1; if linked lists are not circular, and no dummy node is employed, then the first node can be null if the list is empty, or (union type) it can be an object with a read-only field $elem$ of type T, and (intersection type) a read-only field $next$ of type τ_1; both fields are read-only because nodes belong to unmodifiable lists. In τ_2 field $elem$ is read-write, while $next$ is still read-only, whereas in τ_3 both fields are read-write. As expected, in our model $\tau_3 \leq \tau_2 \leq \tau_1$, and $\tau_1 \not\leq \tau_2 \not\leq \tau_3$ (and $\tau_1 \not\leq \tau_3$ as well), capturing the intuition that, for instance, a structurally modifiable list can be safely considered as an unmodifiable list, but not the other way round.

2.5 Coinductive Interpretation of Types

Coinduction is employed at two different levels that must not be confused.

[2] `rp(f:T)` is the Prolog term corresponding to the type $\langle f^+{:}T\rangle$.

At the *syntactic* level, types are regular trees, which include also infinite trees, hence they cannot be defined inductively. Regular trees allow a convenient treatment of recursive types, as previously motivated, and the same approach has been taken for $\mathbb{C}$Duce [17].

At the *semantic* level, types are interpreted as set of values, and when types are recursive their interpretation is recursive as well, and admits both a least and a greatest fixed-point, corresponding to an inductive and coinductive interpretation, respectively.

Let us consider the recursive type specifying nodes of circular linked lists with a dummy node: $\tau = \langle elem^+{:}T\rangle \wedge \langle next^+{:}\tau\rangle$. Because of circularity, and the dummy node, in this case nodes cannot be null, as opposed to the previous examples.

The interpretation of τ, denoted by $[\![\tau]\!]$, has to satisfy the equation $[\![\tau]\!] = [\![\langle elem^+{:}T\rangle \wedge \langle next^+{:}\tau\rangle]\!] = [\![\langle elem^+{:}T\rangle]\!] \cap [\![\langle next^+{:}\tau\rangle]\!]$. The inductive interpretation (least fixed-point) is the empty set, because no well-founded object can have such a type, whereas the coinductive interpretation (greatest fixed-point) is not empty; indeed, it contains non-well-founded objects constituting circular lists.

In our model, $\tau \leq \tau_1$ (with τ_1 as defined in Section 2.4), whereas $\tau_1 \not\leq \tau$; this corresponds to the intuition that a circular linked list can be safely considered as a linked list, but not the other way round. To grasp better such an intuition, let us consider the following code[3] snippet.

```
Node<T> getNode(int index) {
    Node<T> res = this;
    for (int i = 0; i < index; i++)
        res = res.next;
    return res;
}
```

If we assume that `this` (and, hence, variable `res`) has type τ_1, then it is not possible to deduce that the execution of `getNode` cannot throw `NullPointerException`; indeed, `res` could contain null, and, hence, `res.next` could throw `NullPointerException`. However, if we assume that `this` has the more specific type τ, then it is possible to statically guarantee that the method never throws `NullPointerException`.

2.6 Empty Type

With intersection it is possible to define types equivalent to the empty (a.k.a. bottom) type $\mathbf{0}$, whose interpretation is, by definition, the empty set (hence, such types are not inhabited); a simple example is given by $bool \wedge int$, or $\langle f^+{:}bool \wedge int\rangle$, but arbitrary complex types which are not inhabited can be constructed.

As it will be more evident in Section 5, type emptiness is necessary to ensure a sound and complete subtyping decision procedure; indeed, the judgment $\sigma_1 \leq \sigma_2$ holds whenever it is possible to derive that σ_1 is empty, but deciding type emptiness is not trivial. Hence, it is not possible to devise a sound and complete subtyping decision procedure, without a procedure to decide whether a type is empty; as shown in Section 5, checking type emptiness plays a fundamental role also in defining a sound and complete decision procedure for type emptiness, and it is quite challenging, because of the interplay between read-only and write-only fields.

Let us consider the type $\langle f^-{:}bool \vee int\rangle \wedge \langle f^+{:}int\rangle$; this type cannot be inhabited, because, otherwise, the following code would typecheck.

```
⟨f⁻:bool∨int⟩∧⟨f⁺:int⟩ r = ...;
r.f = true;
int i = r.f;
```

The second line of the code above typechecks because, according to type $\langle f^-{:}bool \vee int\rangle$, it is allowed to assign boolean or integer values to field `f` of `r`; the subsequent line typechecks as well, because, according to type $\langle f^+{:}int\rangle$, it is allowed to access field `f` of `r` to get an integer value. Hence, the code must not typecheck because of the declaration of `r`: its type is empty.

More generally, $\langle f^-{:}\tau_1\rangle \wedge \langle f^+{:}\tau_2\rangle \neq \mathbf{0}$ only if $\tau_1 \leq \tau_2$; this dependency between non-empty types and subtyping makes the problem of devising a decision procedure for subtyping more challenging.

The interplay between read-only and write-only fields has also some subtle effect on subtyping. In a functional setting where fields are always immutable, the two types $\langle f{:}\tau_1 \vee \tau_2\rangle$ and $\langle f{:}\tau_1\rangle \vee \langle f{:}\tau_2\rangle$ are equivalent [4]. Surprisingly, in an imperative setting where fields are allowed to be mutable $\langle f^+{:}\tau_1\rangle \vee \langle f^+{:}\tau_2\rangle \leq \langle f^+{:}\tau_1 \vee \tau_2\rangle$ always holds, but $\langle f^+{:}\tau_1 \vee \tau_2\rangle \leq \langle f^+{:}\tau_1\rangle \vee \langle f^+{:}\tau_2\rangle$ holds only if $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$.

For instance, $\langle f^+{:}bool \vee int\rangle \not\leq \langle f^+{:}bool\rangle \vee \langle f^+{:}int\rangle$; indeed, $\langle f^+{:}bool \vee int\rangle \leq \langle f^+{:}bool\rangle \vee \langle f^+{:}int\rangle$ is unsound, because we could deduce from it the following inequalities, where the last one is clearly unsound because $\langle f^+{:}bool \vee int\rangle \not\leq \langle f^+{:}bool\rangle$. Let set $\tau = \langle f^+{:}bool \vee int\rangle \wedge \langle f^-{:}bool\rangle$; then we have the following derivation:

$$\tau \leq (\langle f^+{:}bool\rangle \vee \langle f^+{:}int\rangle) \wedge \langle f^-{:}bool\rangle$$
$$\Leftrightarrow$$
$$\tau \leq (\langle f^+{:}bool\rangle \wedge \langle f^-{:}bool\rangle) \vee (\langle f^+{:}int\rangle \wedge \langle f^-{:}bool\rangle)$$
$$\Leftrightarrow$$
$$\tau \leq (\langle f^+{:}bool\rangle \wedge \langle f^-{:}bool\rangle) \vee \mathbf{0}$$
$$\Leftrightarrow$$
$$\tau \leq \langle f^+{:}bool\rangle \wedge \langle f^-{:}bool\rangle$$

Interestingly enough, while union does not distribute over immutable records, in Section 4 we show that distributivity of intersection over immutable records is sound.

Finally, type emptiness checks can be useful also for warning users, as already happens in the $\mathbb{C}$Duce implementation. While an empty return type could be reasonable for a non terminating function, declaring/inferring an empty type for parameters or local variables is more questionable: parameters and variables that are not allowed to carry values are symptoms of latent bugs; in particular, a function with a pa-

[3] We use Java, but any other object-oriented language would equally work for the purpose.

rameter with an empty type is unusable, because it can never be applied to any value.

3. Technical Background

In this section we overview the main technical notions employed in our formal treatment, and motivate them.

In the rest of the paper, values are modeled by finitely branching trees which are allowed to contain infinite paths, where nodes correspond to value constructors, and the number of children of a node corresponds to its arity. Infinite paths can correspond either to cyclic values (for instance, a record containing itself as a member), or to non-well founded values that can be obtained by a diverging computation.

Analogously, type expressions are modeled by finitely branching trees which are allowed to contain infinite paths, where nodes correspond to type constructors, and the number of children of a node corresponds to its arity; paths can be infinite because types can be recursive. However, since types always correspond to finite sets of definitions, differently from values, type expressions are modeled as *regular trees* (see below).

Trees with infinite paths arise also when considering proof trees of coinductive judgments (see the next subsection).

A formalization of trees with infinite paths and their main properties has been given by Courcelle [15].

3.1 Types and Trees

Recursive types are modeled by infinite but regular trees.

Definition 1. *A regular tree is a possibly infinite tree containing a finite set of subtrees.*

Trees representing type expressions are regular because each (possibly recursive) type is defined by a finite set of definitions (that is, syntactic equations).

The following proposition states a well-known property of regular terms [15].

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

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

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

We define types as all regular trees built on top of the type constructors whose syntax is defined as follows.

$$\begin{array}{lcl} \tau, \rho & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathit{int} \mid \mathit{null} \mid \langle\rangle \mid \langle f^{\nu}{:}\tau\rangle \mid \tau_1 \vee \tau_2 \mid \tau_1 \wedge \tau_2 \\ \nu & ::= & + \mid - \end{array}$$

Since trees are allowed to be infinite (although regular), this is not the standard inductive definition that would generate just finite trees.

The type $\langle\rangle$ represents all record values. The singleton record types $\langle f^{+}{:}\tau\rangle$ and $\langle f^{-}{:}\tau\rangle$ represent all record values containing the read-only, and the write-only field f, respectively.

Union types $\tau_1 \vee \tau_2$ and intersection types $\tau_1 \wedge \tau_2$ [8, 22] correspond to logic disjunction and conjunction, respectively. Types $\mathbf{0}$ and $\mathbf{1}$ are the empty, and the universe (a.k.a. top) type, int represents the set $\mathbb{Z}$, and null denotes the singleton set containing the null reference; the technical treatment can be easily extended to include other primitive types.

Example 1. *Let us consider the type τ_1 introduced in Section 2, such that $\tau_1 = \mathit{null} \vee (\langle \mathit{elem}^{+}{:}T\rangle \wedge \langle \mathit{next}^{+}{:}\tau_1\rangle)$, and let us fix T to be int for simplicity. Then τ_1 is infinite but regular and has only the following six subterms (in infinite regular types a type can be subterm of itself as it happens in this example):*

$$\tau_1 \quad \mathit{null} \quad \langle \mathit{elem}^{+}{:}\mathit{int}\rangle \wedge \langle \mathit{next}^{+}{:}\tau_1\rangle$$

$$\langle \mathit{elem}^{+}{:}\mathit{int}\rangle \quad \langle \mathit{next}^{+}{:}\tau_1\rangle \quad \mathit{int}$$

Let us consider the types τ'_i such that the following equations hold for all natural numbers i.

$$\begin{array}{lcl} \tau'_0 & = & \mathit{null} \\ \tau'_{i+1} & = & \langle \mathit{pred}^{+}{:}\tau'_i\rangle \end{array}$$

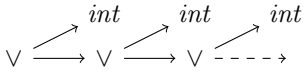
The type $\tau'_0 \vee \tau'_1 \vee \ldots \vee \tau'_n \vee \tau'_{n+1} \ldots$ is infinite, and not regular.

We now introduce the notion of *contractive* type, which allows us to rule out all those types whose intended interpretation is not coinductive.

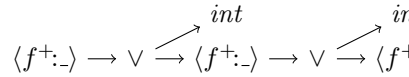
Definition 2. *A type is contractive if all its infinite paths contain a record type constructor.*

The definition above requires all recursive types to be guarded by a record constructor.

Example 2. *The type τ s.t. $\tau = \tau \vee \mathit{int}$ is not contractive, because there exists an infinite path whose nodes are all labeled by the union type constructor.*



The type τ s.t. $\tau = \langle f^{+}{:}\tau \vee \mathit{int}\rangle$ is contractive because all infinite paths have nodes that are alternatively labeled by a record and a union type.



In the non contractive type τ s.t. $\tau = \tau \vee \mathit{int}$ the intended interpretation is the inductive one: $[\![\tau]\!] = \mathbb{Z}$ (the coinductive interpretation is the top type $\mathbf{1}$); in each proof tree showing that a value has type τ the right operand of the union constructor has to be necessarily selected to actually check

something and "consume" the value [6]. Hence, in this case proof trees are required to be finite.

By contrast, in the contractive type τ s.t. $\tau = \langle f^+{:}\tau \vee int\rangle$ the proof tree showing that a value has type τ can be infinite, because each time the rule for record is applied, a check is performed and the value is "consumed". In case the value is not-well-founded (that is, it is a circular object), the rule is applied for an infinite number of times.

Non contractive types do not increase the expressive power of the types, since a non-contractive type can be always turned into an equivalent contractive one [6]. For instance τ s.t. $\tau = \tau \vee int$ is just equivalent to int. Contractive types simplify the formalization, indeed, several definitions and proofs of the claims stated in this paper exploit the assumption that types are contractive; furthermore, contractive types allow optimizations when collecting the set of coinductive hypotheses required to support coinduction in the implementation (Section 6).

For this reason, in the rest of the paper all types are restricted to be regular and contractive.

3.2 Inference Systems and Proof Trees

In the technical sections of the paper several judgments are defined by means of inference systems [2], and, depending on the specific judgment, such inference systems are interpreted either inductively, or coinductively. For all considered inference systems, the associated one step inference operator is monotone (hence, negation is not used in the inference rules), hence the existence of both the least and the greatest fixed point, corresponding to the inductive and coinductive interpretation of the system, respectively, is guaranteed by the Knaster-Tarski theorem.

While the proof trees of inductive judgments have always a finite depth, for coinductive judgments their corresponding proof trees are allowed to contain infinite paths [26].

Since we consider both inductive and coinductive inference systems, we adopt a standard notation [26] for distinguishing inference systems whose intended interpretation is inductive from those having a coinductive intended interpretation: in the latter case, the horizontal lines of the inference rules are thicker.

4. Semantic Subtyping

In the semantic subtyping approach subtyping is not defined in terms of axioms, usually expressed with an inference system, but coincides with set theoretic inclusion between type interpretations:

$$\tau_1 \leq \tau_2 \text{ iff } [\![\tau_1]\!] \subseteq [\![\tau_2]\!].$$

The interpretation $[\![\tau]\!]$ of a type τ is the set of values that have type τ, hence, the definition of $[\![\tau]\!]$ directly depends on the judgment, denoted in this paper by $v \in \tau$, assigning types to values.

$$[\![\tau]\!] = \{v \mid v \in \tau\}.$$

As a consequence, besides types, in the semantic subtyping approach there are other two basic notions that have to be necessarily provided:

- the set of values, and
- the judgment $v \in \tau$ has to be defined.

Since type interpretation does not depend on the notion of term, but uniquely relies on values, semantics subtyping allows a certain degree of language independence; such an independence is reinforced by the fact that an abstract notion of value is employed: values in type interpretations do not coincide with concrete values manipulated by programs. For instance, in our model fields are annotated with type information usually not present in the runtime model of languages. This abstraction allows the same definition of subtyping to be successfully adopted for a family of programming languages sharing some features.

Except for the basic types $null$ and int for which the interpretation is straightforward, the other values are records. In the functional setting, immutable record values are maps from field labels to values, and in the coinductive interpretation they are allowed to be non-well-founded, that is, they can be infinite trees.

For instance, the value uniquely defined by the equation $v = \langle f \mapsto v\rangle$ corresponds to the immutable record with one field f associated with the value itself; that is, v corresponds to a cyclic object.

Mutable record values are more complex, since they have also to carry the information specifying which values can be safely stored in the fields of the record; for instance, the interpretation of type $\langle f^-{:}int \vee null\rangle$ must contain all record values where updating f with either an integer or null is type safe, while $\langle f^-{:}int\rangle$ must contain all record values where updating f with an integer is type safe, and, by contravariance, the type $\langle f^-{:}int \vee null\rangle$ must contain less values than those contained in $\langle f^-{:}int\rangle$: a record value where updating f with an integer is type safe cannot be contained in type $\langle f^-{:}int \vee null\rangle$ which can be safely used also in contexts where f is updated with null.

Therefore the information "field f can be safely updated with values of type τ" must be contained in record values; a simple way to do that is associating field f with type τ, thus introducing a circularity that requires to be properly managed as explained in the rest of this section, since types are interpreted in terms of types.

Values are all finite and infinite trees built on top of the type constructors whose syntax is defined as follows (where $i \in \mathbb{Z}$).

$$\begin{array}{lcl} v & ::= & i \mid \mathrm{null} \mid \langle f_1 \mapsto (\kappa_1, \rho_1), \ldots, f_n \mapsto (\kappa_n, \rho_n)\rangle \\ \kappa & ::= & \emptyset \mid \{v\} \end{array}$$

Record values are finite domain maps from field labels to pairs, hence field names are implicitly assumed to be distinct, and their order is immaterial. The first component of the pair, κ, is a set of cardinality ≤ 1; when empty, it means that its corresponding field is not readable; when it has shape $\{v\}$, then the field is readable, and associated with value v. The second component of the pair is a type that specifies the values that can be stored in the corresponding field. We denote with $\mathbb{R}\mathbb{c}$ the set of all record values.

As an example, the record value

$$\langle f \mapsto (\emptyset, \mathbf{1}), g \mapsto (\{42\}, \mathbf{0}), h \mapsto (\{\mathrm{null}\}, int)\rangle$$

is the record value with three fields f, g, and h; field f is write-only: the empty set $\emptyset$ means that read access for f is forbidden (hence, we do not know the value associated with f), while the top type $\mathbf{1}$ specifies that f can be safely updated with any value; field g is read-only: it is associated with 42, and its value can be read, while the bottom type $\mathbf{0}$ specifies that g can be safely updated with no values; field h is read-write: it is associated with the value null, which is allowed to be read, and can be safely updated with values of type int.

There are three reasons that make the definition of the judgment $v \in \tau$ non trivial.

1. Values can be non-well-founded and types can be infinite regular trees, therefore the judgment has to be defined coinductively; for instance, it should be possible to derive $v \in \tau$ when $v = \langle f \mapsto (\{v\}, \mathbf{0})\rangle$, and $\tau = \langle f^+{:}\tau\rangle$, that is, the read-only cyclic record value v has the recursive type $\tau = \langle f^+{:}\tau\rangle$.
2. If $\tau = \langle f^-{:}\tau'\rangle$, or $\tau = \langle f^+{:}\tau'\rangle$, then the validity of the judgment $v \in \tau$ depends on the validity of subtyping between τ' and the type contained in the record value. For $\tau = \langle f^-{:}\tau'\rangle$ this dependency is rather intuitive; for instance, by contravariance, $\langle f \mapsto (\kappa, \rho)\rangle \in \langle f^-{:}\tau'\rangle$ only if $\tau' \leq \rho$. For $\tau = \langle f^+{:}\tau'\rangle$ the situation is more involved: $\langle f \mapsto (\kappa, \rho)\rangle \in \langle f^+{:}\tau'\rangle$ requires $\kappa = \{v'\}$, and $v' \in \tau'$, but these requirements are not sufficient to guarantee that $\langle f \mapsto (\kappa, \rho)\rangle \in \langle f^+{:}\tau'\rangle$ holds; the condition $\tau' \leq \rho$ is needed as well, otherwise $\langle f^+{:}\tau'\rangle \wedge \langle f^-{:}\rho\rangle$ would be always non empty, even when $\tau' \not\leq \rho$. Indeed, if $v' \in \tau'$, then we would get that both $\langle f \mapsto (\{v'\}, \rho)\rangle \in \langle f^+{:}\tau'\rangle$ and $\langle f \mapsto (\{v'\}, \rho)\rangle \in \langle f^-{:}\rho\rangle$ hold, hence $\langle f \mapsto (\{v'\}, \rho)\rangle \in \langle f^+{:}\tau'\rangle \wedge \langle f^-{:}\rho\rangle$ would hold, and, therefore, $\langle f^+{:}\tau'\rangle \wedge \langle f^-{:}\rho\rangle$ would be non-empty. But assuming that $\langle f^+{:}\tau'\rangle \wedge \langle f^-{:}\rho\rangle$ is non-empty, even when $\tau' \not\leq \rho$, leads to unsoundness. Let us consider, for instance, the following function declaration:

```
int foo(⟨f+:int⟩∧⟨f−:null⟩ x) {
   x.f=null;
   return x.f+1;
}
```

The first line of the body of the function is type safe because `x` has type $\langle f^+{:}int\rangle \wedge \langle f^-{:}null\rangle$, and $\langle f^+{:}int\rangle \wedge \langle f^-{:}null\rangle \leq \langle f^-{:}null\rangle$, while the second line is type safe because `x` has type $\langle f^+{:}int\rangle \wedge \langle f^-{:}null\rangle$, and $\langle f^+{:}int\rangle \wedge \langle f^-{:}null\rangle \leq \langle f^+{:}int\rangle$; but if the type $\langle f^+{:}int\rangle \wedge \langle f^-{:}null\rangle$ is not empty, then it would be possible to call function `foo` with a value in $\langle f^+{:}int\rangle \wedge \langle f^-{:}null\rangle$, but the execution of the body of the function would lead to a type error.
3. There exists a circularity between the definition of $v \in \tau$ and $\tau \leq \tau'$ that has to be properly managed, to show that the two judgments are well-defined.

$$\begin{array}{l}\tau_1 \leq \tau_2 \Leftrightarrow (\forall v.v \in \tau_1 \Rightarrow v \in \tau_2)\\ \langle f \mapsto (\{v\}, \rho)\rangle \in \langle f^+{:}\tau\rangle \Leftrightarrow (v \in \tau, \rho \leq \tau)\\ \langle f \mapsto (\kappa, \rho)\rangle \in \langle f^-{:}\tau\rangle \Leftrightarrow \tau \leq \rho\end{array}$$

Such a circularity can be removed by replacing $\rho \leq \tau$, and $\tau \leq \rho$ on the second and third line with the definition of semantic subtyping given on the first line:

$$\begin{array}{l}\langle f \mapsto (\{v\}, \rho)\rangle \in \langle f^+{:}\tau\rangle \Leftrightarrow (v \in \tau, \forall v'.v' \in \rho \Rightarrow v' \in \tau)\\ \langle f \mapsto (\kappa, \rho)\rangle \in \langle f^-{:}\tau\rangle \Leftrightarrow (\forall v'.v' \in \tau \Rightarrow v' \in \rho)\end{array}$$

In this way we obtain a recursive definition for the judgment $v \in \tau$ which does not depend on the definition of $\tau \leq \tau'$; such a definition has to be interpreted coinductively, since $v \in \tau$ is expected to hold when $v = \langle f \mapsto (\{v\}, \mathbf{0})\rangle$ and $\tau = \langle f^+{:}\tau\rangle$. Unfortunately, the occurrence of $\Rightarrow$ on the right hand side of $\Leftrightarrow$ makes the recursive definition of $v \in \tau$ problematic, because both $v' \in \rho$ and $v' \in \tau$ occur negatively: $v' \in \rho \Rightarrow v' \in \tau$ is equivalent to "not $v' \in \rho$ or $v' \in \tau$", and $v' \in \tau \Rightarrow v' \in \rho$ is equivalent to "not $v' \in \tau$ or $v' \in \rho$". For this reason, the existence of the greatest fixed point is not guaranteed, because the Knaster-Tarski fixed point theorem cannot be applied. To avoid this problem, the negative judgment $v \notin \tau$ is explicitly introduced and defined; in this way, the definitions above can be rewritten as follows:

$$\begin{array}{l}\langle f \mapsto (\{v\}, \rho)\rangle \in \langle f^+{:}\tau\rangle \Leftrightarrow (v \in \tau, \forall v'.v' \notin \rho \text{ or } v' \in \tau)\\ \langle f \mapsto (\kappa, \rho)\rangle \in \langle f^-{:}\tau\rangle \Leftrightarrow (\forall v'.v' \notin \tau \text{ or } v' \in \rho)\end{array}$$

The definitions of the two judgments $v \in \tau$, and $v \notin \tau$ are mutually recursive; for instance, $\langle f \mapsto (\kappa, \rho)\rangle \notin \langle f^-{:}\tau\rangle$ holds if and only if there exists v s.t. $v \in \tau$, and $v \notin \rho$ (that is, $\tau \not\leq \rho$). Unfortunately, such a circularity is problematic because the judgment $v \in \tau$ is interpreted coinductively, whereas, by duality, $v \notin \tau$ requires an inductive definition; for instance, $v \notin \tau$ must not hold when $v = \langle f \mapsto (\{v\}, \mathbf{0})\rangle$, and $\tau = \langle f^+{:}\tau\rangle$. Mutual dependencies between coinductive and inductive judgments is critical [28], since neither the least fixed point, nor the greatest fixed point semantics works properly for mutually recursive definitions involving both coinductive and inductive judgments; to our knowledge, no solution has been proposed in literature for dealing with mutual recursion between inductive and coinductive definitions. To break such a circularity we propose a simple approach which consists in transforming the judgment $v \notin \tau$ into $\Gamma \vdash v \notin \tau$, where Γ is a set of pairs (v, τ), each one corresponding to the hypothesis

$v \in \tau$; accordingly, the meaning of $\Gamma \vdash v \notin \tau$ is as follows: value v does not have type τ, under the assumption that v' has type τ' for all $(v', \tau') \in \Gamma$.

In this way, $v \in \tau$ is defined in terms of both itself, and the judgment $\Gamma \vdash v \notin \tau$, whereas $\Gamma \vdash v \notin \tau$ is defined only in terms of itself; therefore, the definition of the two judgments can be stratified: first, the least fixed point of the recursive definition of $\Gamma \vdash v \notin \tau$ is considered, and its existence is guaranteed by monotonicity, and the Knaster-Tarski theorem. Then, on top of the judgment $\Gamma \vdash v \notin \tau$, the judgment $v \in \tau$ can be defined coinductively; also in this case, the existence of the greatest fixed point is guaranteed by monotonicity, and the Knaster-Tarski theorem.

The complete definitions for the two judgments $v \in \tau$ and $\Gamma \vdash v \notin \tau$ is given in Figure 1 and Figure 2, respectively. We recall that both $v \in \tau$ and $\Gamma \vdash v \notin \tau$ are semantic judgments used to interpret types, and not intended to be computable (indeed, they are not, because values are allowed to be non regular).

Except for the cases on record types, all rules of Figure 1 are straightforward. In rule (rec $\in$) $\langle \ldots \rangle$ denotes any possible record value. The premises of rule (rec$^-$ $\in$) are equivalent to the condition that for all v either v does not have type τ, or v has type ρ (that is, if v has type τ, then it has also type ρ, and, hence, $\tau \leq \rho$); the negative version of the judgment $v \in \tau$ requires a set Γ (which is existentially quantified) of abducted assumptions that have to be verified (hence, $\forall (v_\Gamma, \tau_\Gamma) \in \Gamma . v_\Gamma \in \tau_\Gamma$).

Similar comments apply for rule (rec$^+$ $\in$), but here ρ is expected to be a subtype of τ (recall the explanations given in item 2 on page 8); furthermore, $v \in \tau$ must hold.

Also for the rules in Figure 2 defining $\Gamma \vdash v \notin \tau$, we comment the less trivial cases, that is, when record types are considered.

The judgment $\Gamma \vdash v \notin \langle f^-{:}\tau \rangle$ is derivable in two cases;

- rule (rec$^-$1 $\notin$): v is not a record value, or is a record value with no field f (condition $v \neq \langle f \mapsto \_, \ldots \rangle$);
- rule (rec$^-$2 $\notin$): $v = \langle f \mapsto (\_, \rho), \ldots \rangle$ and $\tau \not\leq \rho$, hence, there exists v' that has type τ (condition $(v', \tau) \in \Gamma$) but does not have type ρ (premise $\Gamma \vdash v' \notin \rho$).

The judgment $\Gamma \vdash v \notin \langle f^+{:}\tau \rangle$ is derivable in three cases;

- rule (rec$^+$1 $\notin$): v is not a record value, or is a record value with no field f (condition $v \neq \langle f \mapsto \_, \ldots \rangle$);
- rule (rec$^+$2 $\notin$): $v = \langle f \mapsto (\_, \rho), \ldots \rangle$ and $\rho \not\leq \tau$, hence, there exists v' that has type ρ (condition $(v', \rho) \in \Gamma$) but does not have type τ (premise $\Gamma \vdash v' \notin \tau$);
- rule (rec$^+$3 $\notin$): $v = \langle f \mapsto (\kappa, \_), \ldots \rangle$, where either $\kappa = \emptyset$, or $\kappa = \{v'\}$, but v' does not have type τ (premise $\Gamma \vdash v' \notin \tau$).

We have arranged the recursive definitions of $v \in \tau$ and $\Gamma \vdash v \notin \tau$ in such a way that we know that there exist the greatest and the least fixed point for the former and the latter judgment, respectively. However, we need to show that one is the negation of the other. This is guaranteed by the following two lemmas.

In the following we denote by OK^Γ the set of all correct assumptions Γ, that is, those containing only derivable pairs:

$$\Gamma \in OK^\Gamma \text{ iff } v \in \tau \text{ is derivable for all } (v, \tau) \in \Gamma.$$

Lemma 1. *If there exists $\Gamma \in OK^\Gamma$ s.t. $\Gamma \vdash v \notin \tau$ is derivable, then $v \in \tau$ is not derivable.*

Proof. The proof is based on a main lemma proved by arithmetic induction and based on the approximation of $\Gamma \vdash v \notin \tau$, and $v \in \tau$. □

Lemma 2. *If for all $\Gamma \in OK^\Gamma$ $\Gamma \vdash v \notin \tau$ is not derivable, then $v \in \tau$ is derivable.*

Proof. The proof is based on a main lemma proved by arithmetic induction and based on the approximation of $\Gamma \vdash v \notin \tau$, and $v \in \tau$. □

4.1 Laws

We conclude this section by showing some of the main laws satisfied by the semantic model, and exploited in the next section for defining the subtyping rules. Most of the proofs of these laws are tacitly based on Lemma 1 and Lemma 2.

As discussed in Section 2 the intersection between read-only and write-only record types, $\langle f^+{:}\tau_1 \rangle \wedge \langle f^-{:}\tau_2 \rangle$, is non-empty only when τ_2 is a subtype of τ_1.

Law 1. $[\![\langle f^+{:}\tau_1 \rangle \wedge \langle f^-{:}\tau_2 \rangle]\!] \neq \emptyset$ *iff* $[\![\tau_2]\!] \subseteq [\![\tau_1]\!]$.

Some (but not all) of the laws that hold in a functional setting where record fields are immutable [4] can be proved for mutable fields as well.

A first example of law that holds also for mutable records concerns empty read-only records.

Law 2. $[\![\langle f^+{:}\tau \rangle]\!] = \emptyset$ *iff* $[\![\tau]\!] = \emptyset$

As opposed to records with read-only fields, records with write-only fields can never be empty.

Law 3. $[\![\langle f^-{:}\tau \rangle]\!] \neq \emptyset$

In particular, the type $\langle f^-{:}\tau \rangle$, where $[\![\tau]\!] = \emptyset$, represents all records having field f.

Law 4. $[\![\langle f^-{:}\tau' \rangle]\!] \subseteq [\![\langle f^-{:}\tau \rangle]\!]$ *and* $[\![\langle f^+{:}\tau' \rangle]\!] \subseteq [\![\langle f^-{:}\tau \rangle]\!]$ *if* $[\![\tau]\!] = \emptyset$

Intersection and record types behave as expected.

Law 5.

1. $[\![\langle f^+{:}\tau_1 \rangle \wedge \langle f^+{:}\tau_2 \rangle]\!] = [\![\langle f^+{:}\tau_1 \wedge \tau_2 \rangle]\!]$
2. $[\![\langle f^-{:}\tau_1 \rangle \wedge \langle f^-{:}\tau_2 \rangle]\!] = [\![\langle f^-{:}\tau_1 \vee \tau_2 \rangle]\!]$

Surprisingly, union types do not behave similarly, as revealed in Section 2.6.

$$\text{(any }\epsilon)\frac{}{v \in \mathbf{1}} \qquad \text{(null }\epsilon)\frac{}{\text{null} \in null} \qquad \text{(int }\epsilon)\frac{}{i \in int} \qquad \text{(rec }\epsilon)\frac{}{\langle \ldots \rangle \in \langle\rangle}$$

$$\text{(and }\epsilon)\frac{v \in \tau_1 \text{ and } v \in \tau_2}{v \in \tau_1 \wedge \tau_2} \qquad \text{(rec}^- \epsilon)\frac{\forall v.(\exists \Gamma.\Gamma \vdash v \notin \tau \text{ and } \forall (v_\Gamma, \tau_\Gamma) \in \Gamma.v_\Gamma \in \tau_\Gamma) \text{ or } v \in \rho}{\langle f \mapsto (\_, \rho), \ldots \rangle \in \langle f^-{:}\tau \rangle}$$

$$\text{(or }\epsilon)\frac{v \in \tau_1 \text{ or } v \in \tau_2}{v \in \tau_1 \vee \tau_2} \qquad \text{(rec}^+ \epsilon)\frac{v \in \tau \text{ and } \forall v'.(\exists \Gamma.\Gamma \vdash v' \notin \rho \text{ and } \forall (v_\Gamma, \tau_\Gamma) \in \Gamma.v_\Gamma \in \tau_\Gamma) \text{ or } v' \in \tau}{\langle f \mapsto (\{v\}, \rho), \ldots \rangle \in \langle f^+{:}\tau \rangle}$$

Figure 1. Definition of $v \in \tau$

$$\text{(empty }\notin)\frac{}{\Gamma \vdash v \notin \mathbf{0}} \qquad \text{(null }\notin)\frac{v \neq \text{null}}{\Gamma \vdash v \notin null} \qquad \text{(or }\notin)\frac{\Gamma \vdash v \notin \tau_1 \text{ and } \Gamma \vdash v \notin \tau_2}{\Gamma \vdash v \notin \tau_1 \vee \tau_2} \qquad \text{(and }\notin)\frac{\Gamma \vdash v \notin \tau_1 \text{ or } \Gamma \vdash v \notin \tau_2}{\Gamma \vdash v \notin \tau_1 \wedge \tau_2}$$

$$\text{(int }\notin)\frac{v \notin \mathbb{Z}}{\Gamma \vdash v \notin int} \qquad \text{(rec }\notin)\frac{v \notin \mathbb{R}\mathbb{c}}{\Gamma \vdash v \notin \langle\rangle} \qquad \text{(rec}^- 1\ \notin)\frac{v \neq \langle f \mapsto \_, \ldots \rangle}{\Gamma \vdash v \notin \langle f^-{:}\tau \rangle} \qquad \text{(rec}^- 2\ \notin)\frac{\exists v.\Gamma \vdash v \notin \rho \text{ and } (v, \tau) \in \Gamma}{\Gamma \vdash \langle f \mapsto (\_, \rho), \ldots \rangle \notin \langle f^-{:}\tau \rangle}$$

$$\text{(rec}^+ 1\ \notin)\frac{v \neq \langle f \mapsto \_, \ldots \rangle}{\Gamma \vdash v \notin \langle f^+{:}\tau \rangle} \qquad \text{(rec}^+ 2\ \notin)\frac{\exists v.\Gamma \vdash v \notin \tau \text{ and } (v, \rho) \in \Gamma}{\Gamma \vdash \langle f \mapsto (\_, \rho), \ldots \rangle \notin \langle f^+{:}\tau \rangle} \qquad \text{(rec}^+ 3\ \notin)\frac{\kappa = \emptyset \text{ or } (\kappa = \{v\} \text{ and } \Gamma \vdash v \notin \tau)}{\Gamma \vdash \langle f \mapsto (\kappa, \_), \ldots \rangle \notin \langle f^+{:}\tau \rangle}$$

Figure 2. Definition of $\Gamma \vdash v \notin \tau$

Law 6.

1. $[\![\langle f^+{:}\tau_1 \rangle \vee \langle f^+{:}\tau_2 \rangle]\!] \subseteq [\![\langle f^+{:}\tau_1 \vee \tau_2 \rangle]\!]$
2. $[\![\langle f^+{:}\tau_1 \vee \tau_2 \rangle]\!] \not\subseteq [\![\langle f^+{:}\tau_1 \rangle \vee \langle f^+{:}\tau_2 \rangle]\!]$
3. $[\![\langle f^-{:}\tau_1 \rangle \vee \langle f^-{:}\tau_2 \rangle]\!] \subseteq [\![\langle f^-{:}\tau_1 \wedge \tau_2 \rangle]\!]$
4. $[\![\langle f^-{:}\tau_1 \wedge \tau_2 \rangle]\!] \not\subseteq [\![\langle f^-{:}\tau_1 \rangle \vee \langle f^-{:}\tau_2 \rangle]\!]$

We have already shown in Section 2.6 why $\langle f^+{:}\tau_1 \vee \tau_2 \rangle \leq \langle f^+{:}\tau_1 \rangle \vee \langle f^+{:}\tau_2 \rangle$ is unsound; an analogous example shows why $\langle f^-{:}\tau_1 \wedge \tau_2 \rangle \leq \langle f^-{:}\tau_1 \rangle \vee \langle f^-{:}\tau_2 \rangle$ is unsound. If $\tau_1 = null \vee bool$ and $\tau_2 = null \vee int$, then $\tau_1 \wedge \tau_2 = null$, and the following inequalities can be deduced.

$$\begin{aligned}
&\langle f^-{:}null \rangle \wedge \langle f^+{:}null \vee int \vee string \rangle \leq \\
&\quad (\langle f^-{:}null \vee bool \rangle \vee \langle f^-{:}null \vee int \rangle) \wedge \\
&\quad \langle f^+{:}null \vee int \vee string \rangle \\
&\langle f^-{:}null \rangle \wedge \langle f^+{:}null \vee int \vee string \rangle \leq \\
&\quad \langle f^-{:}null \vee int \rangle \wedge \langle f^+{:}null \vee int \vee string \rangle
\end{aligned}$$

The last deduced inequality is unsound, because field f of the rhs type can be updated with integer values, but not field f of the lhs type.

Finally, the following law concerns the relationship between read-only and write-only records with the same field.

Law 7. *For all* τ_1, τ_2, $[\![\langle f^+{:}\tau_1 \rangle]\!] \subseteq [\![\langle f^-{:}\tau_2 \rangle]\!]$ *iff* $[\![\tau_1]\!] = \emptyset$ *or* $[\![\tau_2]\!] = \emptyset$.

5. Subtyping

In this section we show how subtyping can be defined by a system of subtyping rules which are sound and complete w.r.t. the semantic subtyping induced by the model defined in the previous section. Such a system drives the Prolog implementation detailed in Section 6.

5.1 Or- and And-sets

To simplify the subtyping rules and make the subtyping algorithm more effective, besides considering only regular and contractive types (we recall the comments in Section 3), union and intersection types are generalized to allow union and intersection over finite non-empty sets of types.

$$\begin{array}{lll}
\pi ::= & \mathbf{0} \mid \mathbf{1} \mid int \mid null \mid \langle\rangle \mid \langle f^\nu{:}\pi \rangle & \\
& \mid \vee\{\pi_1, \ldots, \pi_n\} \mid \wedge\{\pi_1, \ldots, \pi_n\} & \text{(n>0)} \\
\sigma ::= & \vee\{\varsigma_1, \ldots, \varsigma_n\} \mid \varsigma & \text{(n>0)} \\
\varsigma ::= & \wedge\{\iota_1, \ldots, \iota_n\} \mid \iota & \text{(n>0)} \\
\iota ::= & \mathbf{0} \mid \mathbf{1} \mid int \mid null \mid \langle\rangle \mid \langle f^\nu{:}\pi \rangle &
\end{array}$$

Figure 3. Types with or- and and-sets

The type $\vee\{\pi_1, \ldots, \pi_n\}$ (called or-set) represents the union of the types $\pi_1, \ldots, \pi_n$ and $\wedge\{\pi_1, \ldots, \pi_n\}$ (called and-set) represents the intersection of the types $\pi_1, \ldots, \pi_n$.

The meta-variables ι, ς, and σ range over subsets of types that will be used in the subtyping rules. In particular, in a σ type each type which is not guarded by a record type is guaranteed to be in disjunctive normal form (see the next subsection).

For the subtyping rules we assume that types adhere to the more abstract syntax defined above, types expressed with the syntax defined in Section 3 (that is, where binary union and intersection constructors are employed instead of or- and and-sets) can be easily transformed to match the new syntax.

The judgment $\tau \rightsquigarrow \pi$, coinductively defined in Figure 4, is used to transform all binary boolean operators into the corresponding and-/or-sets (where repetitions are removed).

For instance $int \vee null \vee int \rightsquigarrow \vee\{int, null\}$, and $(int \vee int) \wedge (int \vee int) \rightsquigarrow \wedge\{\vee\{int\}\}$; clearly, $\vee\{\pi\}$ and $\wedge\{\pi\}$ are both equivalent to π, but for keeping definitions simpler we avoid this further simplification of singleton or-sets and and-sets.

$$\frac{\tau \in \{null, int, \langle\rangle, \mathbf{0}, \mathbf{1}\}}{\tau \rightsquigarrow \tau} \qquad \frac{\tau \rightsquigarrow \pi}{\langle f^{\nu}{:}\tau\rangle \rightsquigarrow \langle f^{\nu}{:}\pi\rangle}$$

$$\frac{\tau_1 \rightsquigarrow \pi_1 \quad \tau_2 \rightsquigarrow \pi_2}{\tau_1 \vee \tau_2 \rightsquigarrow \pi_1 \veebar \pi_2} \qquad \frac{\tau_1 \rightsquigarrow \pi_1 \quad \tau_2 \rightsquigarrow \pi_2}{\tau_1 \wedge \tau_2 \rightsquigarrow \pi_1 \wedgebar \pi_2}$$

Figure 4. Definition of $\tau \rightsquigarrow \pi$

The definition uses two auxiliary operators: $\veebar$ and $\wedgebar$, their definitions are given in Figure 5 in terms of the parametric operator $\barwedge$, where α can be instantiated with either $\vee$ or $\wedge$.

$$\pi_1 \barwedge \pi_2 = \alpha S$$

$$\text{with } S = \begin{cases} S_1 \cup S_2 & \text{if } \pi_1 = \alpha S_1 \text{ and } \pi_2 = \alpha S_2 \\ \{\pi_1\} \cup S_2 & \text{if } \pi_1 \neq \alpha\{\ldots\} \text{ and } \pi_2 = \alpha S_2 \\ S_1 \cup \{\pi_2\} & \text{if } \pi_1 = \alpha S_1 \text{ and } \pi_2 \neq \alpha\{\ldots\} \\ \{\pi_1\} \cup \{\pi_2\} & \text{if } \pi_1 \neq \alpha\{\ldots\} \text{ and } \pi_2 \neq \alpha\{\ldots\} \end{cases}$$

$$\mathop{\Big\lfloor\!\alpha\!\Big\rfloor}_{i=1}^{n>1} \pi_i = \pi_1 \barwedge \ldots \barwedge \pi_{n-1} \barwedge \pi_n \qquad \mathop{\Big\lfloor\!\alpha\!\Big\rfloor}_{i=1}^{1} \pi_i = \pi_1$$

Figure 5. Definition of auxiliary operators

The operator $\pi_1 \veebar \pi_2$ ($\pi_1 \wedgebar \pi_2$ respectively) returns a flattened or-set (and-set) that represents the union (intersection) of π_1 and π_2, and, therefore, enjoys all the property of set-theoretic union (intersection).

For simplicity, in the examples we keep the syntax defined in Section 3.

5.2 Type Normalization

Besides employing or- and and-sets, we assume that types are initially normalized (see function $norm$ defined below), that is, they are put in disjunctive normal form (DNF), and simplifications are applied to record types (see function $simp$ defined in Figure 8).

Since type normalization does not enter record types, it is performed lazily, therefore, while we assume that types are initially normalized before checking subtyping (hence, function $norm$ is tacitly applied), function $norm$ is also explicitly applied to types in the subtyping rules whenever the type of a record field is accessed.

Function dnf, defined in Figure 7 along with $dstr$, puts a type in DNF; it is the identity function when restricted to basic types ι. Function $dstr$ takes as input an and-set of types that are already in DNF, and returns an equivalent type in DNF by applying the distribution property of intersection over union.

For instance: $dstr(\wedge\{\vee\{\langle f^{+}{:}int\rangle, int\}, null\}) = \vee\{\wedge\{\langle f^{+}{:}int\rangle, null\}, \wedge\{int, null\}\}$

Function $simp$ is applied to types σ in DNF to simplify record types inside or/and-sets types according to Law 5 proved in Section 4: $[\![\langle f^{+}{:}\pi_1\rangle \wedge \langle f^{+}{:}\pi_2\rangle]\!] = [\![\langle f^{+}{:}\pi_1 \wedge \pi_2\rangle]\!]$, and $[\![\langle f^{-}{:}\pi_1\rangle \wedge \langle f^{-}{:}\pi_2\rangle]\!] = [\![\langle f^{-}{:}\pi_1 \vee \pi_2\rangle]\!]$.

Function $norm$ corresponds to the composition of the two functions $simp$, and dnf:

$$norm(\pi) = simp(dnf(\pi)).$$

Lemma 3. $[\![\pi]\!] = [\![norm(\pi)]\!]$.

Proof. The dnf and $simp$ functions apply the distributive law and Law 5, respectively. □

5.3 Subtyping Rules

We define a subtyping judgment with a system of subtyping rules which are sound and complete w.r.t. the definition of semantic subtyping.

According to Law 7, $[\![\langle f^{+}{:}\pi_1\rangle]\!] \subseteq [\![\langle f^{-}{:}\pi_2\rangle]\!]$ iff $[\![\pi_1]\!] = \emptyset$ or $[\![\pi_2]\!] = \emptyset$, therefore the definition of the subtyping judgment depends on the definition of the judgment for checking whether types are empty. These two judgments cannot be merged because subtyping is coinductively defined, whereas emptiness is inductive. By Law 1, $[\![\langle f^{+}{:}\pi_1\rangle \wedge \langle f^{-}{:}\pi_2\rangle]\!] = \emptyset$ iff $[\![\pi_2]\!] \not\subseteq [\![\pi_1]\!]$, therefore the judgment for type emptiness is defined in terms of the negation of the subtyping judgment, which, in turn, is defined in terms of the judgment for non emptiness of types, because $[\![\pi]\!] \neq \emptyset$ iff $[\![\pi]\!] \not\subseteq \emptyset$; again, these two judgments cannot be merged, since non-emptiness is coinductive, whereas negation of subtyping is inductive. Finally, again by Law 1, the definition of the non-emptiness judgment depends on the definition of the subtyping judgment.

The dependency graph between these four judgments is depicted in Figure 9.

As happens in Section 4 for the definition of the coinductive judgment $v \in \tau$, and its negation, there is a circular definition involving coinductive and inductive judgments that has to be broken; the dashed edge in the picture corresponds to the dependency which is removed as similarly done in Section 4: the judgments for emptiness, non-emptiness and negation of subtyping use a set Π of abducted pairs (π_1, π_2) corresponding to the assumptions that $norm(\pi_1) \leq norm(\pi_2)$ holds. In this way, there is no circularity involving coinductive and inductive judgments, the coinductive judgment for non-emptiness is defined only in terms of itself, and is used by the judgment for negation of subtyping, which is used by the inductive judgment for emptiness, and, finally, the coinductive judgment for subtyping depends on the judgment for emptiness. For these reasons, the definitions of the judgments are stratified and fixed points are computed as follows: first, the greatest fixed point of the recursive definition of non-emptiness; then, the least fixed point of emptiness, and the

$$dstr(\wedge\{\pi,\pi_1,\ldots,\pi_n\})=\begin{cases}\bigsqcup_{i=1}^{n}\bigsqcup_{j=1}^{h}\pi'_i\sqcup\pi''_j \\ \qquad \text{if } dstr(\wedge\{\pi_1,\ldots,\pi_n\})=\vee\{\pi''_1,\ldots,\pi''_h\} \\ \bigsqcup_{i=1}^{n}\pi'_i\sqcup dstr(\wedge\{\pi_1,\ldots,\pi_n\}) \text{ otherwise}\end{cases}$$
$$\text{if } \pi=\vee\{\pi'_1,\ldots,\pi'_n\}$$
$$dstr(\wedge\{\pi_1,\ldots,\pi_n\})=\bigsqcap_{i=1}^{n}\pi_i \text{ if } \pi_i\neq\vee\{\ldots\}\ \forall i\in[1,n]$$

Figure 6. Definition of auxiliary operator $dstr$

$$dnf(\vee\{\pi_1,\ldots,\pi_n\})=\bigsqcup_{i=1}^{n}dnf(\pi_i)$$
$$dnf(\wedge\{\pi_1,\ldots,\pi_n\})=dstr(\bigsqcap_{i=1}^{n}dnf(\pi_i))$$
$$dnf(\iota)=\iota$$

Figure 7. Definition of dnf

$$simp(\vee\{\varsigma_1,\ldots,\varsigma_n\})=\bigsqcup_{i=1}^{n}simp(\varsigma_i)$$

$$simp(\wedge\{\langle f^+{:}\pi_1\rangle,\ldots,\langle f^+{:}\pi_n\rangle,\iota'_1,\ldots,\iota'_m\})=$$
$$\langle f^+{:}\bigsqcap_{i=1}^{n}\pi_i\rangle\sqcup simp(\wedge\{\iota'_1,\ldots,\iota'_m\})$$
$$\text{if } n\geq 2 \text{ and } \iota'_i\neq\langle f^+{:}\_\rangle\forall i\in[1,m]$$

$$simp(\wedge\{\langle f^-{:}\pi_1\rangle,\ldots,\langle f^-{:}\pi_n\rangle,\iota'_1,\ldots,\iota'_m\})=$$
$$\langle f^-{:}\bigsqcap_{i=1}^{n}\pi_i\rangle\sqcup simp(\wedge\{\iota'_1,\ldots,\iota'_m\})$$
$$\text{if } n\geq 2 \text{ and } \iota'_i\neq\langle f^-{:}\_\rangle\forall i\in[1,m]$$

$$simp(\sigma)=\sigma \text{ otherwise}$$

Figure 8. Simplification of types

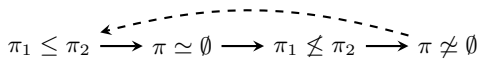


Figure 9. Dependency graph between the four judgments

least fixed point of negation of subtyping; finally, the greatest fixed point of the recursive definition of subtyping.

Subtyping has to be interpreted coinductively because if $\tau_1=\langle f^+{:}null\rangle\vee\langle f^+{:}\tau_1\rangle$, and $\tau_2=\langle f^+{:}null\vee\tau_2\rangle$, then $\tau_1\leq\tau_2$ must hold; therefore, with a syntax directed definition of the subtyping judgment the derivation for $\tau_1\leq\tau_2$ must contain $\tau_1\leq\tau_2$ itself in the descendant nodes of the derivation tree. By duality, negation of subtyping has to be interpreted inductively.

Non emptiness has to be interpreted coinductively because if $\tau=\langle f^+{:}\tau\rangle$, then τ must be non-empty; therefore, with a syntax directed definition of the non emptiness judgment the derivation for $\emptyset\vdash\tau\not\simeq\emptyset$ must contain $\emptyset\vdash\tau\not\simeq\emptyset$ itself in the descendant nodes of the derivation tree. By duality, emptiness has to be interpreted inductively.

All rules in this section are deliberately non algorithmic in order to provide a high level specification of the subtyping algorithm: in particular, several rules overlap, and some of them have premises with existential quantification. In Section 6 we show how these rules can be made algorithmic and effectively implemented.

The non emptiness judgment $\Pi\vdash\sigma\not\simeq\emptyset$ is defined in Figure 10.

Rule ($\vee\not\simeq\emptyset$) states that a union type is not empty if at least one of its operands is not empty.

Rules (any $\not\simeq\emptyset$), (simple $\not\simeq\emptyset$), and (rec $\not\simeq\emptyset$) state that the types $\mathbf{1}$, int, $null$, $\langle\rangle$ are non empty, as expected.

Rule (rec-r $\not\simeq\emptyset$) is driven by Law 2 which states that $[\![\langle f^+{:}\tau\rangle]\!]=\emptyset$ iff $[\![\tau]\!]=\emptyset$; since normalization does not propagate inside record types, the type of field f has to be normalized with function $norm$.

Rule (rec-w $\not\simeq\emptyset$) is driven by Law 3 which states that $[\![\langle f^-{:}\tau\rangle]\!]\neq\emptyset$.

All the remaining rules deal with intersection types.

Rule ($\wedge$single$\not\simeq\emptyset$) corresponds to the base case consisting of singleton and-sets.

Rule ($\wedge$any$\not\simeq\emptyset$) deals with the simple case when an element of the and-set is $\mathbf{1}$; the judgment $\Pi\vdash\wedge\{\mathbf{1}\}\not\simeq\emptyset$ can be correctly derived thanks to rule ($\wedge$single$\not\simeq\emptyset$).

Rule ($\wedge$rec $\not\simeq\emptyset$) deals with the simple case when an element of the and-set is $\langle\rangle$; all other elements in the and-set must not be primitive types; the judgment $\Pi\vdash\wedge\{\langle\rangle\}\not\simeq\emptyset$ can be correctly derived thanks to rule ($\wedge$single$\not\simeq\emptyset$).

Rule (r $\wedge$ w $\not\simeq\emptyset$) is the only one that requires the use of Π; if both $\langle f^+{:}\pi\rangle$ and $\langle f^-{:}\pi'\rangle$ are contained in the same and-set then by Law 1, their intersection is not empty only if $\pi'\leq\pi$. To break circularity the rule requires the assumption $\pi'\leq\pi$ with the side condition $(\pi',\pi)\in\Pi$. Furthermore, the intersections of the record types with the rest of the and-set must be non empty.

Rules (r$\wedge\not\simeq\emptyset$) and (w$\wedge\not\simeq\emptyset$) deal with the simpler cases where the and-set does not contain two record types with the same field, and different annotations (recall that type normalization exploits function $simp$ which merges together record types in and-sets with the same field and annotation); this covers also the case where there are record types with different fields, since they cannot interfere. Rules (r$\wedge\not\simeq\emptyset$) requires also the premise $\Pi\vdash norm(\pi)\not\simeq\emptyset$ (as for rule (rec-r $\not\simeq\emptyset$), the type of the field has to be normalized with function $norm$, since normalization does not propagate inside record types) because, by Law 2, the type of the field of a non empty covariant record type must be non empty.

The inductive rules for checking emptiness of a type are defined in Figure 11.

$$(\vee\not\simeq\emptyset)\ \frac{\Pi \vdash \varsigma \not\simeq \emptyset}{\Pi \vdash \vee\{\ldots,\varsigma,\ldots\} \not\simeq \emptyset} \qquad (\text{any}\not\simeq\emptyset)\ \frac{}{\Pi \vdash \mathbf{1} \not\simeq \emptyset} \qquad (\text{simple}\not\simeq\emptyset)\ \frac{\iota \in \{int, null, \langle\rangle\}}{\Pi \vdash \iota \not\simeq \emptyset}$$

$$(\text{rec-r}\not\simeq\emptyset)\ \frac{\Pi \vdash norm(\pi) \not\simeq \emptyset}{\Pi \vdash \langle f^{+}{:}\pi\rangle \not\simeq \emptyset} \qquad (\text{rec-w}\not\simeq\emptyset)\ \frac{}{\Pi \vdash \langle f^{-}{:}\_\rangle \not\simeq \emptyset} \qquad (\wedge\text{any}\not\simeq\emptyset)\ \frac{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}{\Pi \vdash \wedge\{\mathbf{1},\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\wedge\text{single}\not\simeq\emptyset)\ \frac{\Pi \vdash \iota \not\simeq \emptyset}{\Pi \vdash \wedge\{\iota\} \not\simeq \emptyset} \qquad (\wedge\text{rec}\not\simeq\emptyset)\ \frac{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \text{ and } \forall i \in [1,n]\ \iota_i \notin \{null, int\}}{\Pi \vdash \wedge\{\langle\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\text{r}\wedge\text{w}\not\simeq\emptyset)\ \frac{\Pi \vdash \wedge\{\langle f^{+}{:}\pi\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \quad \Pi \vdash \wedge\{\langle f^{-}{:}\pi'\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \quad (\pi',\pi) \in \Pi}{\Pi \vdash \wedge\{\langle f^{+}{:}\pi\rangle,\langle f^{-}{:}\pi'\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\wedge\text{r}\not\simeq\emptyset)\ \frac{\Pi \vdash norm(\pi) \not\simeq \emptyset \text{ and } \Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \text{ and } \forall i \in [1,n]\ \iota_i \notin \{null, int, \langle f^{-}{:}\_\rangle\}}{\Pi \vdash \wedge\{\langle f^{+}{:}\pi\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\wedge\text{w}\not\simeq\emptyset)\ \frac{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \text{ and } \forall i \in [1,n]\ \iota_i \notin \{null, int, \langle f^{+}{:}\_\rangle\}}{\Pi \vdash \wedge\{\langle f^{-}{:}\_\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

Figure 10. Non-emptiness of types σ

$$(\vee\simeq\emptyset)\ \frac{\Pi \vdash \varsigma_i \simeq \emptyset\ \forall i \in [1,n]}{\Pi \vdash \vee\{\varsigma_1,\ldots,\varsigma_n\} \simeq \emptyset} \qquad (\wedge\simeq\emptyset)\ \frac{\Pi \vdash \iota \simeq \emptyset}{\Pi \vdash \wedge\{\ldots,\iota,\ldots\} \simeq \emptyset} \qquad (\text{rec-r}\simeq\emptyset)\ \frac{\Pi \vdash norm(\pi) \simeq \emptyset}{\Pi \vdash \langle f^{+}{:}\pi\rangle \simeq \emptyset}$$

$$(\text{empty}\simeq\emptyset)\ \frac{}{\Pi \vdash \mathbf{0} \simeq \emptyset} \qquad (\wedge\text{prim}\simeq\emptyset)\ \frac{\iota \in \{int, null\} \quad \iota' \neq \mathbf{1}}{\Pi \vdash \wedge\{\ldots,\iota,\iota',\ldots\} \simeq \emptyset} \qquad (\text{r}\wedge\text{w}\simeq\emptyset)\ \frac{\Pi \vdash norm(\pi') \not\leq norm(\pi)}{\Pi \vdash \wedge\{\ldots,\langle f^{+}{:}\pi\rangle,\langle f^{-}{:}\pi'\rangle,\ldots\} \simeq \emptyset}$$

Figure 11. Emptiness of types σ

As depicted in Figure 9, the judgment $\Pi \vdash \sigma \simeq \emptyset$ is defined in terms of the judgment $\Pi \vdash \sigma_1 \not\leq \sigma_2$, which is interpreted inductively as well.

Rule $(\vee \simeq \emptyset)$ states that an or-set is empty if all of its elements are empty.

Rule $(\wedge \simeq \emptyset)$ states that an and-set is empty if at least one of its elements is empty.

Rule (rec-r$\simeq\emptyset$) is based on Law 2: a covariant record type is empty if the type of its field is empty; as in other cases, type normalization has to be propagated to the type of the field by applying function $norm$.

Rule (empty$\simeq\emptyset$) is immediate.

Rule $(\wedge \text{prim} \simeq \emptyset)$ states that an and-set is empty if it contains at least two different types ι and ι' when one is either $null$ or int, and the other is not $\mathbf{1}$.

Finally, rule $(\text{r} \wedge \text{w} \simeq \emptyset)$ is the only one which requires the use of the judgment $\Pi \vdash \sigma_1 \not\leq \sigma_2$; by Law 1, an and-set containing both $\langle f^{+}{:}\pi\rangle$ and $\langle f^{-}{:}\pi'\rangle$ is empty if π' is not a subtype of π.

The inductive rules defining the judgment $\Pi \vdash \sigma_1 \not\leq \sigma_2$ are defined in Figure 12.

Rules (any$\not\leq$) and (simple$\not\leq$) deal with the cases involving two basic types in $\{null, int, \langle\rangle, \mathbf{1}\}$.

Rules (l-or$\not\leq$) and (r-or$\not\leq$) deal with or-sets; the former corresponds to the set-theoretic property $A \cup B \subseteq C \Rightarrow A \subseteq C$, that is, $A \not\subseteq C \Rightarrow A \cup B \not\subseteq C$, while the latter corresponds to a property which is peculiar of this type system, and does not hold in general for sets.

Similarly, rules (l-and$\not\leq$) and (r-and$\not\leq$) deal with and-sets; the former corresponds to a property which is peculiar of this type system, and does not hold in general for sets, while the latter corresponds to the set-theoretic property $A \subseteq B \cap C \Rightarrow A \subseteq B$, that is, $A \not\subseteq B \Rightarrow A \not\subseteq B \cap C$. In particular, rule (l-and$\not\leq$) requires the additional premise $\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset$, otherwise it would be possible to derive invalid judgments, as $\Pi \vdash \wedge\{null, int\} \not\leq \langle\rangle$.

Rule $(empty \not\leq)$ uses the judgment for non emptiness, and deals with cases when the type on the right-hand side is empty, but the type on the left-hand side is not; in this way it is possible to cover situations that are not considered by other rules; for instance, rule $(empty \not\leq)$ is required to derive $\Pi \vdash int \not\leq \vee\{\wedge\{\mathbf{0}, int\}, \mathbf{0}\}$.

Rule (r\w$\not\leq$) is driven by Law 7 stating that $[\![\langle f^{+}{:}\tau_1\rangle]\!] \subseteq [\![\langle f^{-}{:}\tau_2\rangle]\!]$ iff $[\![\tau_1]\!] = \emptyset$ (and hence, by Law 2, $[\![\langle f^{+}{:}\tau_1\rangle]\!] = \emptyset$) or $[\![\tau_2]\!] = \emptyset$, while rule (w\r$\not\leq$) states that $\langle f^{-}{:}\sigma_1\rangle$ can never be a subtype of $\langle f^{+}{:}\sigma_2\rangle$ for any σ_1, σ_2; indeed,

$$\text{(any}\not\leq\text{)}\frac{\iota \neq \mathbf{1}}{\Pi \vdash \mathbf{1} \not\leq \iota} \quad \text{(empty}\not\leq\text{)}\frac{\Pi \vdash \varsigma \not\simeq \emptyset}{\Pi \vdash \varsigma \not\leq \mathbf{0}} \quad \text{(l-or}\not\leq\text{)}\frac{\exists i \in [1,n]\ \Pi \vdash \varsigma_i \not\leq \sigma}{\Pi \vdash \vee\{\varsigma_1, \ldots, \varsigma_n\} \not\leq \sigma} \quad \text{(r-or}\not\leq\text{)}\frac{\forall i \in [1,n]\ \Pi \vdash \varsigma \not\leq \varsigma_i}{\Pi \vdash \varsigma \not\leq \vee\{\varsigma_1, \ldots, \varsigma_n\}}$$

$$\text{(l-and}\not\leq\text{)}\frac{\forall i \in [1,n]\ \Pi \vdash \iota_i \not\leq \iota \quad \Pi \vdash \wedge\{\iota_1, \ldots, \iota_n\} \not\simeq \emptyset}{\Pi \vdash \wedge\{\iota_1, \ldots, \iota_n\} \not\leq \iota} \quad \text{(r-and}\not\leq\text{)}\frac{\exists i \in [1,n]\ \Pi \vdash \varsigma \not\leq \iota_i}{\Pi \vdash \varsigma \not\leq \wedge\{\iota_1, \ldots, \iota_n\}}$$

$$\text{(r\w}\not\leq\text{)}\frac{\Pi \vdash \langle f^+{:}\pi\rangle \not\simeq \emptyset \quad \Pi \vdash norm(\pi') \not\simeq \emptyset}{\Pi \vdash \langle f^+{:}\pi\rangle \not\leq \langle f^-{:}\pi'\rangle} \quad \text{(w\r}\not\leq\text{)}\frac{}{\Pi \vdash \langle f^-{:}\_\rangle \not\leq \langle f^+{:}\_\rangle} \quad \text{(rec}\not\leq\text{)}\frac{f_1 \neq f_2}{\Pi \vdash \langle {f_1}^{\nu_1}{:}\_\rangle \not\leq \langle {f_2}^{\nu_2}{:}\_\rangle}$$

$$\text{(simple}\not\leq\text{)}\frac{\iota \in \{null, int, \langle\rangle\} \quad \iota' \notin \{\iota, \mathbf{1}\}}{\Pi \vdash \iota \not\leq \iota'} \quad \text{(w\w}\not\leq\text{)}\frac{\Pi \vdash norm(\pi') \not\leq norm(\pi)}{\Pi \vdash \langle f^-{:}\pi\rangle \not\leq \langle f^-{:}\pi'\rangle} \quad \text{(r\r}\not\leq\text{)}\frac{\Pi \vdash norm(\pi) \not\leq norm(\pi')}{\Pi \vdash \langle f^+{:}\pi\rangle \not\leq \langle f^+{:}\pi'\rangle}$$

Figure 12. Rules for negation of subtyping

$\langle f \mapsto (\emptyset, \sigma_1)\rangle \in [\![\langle f^-{:}\sigma_1\rangle]\!]$, but $\langle f \mapsto (\emptyset, \sigma_1)\rangle$ never belongs to $\langle f^+{:}\sigma_2\rangle$.

Rule (rec$\not\leq$) states that two record types with different fields can never be in subtyping relation, independently of the variance annotations.

Finally, (w\w$\not\leq$) and (r\r$\not\leq$) are the counterparts of the standard rules for contravariant and covariant record subtyping, respectively; as in analogous cases, type normalization has to be propagated to the types of the fields by applying function $norm$.

The coinductive rules defining the subtyping relation can be found in Figure 13.

Subtyping is the only judgment that does not depend on a set of assumptions; we recall that the set Π of subtyping assumptions is required for breaking the cyclic dependency between the non emptiness and the subtyping judgment. Of course, such assumptions need to be verified by the subtyping judgment. In Section 6 the existential quantification over subtyping assumptions in the premises of the subtyping rules will be removed, and the set of assumptions will be abducted by the implemented algorithm. In the typing rules, which are expressed in a purely declarative way, Π can be considered as an input to the judgments, whereas in the implementation it is actually an output.

Rule (prim$\leq$) imposes reflexivity between the primitive types $null$ and int.

Rule (empty$\leq$) states that an empty type σ_1 is always in subtyping relation with any type; however, all subtyping assumptions in Π required to derive that σ_1 is empty need to be verified. Furthermore, types in Π have to be normalized.

Rules (l-or$\leq$) and (r-or$\leq$) deal with or-sets, whereas (r-and$\leq$) and (l-and$\leq$) deal with and-sets. While rules (l-or$\leq$) and (r-and$\leq$) are standard and can be correctly read in both directions (hence, set-theoretically, they are sound and complete), in set theory rules (r-or$\leq$) and (l-and$\leq$) are sound but not complete; nevertheless, they are also complete in this type system.

Rule (any$\leq$) states that type $\mathbf{1}$ is a supertype of any type.

Rule (r\w$\leq$) is driven by Law 7 which states that $[\![\langle f^+{:}\tau_1\rangle]\!] \subseteq [\![\langle f^-{:}\tau_2\rangle]\!]$ iff $[\![\tau_1]\!] = \emptyset$ or $[\![\tau_2]\!] = \emptyset$. As for rule (empty$\leq$), all subtyping assumptions in Π required to derive that π is empty need to be verified, and types in Π have to be normalized. The case when the type on the left-hand side is empty is covered by rule (empty$\leq$).

Rules (r\r$\leq$) and (w\w$\leq$) are the standard ones for covariant and contravariant subtyping, respectively.

Finally, rule ($\langle\rangle \leq$) states that type $\langle\rangle$ is the supertype of all record types.

Before stating and proving the main results on the defined judgments, we provide an example of derivation tree for the subtyping judgment $\tau_3 \leq \tau_1$ where τ_3 and τ_1 correspond to the normalization of types in the example in Section 2.4 on unmodifiable/modifiable linked lists; for simplicity, we assume that the type T of the elements of the lists is int.

$$\begin{array}{l}\tau_1 = \vee\{null, \wedge\{S_1\}\}\\ \tau_3 = \vee\{null, \wedge\{S_2\}\}\\ S_1 = \{\langle elem^+{:}int\rangle, \langle next^+{:}\tau_1\rangle\}\\ S_2 = S_1 \cup \{\langle elem^-{:}int\rangle, \langle next^-{:}\tau_3\rangle\}\end{array}$$

The infinite, but regular, proof tree for $\tau_3 \leq \tau_1$ is the following one

$$\text{(l-or}\leq\text{)}\frac{\text{(r-or}\leq\text{)}\dfrac{\text{(prim}\leq\text{)}\dfrac{}{null \leq null}}{null \leq \tau_1} \quad \text{(r-or}\leq\text{)}\dfrac{\vdots}{\wedge\{S_2\} \leq \wedge\{S_1\}}}{\wedge\{S_2\} \leq \tau_1}}{\tau_3 \leq \tau_1}$$

where the proof tree for $\wedge\{S_2\} \leq \wedge\{S_1\}$ is as follows

$$\text{(r-and}\leq\text{)}\frac{\dfrac{\vdots}{\wedge\{S_2\} \leq \langle elem^+{:}int\rangle} \quad \dfrac{\vdots}{\wedge\{S_2\} \leq \langle next^+{:}\tau_1\rangle}}{\wedge\{S_2\} \leq \wedge\{S_1\}}$$

$$\text{(prim}\leq)\ \frac{\iota \in \{int, null\}}{\iota \leq \iota} \qquad \text{(empty}\leq)\ \frac{\exists \Pi.\Pi \vdash \sigma_1 \simeq \emptyset \quad \forall (\pi_1, \pi_2) \in \Pi. norm(\pi_1) \leq norm(\pi_2)}{\sigma_1 \leq \sigma_2} \qquad \text{(any}\leq)\ \frac{}{\sigma \leq \mathbf{1}}$$

$$\text{(l-or}\leq)\ \frac{\forall i \in [1,n]\ \varsigma_i \leq \sigma}{\vee\{\varsigma_1, \ldots, \varsigma_n\} \leq \sigma} \qquad \text{(r-or}\leq)\ \frac{\exists i \in [1,n]\ \varsigma \leq \varsigma_i}{\varsigma \leq \vee\{\varsigma_1, \ldots, \varsigma_n\}} \qquad \text{(r-and}\leq)\ \frac{\forall i \in [1,n]\ \varsigma \leq \iota_i}{\varsigma \leq \wedge\{\iota_1, \ldots, \iota_n\}} \qquad \text{(l-and}\leq)\ \frac{\exists i \in [1,n]\ \iota_i \leq \iota}{\wedge\{\iota_1, \ldots, \iota_n\} \leq \iota}$$

$$\text{(r\textbackslash w}\leq)\ \frac{\exists \Pi.\Pi \vdash norm(\pi) \simeq \emptyset \quad \forall (\pi_1, \pi_2) \in \Pi. norm(\pi_1) \leq norm(\pi_2)}{\langle f^+{:}\_\rangle \leq \langle f^-{:}\pi\rangle}$$

$$\text{(w\textbackslash w}\leq)\ \frac{norm(\pi') \leq norm(\pi)}{\langle f^-{:}\pi\rangle \leq \langle f^-{:}\pi'\rangle} \qquad \text{(r\textbackslash r}\leq)\ \frac{norm(\pi) \leq norm(\pi')}{\langle f^+{:}\pi\rangle \leq \langle f^+{:}\pi'\rangle} \qquad \text{(}\langle\rangle\leq)\ \frac{\iota \in \{\langle f^+{:}\_\rangle, \langle f^-{:}\_\rangle, \langle\rangle\}}{\iota \leq \langle\rangle}$$

Figure 13. Subtyping rules

and the proof trees for $\wedge\{S_2\} \leq \langle elem^+{:}int\rangle$ and $\wedge\{S_2\} \leq \langle next^+{:}\tau_1\rangle$ are

$$\text{(l-and}\leq)\ \frac{\text{(r\textbackslash r}\leq)\ \frac{\text{(prim}\leq)\ \overline{int \leq int}}{\langle elem^+{:}int\rangle \leq \langle elem^+{:}int\rangle}}{\wedge\{S_2\} \leq \langle elem^+{:}int\rangle}$$

$$\text{(l-and}\leq)\ \frac{\text{(r\textbackslash r}\leq)\ \frac{\vdots}{\tau_3 \leq \tau_1}}{\langle next^+{:}\tau_3\rangle \leq \langle next^+{:}\tau_1\rangle}}{\wedge\{S_2\} \leq \langle next^+{:}\tau_1\rangle}$$

The vertical dots in the premises of $\tau_3 \leq \tau_1$ mean that the tree is infinite (although regular) and the derivation continues as already specified above.

5.4 Main Results

We provide the main claims stating that the judgments are well-defined and the subtyping judgment is sound and complete w.r.t. semantic subtyping. All proofs are available in an extended version[4] of this paper.

Lemma 4. *If $\Pi \in OK^\Pi$ and $\Pi \vdash \sigma \not\simeq \emptyset$ is derivable, then $\Pi \vdash \sigma \simeq \emptyset$ is not derivable; if there exists $\Pi \in OK^\Pi$ s.t. $\Pi \vdash \sigma_1 \not\leq \sigma_2$ is derivable, then $\sigma_1 \leq \sigma_2$ is not derivable.*

Lemma 5. *If $\Pi \in OK^\Pi$ and $\Pi \vdash \sigma \not\simeq \emptyset$ is not derivable, then $\Pi \vdash \sigma \simeq \emptyset$ is derivable; if for all $\Pi \in OK^\Pi$ $\Pi \vdash \sigma_1 \not\leq \sigma_2$ is not derivable, then $\sigma_1 \leq \sigma_2$ is derivable.*

The proofs of the two lemmas above follow the same technique adopted for Lemma 1 and 2 in Section 4.

The next two theorems state soundness and completeness of the subtyping rules. All types are assumed to be in normal form.

Theorem 1 (Soundness)**.** *If $\pi_1 \leq \pi_2$, then $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$.*

[4] Downloadable at `http://www.disi.unige.it/person/AnconaD/papers/AnconaCorradiOOPSLA2016extended.pdf`.

Proof. The proof exploits the approximations of the four judgments. □

Theorem 2 (Completeness)**.** *If $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$, then $\pi_1 \leq \pi_2$.*

Proof. By Theorem 1, Lemma 4 and Lemma 5, and by contraposition. □

6. Algorithm and Implementation

From the inference rules defined in Section 5 it is not possible to directly derive an algorithm for deciding semantic subtyping between coinductive types. In particular, coinductive judgments, and abduction of sets Π of subtyping assumptions have to be implemented.

Once that Lemma 4 and 5 prove that the judgment defining $\not\leq$ is actually the complement of $\leq$, negation can be exploited in the pseudo-code defining the algorithm, hence subtyping depends on emptiness which in turns depends on the negation of subtyping, therefore only the two predicates `subtype` and `is_empty` need to be defined: the former is coinductive and depends on the latter which is inductive and depends on the negation of the former. To break circularity, the `is_empty` predicate abducts sets $\bar{\Pi}$ of pairs (π_1, π_2) corresponding to the assumptions $norm(\pi_1) \not\leq norm(\pi_2)$, as opposed to what happens in the inference rules in Section 5, where judgments refer to sets Π of positive hypotheses $norm(\pi_1) \leq norm(\pi_2)$.

Pseudo-code for predicate `is_empty` is defined in Listing 1; to be closer to the real prototype implementation in SWI Prolog, pseudo-code is expressed with high-level Horn clauses where some details have been abstracted away; furthermore, some auxiliary predicates have been omitted, and their behavior is only specified informally.

Atom `is_empty(`σ`, `$\bar{\Pi}$`)` succeeds if σ (a simplified type in DNF) is empty under the assumptions in set $\bar{\Pi}$; hence, σ and $\bar{\Pi}$ are treated as input and output, respectively. The predicate is defined in terms of another predicate with the same name, but four arguments:

- input σ: the simplified type in DNF that has to be checked;
- input Ψ: the set of coinductive hypotheses, corresponding to the types on which emptiness has been already checked; this is essential to guarantee termination in presence of recursive types (that is, cyclic Prolog terms); since types are contractive, only non basic record types (that is, $\langle f^+{:}ext\rangle$ or $\langle f^-{:}ext\rangle$, but not $\langle\rangle$) need to be added to Ψ, since, by contractivity, an infinite path in a recursive type must necessarily involve a record type. The same consideration applies for the `subtype` predicate defined below.
- input $\bar{\Pi}'$: the initial set of abducted assumptions; indeed, this argument is used to accumulate abducted assumptions;
- output $\bar{\Pi}$: the final set of abducted assumptions, returned if σ can be empty; the invariant $\bar{\Pi}' \subseteq \bar{\Pi}$ always holds, that is, the returned final set $\bar{\Pi}$ of abducted hypotheses is always a super-set of the initial set $\bar{\Pi}'$ of abducted hypotheses.

Except for the first straightforward rule dealing with the empty type $\mathbf{0}$, all other rules are applicable only if $\sigma \notin \Psi$, to ensure termination if a recursive type is processed more than once. In this case the predicate has to fail; for instance, this happens when $\sigma = \langle f^+{:}\sigma\rangle$.

The two rules dealing with union types are intuitive, whereas the rule for intersection types is more involved. The auxiliary predicate `partition_by_field`, whose definition has been omitted, partitions the set $\wedge\{\iota_1, \ldots, \iota_n\}$ in two collections, `RecsMap` and `Others`. The former is a map containing all basic types of shape $\langle f^\nu{:}\pi\rangle$, indexed by their fields; since types are normalized, each field f can be mapped at most to the two types $\langle f^+{:}\_\rangle$ and $\langle f^-{:}\_\rangle$. The latter is a set containing all other basic types $\mathbf{0}$, $\mathbf{1}$, int, $null$, or $\langle\rangle$.

The first part of the body of the clause covers rule ($\wedge\simeq\emptyset$), instantiated with $\iota = \mathbf{0}$, and rule ($\wedge$prim$\simeq\emptyset$).

The remaining part (`get_keys(RecsMap, Fs), ...`) covers rule ($\wedge\simeq\emptyset$), instantiated with $\iota = \langle f^+{:}\pi\rangle$, and rule (r$\wedge$w$\simeq\emptyset$). The auxiliary predicate `get_keys`, whose definition has been omitted, returns all fields which are mapped to some record type. Predicate `is_empty_and` tries to apply to some existing field either rule ($\wedge\simeq\emptyset$), instantiated with $\iota = \langle f^+{:}\pi\rangle$, or rule (r$\wedge$w$\simeq\emptyset$). This latter rule is applicable only when $\pi_2 \not\leq \pi_1$, hence the pair (π_2, π_1) is added to the set of abducted assumptions. The auxiliary predicate `lookup`, whose definition has been omitted, returns the set of records associated with a specific field in the map.

Pseudo-code for predicate `subtype` is defined in Listing 2; the main predicate is defined in terms of an auxiliary predicate with the same name but one more argument.

In `subtype(`Ψ`, `σ_1`, `σ_2`)` all arguments are considered as input; `subtype(`Ψ`, `σ_1`, `σ_2`)` succeeds if σ_1 is a subtype of σ_2 (where both types are simplified and in DNF), under the set Ψ of coinductive hypotheses, corresponding to the pairs of types on which subtyping has been already checked; again, this is essential to guarantee termination in presence of recursive types. As opposed to what happens for the inductive predicate `is_empty`, the coinductive predicate `subtype` succeeds if the pair consisting of the two types σ_1, and σ_2 belongs to Ψ.

The clauses defining `subtype(`Ψ`, `σ_1`, `σ_2`)` are very similar to the rules defined in Figure 13, except for some cases. A clause has been added to ensure termination in case of recursive types, for dealing with coinduction: if $(\sigma_1, \sigma_2) \in \Psi$ holds, then the predicate succeeds by virtue of the coinductive interpretation. Iteration over the elements of or-sets and and-sets in rules(l-or$\leq$) and (r-and$\leq$), respectively, is implemented through recursion, hence, two clauses have added to deal with the base cases. Finally, the clauses corresponding to rules (empty$\leq$) and (r\w$\leq$) use negation, because, as already explained, this allows an implementation based on the definition of just two predicates, instead of four.

6.1 Prototype Implementation

We have developed a prototype implementation in Prolog; besides allowing rapid prototyping and conciseness, Prolog has the advantage of offering native support for cyclic terms, unification, and backtracking. In particular, backtracking is needed to properly deal with abducted assumptions. Consider for instance the type

$$\sigma = \vee\{\wedge\{\langle f^+{:}\mathbf{1}\rangle, \langle f^-{:}null\rangle\}, \wedge\{\langle f^+{:}null\rangle, \langle f^-{:}\mathbf{1}\rangle\}\}$$

whose semantics is the empty set. The atom `is_empty(`σ`, `$\bar{\Pi}$`)` succeeds for two different sets of abducted assumptions, $\bar{\Pi} = \{(null, \mathbf{1})\}$, or $\bar{\Pi} = \{(\mathbf{1}, null)\}$; in this case, only the second set of assumptions holds (that is, $\mathbf{1}$ is not a subtype of $null$).

Besides the implementation shown here, we have experimented another solution based on the implementation of the judgments $\Pi \vdash \sigma \simeq \emptyset$ and $\sigma_1 \not\leq \sigma_2$. Since both judgments are inductive, in this case stratification is not needed, and the benchmark shows that this last solution seems to be more efficient.

The benchmark consists of more than a hundred tests, including all examples presented in Section Section 2. Experiments have been performed with SWI Prolog 7.2.3, running on a i7-3610QM machine with GNU/Debian.

123 tests	$\not\leq$
Total time:	1.7e+00s
Average time:	1.4e-02s
Total number of inferences:	14 620 376
Average number of inferences:	118 865

Table 1. Benchmark results summary

The implementation based on $\not\leq$ performs two order of magnitude better than the one based on $\leq$.

We conjecture that this difference is due to the fact that the implementation based on $\not\leq$ employs two inductive predicates and hence does not rely on stratification.

Listing 1. Pseudo-code for is_empty

```
is_empty(σ, Π̄) ← is_empty(∅, σ, ∅, Π̄).

is_empty(_, 0, Π̄, Π̄) ← true. % rule (empty)
is_empty(Ψ, ⟨f⁺:π⟩, Π̄′, Π̄) ← ⟨f⁺:π⟩ ∉ Ψ, is_empty({⟨f⁺:π⟩} ∪ Ψ, norm(π), Π̄′, Π̄). % rule (rec-r)
is_empty(Ψ, ∨{}, Π̄′, Π̄) ← ∨{} ∉ Ψ. % rule (\/)
is_empty(Ψ, ∨{ς1,...,ςn}, Π̄′, Π̄) ← n > 0, ∨{ς1,...,ςn} ∉ Ψ, % rule (\/)
  is_empty(Ψ, ς1, Π̄′, Π̄′′), is_empty(Ψ, ∨{ς2,...,ςn}, Π̄′′, Π̄).
is_empty(Ψ, ∧{ι1,...,ιn}, Π̄′, Π̄) ← n > 0, ∧{ι1,...,ιn} ∉ Ψ, % rule (/\)
  partition_by_field(∧{ι1,...,ιn}, RecsMap, Others), % RecsMap map from fields to a set of record types
                                                     % Others contains all types that are not ⟨f⁺:π⟩ or ⟨f⁻:π⟩
  ((0 ∈Others, Π̄′=Π̄) or % rule (and)
   ((int ∈Others or null ∈Others), (size(Others) > 1 or not empty(RecsMap)), Π̄′=Π̄) or % rule (/\prim)
   (get_keys(RecsMap, Fs), is_empty_and(Ψ, Fs, RecsMap, Π̄′, Π̄))).

is_empty_and(Ψ, [F|_], RecsMap, Π̄′, Π̄) ← % F is a fieldname
    lookup(F, RecsMap, Recs), ⟨F⁺:π1⟩ ∈Recs,
    (is_empty(Ψ, ⟨F⁺:π1⟩, Π̄′, Π̄) or % rule (/\)
     ⟨F⁻:π2⟩ ∈Recs, Π̄={(π2,π1)} ∪ Π̄′). % rule (r/\w)
is_empty_and(Ψ, [_|Fs], RecsMap, Π̄′, Π̄) ← is_empty_and(Ψ, Fs, RecsMap, Π̄′, Π̄).
```

Listing 2. Pseudo-code for subtype

```
subtype(σ1, σ2) ← subtype(∅, σ1, σ2).

subtype(_, σ, σ) ← is_primitive(σ). % rule (prim)
subtype(_, _, any) ← true. % rule (any)
subtype(_, σ, ⟨⟩) ← (σ=⟨f⁺:_⟩ or σ=⟨f⁻:_⟩ or σ=⟨⟩). % rule (<>)
subtype(Ψ, σ1, σ2) ← (σ1,σ2) ∈ Ψ. % termination condition
subtype(Ψ, ∨{}, σ2) ← true. % rule (l-or)
subtype(Ψ, ∨{ς1,...,ςn}, σ2) ← n > 0, subtype(Ψ, ς1, σ2), subtype(Ψ, ∨{ς2,...,ςn}, σ2). % rule (l-or)
subtype(Ψ, σ1, ∨{ς1,...,ςn}) ← n > 0, subtype(Ψ, σ1, ς1)or subtype(Ψ, σ1, ∨{ς2,...,ςn}). % rule (r-or)
subtype(Ψ, σ1, σ2) ← is_empty(σ1, Π̄), % rule (empty)
  Ψ′ = {(σ1,σ2)} ∪ Ψ, ∀(π1,π2) ∈ Π̄ not subtype(Ψ′,norm(π1),norm(π2)).
subtype(Ψ, σ1, ∧{}) ← true. % rule (r-and)
subtype(Ψ, σ1, ∧{ι1,...,ιn}) ← n > 0, subtype(Ψ,σ1,ι1), subtype(Ψ, ∧{ι2,...,ιn}, σ1). % rule (r-and)
subtype(Ψ, ∧{ι1,...,ιn}, σ2) ← n > 0, subtype(Ψ, ι1, σ2) or subtype(Ψ,∧{ι2,...,ιn}, σ2). % rule (l-and)
subtype(Ψ, ⟨f⁺:π1⟩, ⟨f⁻:π2⟩) ← is_empty(π2, Π̄), % rule (r\w)
  Ψ′ = {(⟨f⁺:π1⟩,⟨f⁻:π2⟩)} ∪ Ψ, ∀(π1,π2) ∈ Π̄ not subtype(Ψ′,norm(π1),norm(π2)).
subtype(Ψ, ⟨f⁺:π1⟩, ⟨f⁺:π2⟩) ← % rule (r\r)
  Ψ′ = {(⟨f⁺:π1⟩,⟨f⁺:π2⟩)} ∪ Ψ, subtype(Ψ′,norm(π1),norm(π2)).
subtype(Ψ, ⟨f⁻:π1⟩, ⟨f⁻:π2⟩) ← % rule (w\w)
  Ψ′ = {(⟨f⁻:π1⟩,⟨f⁻:π2⟩)} ∪ Ψ, subtype(Ψ′,norm(π2),norm(π1)).
```

More details on the experimental results can be found in the documentation of the accompanying artifact.

6.2 Termination

We provide a proof sketch for termination of predicate `is_empty`.

The measure of the function `isEmpty` is defined by the function $\mathcal{D}(\Psi,\sigma)$.

$$\mathcal{D}(\Psi,\sigma) = \begin{cases} 1+\Sigma_{i=1}^{n}\mathcal{D}(\Psi,\varsigma_i) \text{ if } \sigma = \vee\{\varsigma_1,\ldots,\varsigma_n\} \\ 1+\Sigma_{i=1}^{n}\mathcal{D}(\Psi,\iota_i) \text{ if } \sigma = \wedge\{\iota_1,\ldots,\iota_n\} \\ 1+\mathcal{D}(\Psi\cup\{\sigma\},norm(\pi)) \text{ if } \sigma = \langle f^{+}{:}\pi\rangle \wedge \sigma \notin \Psi \\ 1 \text{ if } \sigma = \langle f^{+}{:}\pi\rangle \wedge \sigma \in \Psi \\ 1 \text{ if } \sigma \in \{\langle f^{-}{:}\_\rangle, int, null, \mathbf{0}, \mathbf{1}\} \end{cases}$$

For instance, let $T = \langle f^{+}{:}int\rangle \vee \langle g^{+}{:}T\rangle$, then we have that $norm(T) = \vee\{\langle f^{+}{:}int\rangle, \langle g^{+}{:}T\rangle\}$, and then:

$$\begin{aligned} \mathcal{D}(\emptyset, \vee\{\langle f^{+}{:}int\rangle, \langle g^{+}{:}T\rangle\}) = \\ 1+\mathcal{D}(\emptyset,\langle f^{+}{:}int\rangle)+\mathcal{D}(\emptyset,\langle g^{+}{:}T\rangle) = \\ 1+(1+\mathcal{D}(\{\langle f^{+}{:}int\rangle\},norm(int)))+ \\ (1+\mathcal{D}(\{\langle g^{+}{:}T\rangle\},norm(T))) = \\ 3+(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle\},\vee\{\langle f^{+}{:}int\rangle,\langle g^{+}{:}T\rangle\})) = \\ 4+(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle\},\langle f^{+}{:}int\rangle)+ \\ \mathcal{D}(\{\langle g^{+}{:}T\rangle\},\langle g^{+}{:}T\rangle)) = \\ 5+(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle,\langle f^{+}{:}int\rangle\},norm(int)))+1 = \\ 8 \end{aligned}$$

The function $\mathcal{D}(\Psi,\sigma)$ is well-defined, and returns positive numbers. The type σ is regular and contractive, hence the set of its subtrees is finite; $\vee\{\varsigma_1,\ldots,\varsigma_n\}$ and $\wedge\{\iota_1,\ldots,\iota_n\}$

are always bounded. The set Ψ can contain only subtrees (eventually normalized) of σ, hence is bounded.

The set Π can contain only pairs of types that are subtrees of σ, hence its size is bounded.

We now prove that the measure strictly decreases for each recursive occurrence of the predicate.

When $\sigma = \langle f^{+}{:}\pi \rangle$ let $\mathcal{D}(\Psi, \sigma)$ be the measure value, then the value for the recursive call is $\mathcal{D}(\Psi \cup \{\sigma\}, norm(\pi))$ and $\mathcal{D}(\Psi \cup \{\sigma\}, norm(\pi)) < \mathcal{D}(\Psi, \sigma)$.

When $\sigma = \vee\{\varsigma_1, \ldots, \varsigma_n\}$ let $\mathcal{D}(\Psi, \sigma)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, \varsigma_1)$ and $\mathcal{D}(\Psi, \varsigma_1) < \mathcal{D}(\Psi, \sigma)$; the value for the second recursive call is $\mathcal{D}(\Psi, \vee\{\varsigma_2 \ldots \varsigma_n\})$ and $\mathcal{D}(\Psi, \vee\{\varsigma_2 \ldots \varsigma_n\}) < \mathcal{D}(\Psi, \sigma)$.

When $\sigma = \wedge\{\iota_1, \ldots, \iota_n\}$ let $\mathcal{D}(\Psi, \sigma)$ be the measure value, then the value for the recursive call is $\mathcal{D}(\Psi, \iota_i)$ where $\iota_i = \langle f^{+}{:}\pi \rangle$ and $\mathcal{D}(\Psi, \iota_i) < \mathcal{D}(\Psi, \sigma)$.

As done for `is_empty`, we provide a proof sketch for termination of predicate `subtype`, based on the definition of a similar measure. We define $\delta_i(\Psi)$ as the set of the i-th component of the pairs in Ψ.

$\mathcal{D}(\Psi, \sigma_1, \sigma_2) = \Sigma_{i=1}^{2} \Sigma_{j=1}^{2} \mathcal{D}_{aux}(\delta_i(\Psi), \sigma_j)$

$\mathcal{D}_{aux}(\Psi, \sigma) =$

$$\begin{cases} 1 + \Sigma_{i=1}^{n} \mathcal{D}_{aux}(\Psi, \varsigma_i) \text{ if } \sigma = \vee\{\varsigma_1, \ldots, \varsigma_n\} \\ 1 + \Sigma_{i=1}^{n} \mathcal{D}_{aux}(\Psi, \iota_i) \text{ if } \sigma = \wedge\{\iota_1, \ldots, \iota_n\} \\ 1 + \mathcal{D}_{aux}(\Psi \cup \{\sigma\}, norm(\pi)) \text{ if } \sigma = \langle f^{\nu}{:}\pi \rangle \wedge \sigma \notin \Psi \\ 1 \text{ if } \sigma = \langle f^{\nu}{:}\pi \rangle \wedge \sigma \in \Psi \\ 1 \text{ if } \sigma \in \{int, null, \mathbf{0}, \mathbf{1}\} \end{cases}$$

We now prove that the measure strictly decreases for each recursive occurrence of the predicate.

When $\sigma_1 = \vee\{\varsigma_1, \ldots, \varsigma_n\}$ let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, \varsigma, \sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi), \varsigma) < \mathcal{D}_{aux}(\delta_1(\Psi), \sigma_1)$; the value for the second recursive call is $\mathcal{D}(\Psi, \vee\{\varsigma_2, \ldots, \varsigma_n\}, \sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi), \vee\{\varsigma_2, \ldots, \varsigma_n\}) < \mathcal{D}_{aux}(\delta_1(\Psi), \sigma_1)$.

When $\sigma_2 = \vee\{\varsigma_1, \ldots, \varsigma_n\}$ let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, \sigma_1, \varsigma_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi), \varsigma_1) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$; then, the value for the second recursive call in clause is $\mathcal{D}(\Psi, \sigma_1, \vee\{\varsigma_2, \ldots, \varsigma_n\})$ and $\mathcal{D}_{aux}(\delta_2(\Psi), \vee\{\varsigma_2, \ldots, \varsigma_n\}) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$.

In the case corresponding to rule $(empty)$ let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, norm(\pi_1), norm(\pi_2))$, since $\bar{\Pi}$ contains only subterms of σ_2 we have that $\mathcal{D}(\Psi, norm(\pi_1), norm(\pi_2)) < \Sigma_{i=1}^{2} \mathcal{D}_{aux}(\delta_i(\Psi), \sigma_2)$.

When $\sigma_1 = \wedge\{\iota_1, \ldots, \iota_n\}$ let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, \iota_1, \sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi), \iota_1) < \mathcal{D}_{aux}(\delta_1(\Psi), \sigma_1)$; the value for the second recursive call is $\mathcal{D}(\Psi, \wedge\{\iota_2, \ldots, \iota_n\}, \sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi), \wedge\{\iota_2, \ldots, \iota_n\}) < \mathcal{D}_{aux}(\delta_1(\Psi), \sigma_1)$.

When $\sigma_2 = \wedge\{\iota_1, \ldots, \iota_n\}$ let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, \sigma_1, \iota_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi), \iota_1) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$; the value for the second recursive call is $\mathcal{D}(\Psi, \sigma_1, \wedge\{\iota_2, \ldots, \iota_n\})$ and $\mathcal{D}_{aux}(\delta_2(\Psi), \wedge\{\iota_2, \ldots, \iota_n\}) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$.

When $\sigma_1 = \langle f^{+}{:}\_ \rangle$ and $\sigma_2 = \langle f^{-}{:}\pi \rangle$, let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, and $\Psi' = \{(\sigma_1, \sigma_2)\} \cup \Psi$; then the value for the recursive call is $\mathcal{D}(\Psi, norm(\pi_1), norm(\pi_2))$; note that π_1 and π_2 are subterms of $norm(\pi)$ and that $\mathcal{D}_{aux}(\delta_2(\Psi), norm(\pi_1)) + \mathcal{D}_{aux}(\delta_2(\Psi), norm(\pi_2)) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$.

When $\sigma_1 = \langle f^{+}{:}\pi_1 \rangle$ and $\sigma_2 = \langle f^{+}{:}\pi_2 \rangle$, let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, and $\Psi' = \{(\sigma_1, \sigma_2)\} \cup \Psi$; then the value for the recursive call is $\mathcal{D}(\Psi', norm(\pi_1), norm(\pi_2))$ and $\mathcal{D}_{aux}(\delta_1(\Psi'), norm(\pi_1)) < \mathcal{D}_{aux}(\delta_1(\Psi), \sigma_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi'), norm(\pi_2)) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$.

When $\sigma_1 = \langle f^{-}{:}\pi_1 \rangle$ and $\sigma_2 = \langle f^{-}{:}\pi_2 \rangle$ let $\mathcal{D}(\Psi, \sigma_1, \sigma_2)$ be the measure value, and $\Psi' = \{(\sigma_1, \sigma_2)\} \cup \Psi$ then the value for the recursive call is $\mathcal{D}(\Psi', norm(\pi_2), norm(\pi_1))$ and $\mathcal{D}_{aux}(\delta_1(\Psi'), norm(\pi_1)) < \mathcal{D}_{aux}(\delta_1(\Psi), \sigma_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi'), norm(\pi_2)) < \mathcal{D}_{aux}(\delta_2(\Psi), \sigma_2)$.

7. Conclusion

In this paper we have investigated a coinductive semantic model for record types with read/write field annotations, supporting union, intersection, and recursive types.

We have provided an interpretation of types which accommodates read/write annotations with the semantic subtyping approach, so that subtyping corresponds to set inclusion between type interpretations. Although intuitive, the model poses some challenging issues, since subtyping is defined in terms of values, but values in turn contain type annotations to indicate what can be safely assigned to fields. This leads to a circular definition between the coinductive judgment for typing values, and its negation, which is inductive; to break this circularity, the inductive judgment depends on a set of type assumptions on values that must hold for the judgment to be derivable.

The semantic model has allowed us to study the main laws underlying subtyping for record types with read/write field annotations, supporting union, intersection, and recursive types.

Furthermore, we have tackled the challenging problem of defining a system of subtyping rules which are sound and complete w.r.t. the definition of semantic subtyping. Similar issues concerning circularity between coinductive judgments and their corresponding negations (which, by duality, are inductive) have been faced.

In Section 6 we show how such rules can be effectively implemented.

To our knowledge, this is the first implementation of a sound and complete procedure to decide subtyping for record types with read/write field annotations, supporting union, intersection, and recursive types.

There are several interesting directions for future extensions to types and the corresponding subtyping relation. In Section 2 we have shown that union and intersection types to-

gether with read/write annotations can be used to enforce monotonic initialization for fields; for instance, the type $\langle f^+{:}null \vee T\rangle \wedge \langle f^-{:}T\rangle$ allows field f to be associated with the initial null value, but forces any assignment to f to store a non-null value (if we assume that $null \not\leq T$). Once a value of type T has been assigned to field $x.f$, a system supporting strong updates would allow narrowing the type of x from $\langle f^+{:}null \vee T\rangle \wedge \langle f^-{:}T\rangle$ to $\langle f^+{:}T\rangle \wedge \langle f^-{:}T\rangle$; in general, to be sound, such a narrowing requires non trivial points-to analysis, because x could be aliased by a variable y having the subtype $\langle f^+{:}null \vee T\rangle \wedge \langle f^-{:}null \vee T\rangle$, hence the null value could be reassigned to field $x.f$ through y. While write-only fields allow safe contravariant subtyping, it would be also possible to introduce record types $\langle f^{\ominus}{:}T\rangle$ with write-only invariant fields, s.t. $\langle f^{\ominus}{:}T_1\rangle \leq \langle f^{\ominus}{:}T_2\rangle$ iff $T_1 \equiv T_2$. Invariant write-only fields support strong updates without requiring points-to analysis: for instance, after field $x.f$ is assigned to a value of type T, the type of x can be safely narrowed from $\langle f^+{:}null \vee T\rangle \wedge \langle f^{\ominus}{:}T\rangle$ to $\langle f^+{:}T\rangle \wedge \langle f^{\ominus}{:}T\rangle$, independently of aliasing, because x can be aliased by variables whose type only allows assignment of values of type T to field f.

Another interesting extension would consist in the introduction of record types with negative information to specify the absence of fields.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical logic*. North Holland, 1977.

[3] J. Altidor, C. Reichenbach, and Y. Smaragdakis. Java wildcards meet definition-site variance. In *ECOOP'12*, pages 509–534, 2012.

[4] D. Ancona and A. Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference*, volume 8586 of *Lecture Notes in Computer Science*, pages 282–307. Springer, 2014.

[5] D. Ancona and A. Corradi. Semantic subtyping between coinductive mutable record types with unions and intersections. In *Italian Conference on Theoretical Computer Science (ICTCS 2015)*, 2015. On-line proceedings.

[6] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *LNCS*, pages 2–26. Springer, 2009. ISBN 978-3-642-03012-3.

[7] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP 2005*, pages 428–452, 2005.

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

[9] M. Bonsangue, J. Rot, D. Ancona, F. de Boer, and J. Rutten. A coalgebraic foundation for coinductive union types. In *ICALP 2014 - 41st International Colloquium on Automata, Languages and Programming*, pages 62–73, 2014.

[10] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *ECOOP 2008*, pages 2–26, 2008.

[11] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. *Theoretical Computer Science*, 398 (1-3):217–242, 2008. doi: 10.1016/j.tcs.2008.01.049.

[12] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL '14*, pages 5–18, 2014.

[13] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *POPL 2015*, pages 289–302, 2015.

[14] G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. In *ICFP 2016*, 2016. To appear.

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

[16] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA 2003*, pages 302–312, 2003.

[17] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

[18] J. Gil and I. Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA '08*, 2008.

[19] B. Hackett and S. Guo. Fast and precise hybrid type inference for javascript. In *PLDI '12*, pages 239–250, 2012.

[20] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP 2010*, pages 200–224, 2010.

[21] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

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

[23] T. Jones, M. Homer, and J. Noble. Brand Objects for Nominal Typing. In *ECOOP 2015*, pages 198–221, 2015.

[24] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *DLS'13*, pages 17–26, 2013.

[25] B. Lerner, J. Politz, A. Guha, and S. Krishnamurthi. Tejas: retrofitting type systems for javascript. In *DLS'13*, pages 1–16, 2013.

[26] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207:284–304, 2009.

[27] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP 2008*, pages 260–284, 2008.

[28] 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.

[29] M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, 2004.

[30] T. Wang and S. 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.