Decidable, Tag-Based Semantic Subtyping for Nominal Types, Tuples, and Unions

Julia Belyakova
Northeastern University
belyakova.y@northeastern.edu

Abstract

Subtyping is used in many programming languages for static type checking and dynamic dispatch. *Semantic* subtyping enables simple, set-theoretical reasoning about types by interpreting a type as a set of its values. Previously, semantic subtyping has been studied primarily in the context of statically typed languages with structural typing. In this paper, we are bridging the gap between semantic subtyping and *dynamic* languages with *nominal* types, which rely on run-time checks of type tags instead of preventive static type checking. We propose *tag-based semantic* subtyping for nominal types, tuples, and unions; types are interpreted set-theoretically, as sets of type tags (instead of values). The proposed subtyping relation is shown to be decidable, and a corresponding syntax-directed definition is provided. The implications of using semantic subtyping for multiple dispatch in a dynamic language are also discussed.

Keywords semantic subtyping, type tags, nominal typing, pair-union distributivity, decidability

1 Introduction

Many static type systems rely on subtyping. Informally, a subtyping relation `T <: S` states that a value of type `T` is safe to use in the context that expects a value of type `S`. For example, if class `Rectangle` is a subtype of class `Shape`, an instance of `Rectangle` can be passed to a function with an argument of type `Shape`.

In languages with nominal typing, subtyping can also be used for run-time dispatch, in particular, *multiple dynamic dispatch* (MDD) [4, 5]. It allows a function to have several implementations for different types of arguments, and the most suitable implementation for a particular call is picked dynamically, based on the run-time types of all arguments. For example, consider two implementations of addition, `+(Number, Number)` and `+(String, String)`, and the call `3+5`. In this case, a language run-time should pick the implementation for numbers because `Int <: Number` but `Int` $\not<$`: String`.

It is often convenient to think of subtyping `T <: S` in terms of the set inclusion: "the elements of `T` are a subset of the elements of `S`" [13]. This intuition is not always correct, but there has been research on so-called *semantic subtyping* [1, 7, 8] where subtyping is defined exactly as the subset relation. Namely, types are given the set-theoretic interpretation $[\![\tau]\!] = \{v \mid \vdash v : \tau\}$, and subtyping $\tau_1 <: \tau_2$ is defined as $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. In this way, a semantic definition of subtyping intertwines with a static typing relation.

However, subtyping is not limited to statically typed languages — it is also applicable in the context of *dynamically* typed ones. As mentioned before, subtyping can be used for multiple dynamic dispatch, and MDD is rather widespread among dynamic languages, for instance, Common Lisp, Julia, Clojure. Such languages do not prevent type errors with static type checking but detect them at run-time, by inspecting type tags associated with values. Type tags are also used during dynamic dispatch: the language run-time looks at the tags attached to the arguments to find out their types.

In this paper, we are bridging the gap between *semantic subtyping* and *dynamically* typed languages with *nominal* types. Instead of directly interpreting types as sets of values, we interpret them as sets of *type tags* assuming each value is associated with a tag. Our contributions are as follows:

1. Tag-based semantic interpretation of types for a language with nominal types, tuples, and unions (Sec. 2).
2. Two syntactic definitions of subtyping, declarative and reductive, along with the Coq-mechanized proofs that the definitions are equivalent and coincide with the semantic interpretation (Sec. 3).
3. Proof of decidability of the reductive subtyping.
4. Discussion of the implications of using semantic subtyping for multiple dynamic dispatch (Sec. 4).

2 Semantic Subtyping in MiniJl

We base this work on a small type language MiniJl, presented in Fig. 1. Types, denoted by $\tau \in$ Type, include pairs, unions, and nominal types: *cname* denotes *concrete* nominal types that have direct instances, and *aname* denotes *abstract* nominal types.

To simplify the development, we decided to work with a particular hierarchy of nominal types (presented in Fig. 1 as a tree) instead of a general class table. There are four concrete, leaf types (depicted in rectangles) and two abstract types in the hierarchy. Formally, the hierarchy can be represented with a list of declarations $n_1 \rhd n_2$ read as "n_1 is a declared subtype of n_2" where n is either *cname* or *aname*. In the case

FTfJP'19, July 15–19, 2019, London, United Kingdom
2019.

$$
\begin{array}{llll}
\tau \in \textsc{Type} & ::= & & \textit{Types} \\
& \mid & \tau_1 \times \tau_2 & \text{covariant pair} \\
& \mid & \tau_1 \cup \tau_2 & \text{untagged union} \\
& \mid & \textit{cname} & \text{concrete nominal type} \\
& \mid & \textit{aname} & \text{abstract nominal type}
\end{array}
$$

$$
\begin{array}{lll}
\textit{cname} & \in & \{\mathsf{Int}, \mathsf{Flt}, \mathsf{Cmplx}, \mathsf{Str}\} \\
\textit{aname} & \in & \{\mathsf{Real}, \mathsf{Num}\}
\end{array}
$$

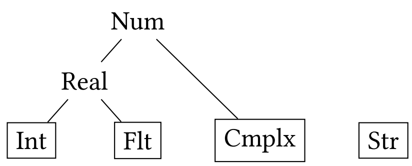

Figure 1. MiniJl: type grammar and nominal hierarchy

$$
\begin{array}{llll}
v \in \textsc{ValType} & ::= & & \textit{Value Types} \\
& \mid & \textit{cname} & \text{concrete nominal type} \\
& \mid & v_1 \times v_2 & \text{pair of value types}
\end{array}
$$

Figure 2. Value types

of MiniJl, the hierarchy is defined as follows:

$$\mathsf{NomHrc} = [\mathsf{Real}{\rhd}\mathsf{Num}, \mathsf{Int}{\rhd}\mathsf{Real}, \mathsf{Flt}{\rhd}\mathsf{Real}, \mathsf{Cmplx}{\rhd}\mathsf{Num}].$$

Nominal hierarchies should not have cycles, and each type can have only one parent. In the spirit of the Julia language, we do not allow concrete types to have declared subtypes, i.e. concrete types are always leaves.

Value Types Only types that can be instantiated can serve as type tags, and we call these types **value types**. Their formal definition is given in Fig. 2: value type $v \in \textsc{ValType}$ is either a concrete nominal type or a pair of value types. For example, Flt, $\mathsf{Int}\times\mathsf{Int}$, and $\mathsf{Str}\times(\mathsf{Int}\times\mathsf{Int})$ are all value types. Union types, just as abstract nominal types, are not value types. Even the type $\mathsf{Int}\cup\mathsf{Int}$ is not a value type, though it describes the same set of values as the value type Int. Note that $\textsc{ValType} \subset \textsc{Type}$, i.e. each value type is a type.

2.1 Semantic Interpretation of Types

As mentioned in Sec. 1, we interpret types as sets of type tags (i.e. value types) instead of values, so we call this semantic interpretation *tag-based.* Formally, the interpretation is given by the function $[\![\cdot]\!]$ that maps a type $\tau \in \textsc{Type}$ into a set of value types $s \in \mathcal{P}(\textsc{ValType})$, as presented in Fig. 3.

The interpretation of a type states what values have that type: if $v \in [\![\tau]\!]$, then values ν tagged with v (i.e. instances of v) belong to τ. Thus, a *concrete nominal* type *cname* is comprised only of its direct instances. *Abstract nominal* types do not have direct instances, but we want their interpretation to reflect the nominal hierarchy: for example, a Num value is

$$
\begin{array}{rcl}
[\![\cdot]\!] : \textsc{Type} & \to & \mathcal{P}(\textsc{ValType}) \\
[\![\textit{cname}]\!] & = & \{\textit{cname}\} \\
[\![\mathsf{Real}]\!] & = & \{\mathsf{Int}, \mathsf{Flt}\} \\
[\![\mathsf{Num}]\!] & = & \{\mathsf{Int}, \mathsf{Flt}, \mathsf{Cmplx}\} \\
[\![\tau_1 \times \tau_2]\!] & = & \{v_1 \times v_2 \mid v_1 \in [\![\tau_1]\!], v_2 \in [\![\tau_2]\!]\} \\
[\![\tau_1 \cup \tau_2]\!] & = & [\![\tau_1]\!] \cup [\![\tau_2]\!]
\end{array}
$$

Figure 3. Tag-based semantic interpretation of types

either a concrete complex or a real number, which, in turn, is either a concrete integer or a floating point value. Therefore, the set of values of type Num is described by the set of value types $\{\mathsf{Cmplx}, \mathsf{Int}, \mathsf{Flt}\}$. More generally, the interpretation of abstract nominal types *aname* can be given as follows:

$$[\![\textit{aname}]\!] = \{\textit{cname} \mid \textit{cname} \rhd^* \textit{aname}\},$$

where the relation $n_1 \rhd^* n_2$ means that n_1 is transitively a declared subtype of n_2:

$$\frac{n_1 \rhd n_2 \in \mathsf{NomHrc}}{n_1 \rhd^* n_2} \qquad \frac{n_1 \rhd^* n_2 \qquad n_2 \rhd^* n_3}{n_1 \rhd^* n_3}.$$

Finally, *pairs* and *unions* are interpreted set-theoretically.

Once we have the interpretation of types, we define **tag-based semantic subtyping** as the subset relation:

$$\tau_1 \overset{\text{sem}}{<:} \tau_2 \quad \overset{\text{def}}{\equiv} \quad [\![\tau_1]\!] \subseteq [\![\tau_2]\!].$$

2.2 Syntactic Model of Semantic Subtyping

Let us unroll the definition of semantic subtyping[1]:

$$\tau_1 \overset{\text{sem}}{<:} \tau_2 \quad \overset{\text{def}}{\equiv} \quad \forall v.\, (v \in [\![\tau_1]\!] \implies v \in [\![\tau_2]\!]). \qquad (1)$$

Its main ingredient is the relation $v \in [\![\tau]\!]$, which has a syntactic counterpart — an equivalent[2] inductive, syntax-directed relation $v \prec \tau$ defined in Fig. 4. We call the latter a **matching relation**, read "tag v matches type τ".

Using the matching relation, we define a syntactic model of semantic subtyping $\tau_1 <: \tau_2$ as follows:

$$\tau_1 <: \tau_2 \quad \overset{\text{def}}{\equiv} \quad \forall v.\, (v \prec \tau_1 \implies v \prec \tau_2). \qquad (2)$$

Because $v \in [\![\tau]\!]$ and $v \prec \tau$ are equivalent, definitions (1) and (2) are also equivalent:

$$\tau_1 <: \tau_2 \quad \Longleftrightarrow \quad \tau_1 \overset{\text{sem}}{<:} \tau_2. \qquad (3)$$

We will use the syntactic model (2) in Sec. 3.2 to prove that alternative, syntactic definitions of subtyping coincide with the semantic interpretation.

[1] In set theory, the subset relation $X \subseteq Y \overset{\text{def}}{\equiv} \forall x.\, (x \in X \implies x \in Y)$.
[2] It is easy to show by induction on τ that the matching relation is sound and complete with respect to the member-of-interpretation relation, i.e. $\forall v.\, (v \prec \tau \iff v \in [\![\tau]\!])$.

$$\frac{}{\mathit{cname} \prec \mathit{cname}}\ \textsc{MT-CName}$$

$$\frac{}{\mathsf{Int} \prec \mathsf{Real}}\ \textsc{MT-IntReal} \qquad \frac{}{\mathsf{Flt} \prec \mathsf{Real}}\ \textsc{MT-FltReal}$$

$$\frac{}{\mathsf{Int} \prec \mathsf{Num}}\ \textsc{MT-IntNum} \qquad \frac{}{\mathsf{Flt} \prec \mathsf{Num}}\ \textsc{MT-FltNum} \qquad \frac{}{\mathsf{Cmplx} \prec \mathsf{Num}}\ \textsc{MT-CmplxNum}$$

$$\frac{v_1 \prec \tau_1 \qquad v_2 \prec \tau_2}{v_1 \times v_2 \prec \tau_1 \times \tau_2}\ \textsc{MT-Pair}$$

$$\frac{v \prec \tau_1}{v \prec \tau_1 \cup \tau_2}\ \textsc{MT-Union1} \qquad \frac{v \prec \tau_2}{v \prec \tau_1 \cup \tau_2}\ \textsc{MT-Union2}$$

Figure 4. Matching relation in MiniJl

3 Syntactic Definitions of Subtyping

While the semantic approach does provide us with a useful intuition, we need to be able to build the subtyping algorithm in order to compute subtypes. However, neither of the definitions (1), (2) can be directly computed because of the quantification $\forall v$. Therefore, we provide a *syntactic* definition of subtyping, equivalent to the semantic one, which is straightforward to implement.

We do this in two steps. First, we give an inductive definition that is handy to reason about, called *declarative* subtyping. We prove it equivalent to the semantic subtyping using the syntactic model discussed above. Second, we provide a *reductive*, syntax-directed definition of subtyping and prove it equivalent to the declarative definition (and, hence, the semantic one as well). We prove that the reductive subtyping is decidable, that is, for any two types τ_1 and τ_2, it is possible to prove that τ_1 either is a subtype of τ_2 or is not. Since the Coq-proof is constructive, it essentially gives an algorithm to decide subtyping. However, it is also possible to devise the algorithm as a straightforward recursive function.

3.1 Declarative Subtyping

The declarative definition of subtyping is provided in Fig. 5. The definition is mostly comprised of the standard rules of syntactic subtyping for unions and pairs. Namely, reflexivity and transitivity (SD-Refl and SD-Trans), subtyping of pairs (SD-Pairs), and subtyping of unions (SD-UnionL, SD-UnionR1, SD-UnionR2). Though SD-UnionR* rules are seemingly very strict, transitivity allows us to derive judgments such as $\mathsf{Int} \le \mathsf{Str} \cup \mathsf{Real}$ via $\mathsf{Int} \le \mathsf{Real}$ and $\mathsf{Real} \le \mathsf{Str} \cup \mathsf{Real}$. Note that we do need the syntactic definition of subtyping to be *reflexive* and *transitive* because so is semantic subtyping, which is just the subset relation.

Semantic subtyping also forces us to add rules for distributing pairs over unions, SD-Distr1 and SD-Distr2. For example, consider two types, $(\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Flt})$ and $\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})$. They have the same semantic interpretation — $\{\mathsf{Str} \times \mathsf{Int}, \mathsf{Str} \times \mathsf{Flt}\}$. Therefore, we should be able to derive the equivalence of the types using the declarative definition, i.e., declarative subtyping should hold in both directions. One direction is trivial:

$$\frac{\dfrac{\mathsf{Str} \le \mathsf{Str} \qquad \mathsf{Int} \le \mathsf{Int} \cup \mathsf{Flt}}{\mathsf{Str} \times \mathsf{Int} \le \mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})} \qquad \dfrac{\ldots}{\mathsf{Str} \times \mathsf{Flt} \le \ldots}}{(\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Flt}) \le \mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})}\ .$$

But the other direction,

$$\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt}) \le (\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Flt}),$$

cannot be derived without SD-Distr2 rule: $\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})$ is not a subtype of either $\mathsf{Str} \times \mathsf{Int}$ or $\mathsf{Str} \times \mathsf{Flt}$, so we cannot apply SD-UnionR* rules[3].

The most interesting part of the definition hides in subtyping of nominal types. There are four obvious rules coming directly from the nominal hierarchy. For instance, SD-RealNum mirrors the fact that $\mathsf{Real} \rhd \mathsf{Num} \in \mathsf{NomHrc}$. But there are also new rules, SD-RealUnion and SD-NumUnion, (highlighted in Fig. 5), which are dictated by semantic subtyping. For example, we need SD-RealUnion to prove the equivalence of types $\mathsf{Int} \cup \mathsf{Flt}$ and Real, which have the same interpretation — $\{\mathsf{Int}, \mathsf{Flt}\}$.

3.2 Correctness of Declarative Subtyping

In order to show that the declarative subtyping is equivalent to the semantic one, we need to prove that the former is sound and complete with respect to the latter, that is:

$$\forall \tau_1, \tau_2.\ (\tau_1 \le \tau_2 \iff \tau_1 \overset{\text{sem}}{<:} \tau_2).$$

As discussed in Sec. 2.2, the syntactic model $\tau_1 <: \tau_2$ is equivalent to the semantic definition of subtyping $\tau_1 \overset{\text{sem}}{<:} \tau_2$. We will use the model in our proofs.

Theorem 1 (Correctness of Declarative Subtyping)**.**

$$\forall \tau_1, \tau_2.\ (\tau_1 \le \tau_2 \iff \tau_1 <: \tau_2)$$

In order to prove the theorem, we need several auxiliary observations. Let us recall the definition of $\tau_1 <: \tau_2$:

$$\tau_1 <: \tau_2 \quad \overset{\text{def}}{\equiv} \quad \forall v.\ (v \prec \tau_1 \implies v \prec \tau_2).$$

The first thing to note is that subtyping a value type coincides with matching:

$$\forall v, \tau.\ (v \le \tau \iff v \prec \tau). \tag{4}$$

Having that, it is easy to prove the *soundness* direction of Theorem 1.

Lemma 1 (Soundness of Declarative Subtyping)**.**

$$\forall \tau_1, \tau_2.\ [\tau_1 \le \tau_2 \implies \forall v.\ (v \prec \tau_1 \implies v \prec \tau_2)]$$

[3]Transitivity does not help in this case.

$$\frac{}{\tau \ \le\ \tau}\ \textsc{SD-Refl} \qquad \frac{\tau_1 \ \le\ \tau_2 \qquad \tau_2 \ \le\ \tau_3}{\tau_1 \ \le\ \tau_3}\ \textsc{SD-Trans}$$

$$\frac{\textsc{SD-IntReal}}{\mathsf{Int} \ \le\ \mathsf{Real}} \qquad \frac{\textsc{SD-FltReal}}{\mathsf{Flt} \ \le\ \mathsf{Real}}$$

$$\frac{\textsc{SD-RealNum}}{\mathsf{Real} \ \le\ \mathsf{Num}} \qquad \frac{\textsc{SD-CmplxNum}}{\mathsf{Cmplx} \ \le\ \mathsf{Num}}$$

$$\frac{\textsc{SD-RealUnion}}{\mathsf{Real} \ \le\ \mathsf{Int} \cup \mathsf{Flt}} \qquad \frac{\textsc{SD-NumUnion}}{\mathsf{Num} \ \le\ \mathsf{Real} \cup \mathsf{Cmplx}}$$

$$\frac{\tau_1 \ \le\ \tau_1' \qquad \tau_2 \ \le\ \tau_2'}{\tau_1 \times \tau_2 \ \le\ \tau_1' \times \tau_2'}\ \textsc{SD-Pair}$$

$$\frac{\tau_1 \ \le\ \tau' \qquad \tau_2 \ \le\ \tau'}{\tau_1 \cup \tau_2 \ \le\ \tau'}\ \textsc{SD-UnionL}$$

$$\frac{\textsc{SD-UnionR1}}{\tau_1 \ \le\ \tau_1 \cup \tau_2} \qquad \frac{\textsc{SD-UnionR2}}{\tau_2 \ \le\ \tau_1 \cup \tau_2}$$

$$\frac{}{(\tau_{11} \cup \tau_{12}) \times \tau_2 \ \le\ (\tau_{11} \times \tau_2) \cup (\tau_{12} \times \tau_2)}\ \textsc{SD-Distr1}$$

$$\frac{}{\tau_1 \times (\tau_{21} \cup \tau_{22}) \ \le\ (\tau_1 \times \tau_{21}) \cup (\tau_1 \times \tau_{22})}\ \textsc{SD-Distr2}$$

Figure 5. Declarative subtyping for MiniJl

Proof. We know $v \prec \tau_1$ and $\tau_1 \ \le\ \tau_2$. We need to show that $v \prec \tau_2$. First, we apply (4) to $v \prec \tau_1$ and $v \prec \tau_2$. Now it suffices to show that $v \ \le\ \tau_2$ follows from $v \ \le\ \tau_1$ and $\tau_1 \ \le\ \tau_2$, which is trivially true by SD-Trans. □

Lemma 2 (Completeness of Declarative Subtyping)**.**

$$\forall \tau_1, \tau_2.\ (\tau_1 \ <:\ \tau_2 \implies \tau_1 \ \le\ \tau_2)$$

This direction of Theorem 1 is more challenging. The key observation here is that Lemma 2 can be shown for τ_1 of the form $v_1 \cup v_2 \cup \ldots \cup v_n$ (we omit parenthesis because union is associative). In this case, in the definition of $\tau_1 \ <:\ \tau_2$ the only vs that match τ_1 and τ_2 are v_i. By (4) we know that matching implies subtyping, so we also have $v_i \ \le\ \tau_2$. From the latter, it is easy to show that $\tau_1 \ \le\ \tau_2$ because τ_1 is just a union of value types, and subtyping of the left-hand side union amounts to subtyping its components, according to the SD-UnionL rule.

Normal Form We say that a type $\tau \equiv v_1 \cup v_2 \cup \ldots \cup v_n$ is in **normal form** and denote this fact by $\mathrm{InNF}(\tau)$ (formal definition of InNF is given in Fig. 8, App. A). For each type τ, there is an equivalent normalized type that can be computed

$$\begin{array}{rcl}
\mathrm{NF} : \textsc{Type} & \longrightarrow & \textsc{Type} \\
\mathrm{NF}(\mathit{cname}) & = & \mathit{cname} \\
\mathrm{NF}\ (\mathsf{Real}) & = & \mathsf{Int} \cup \mathsf{Flt} \\
\mathrm{NF}\ (\mathsf{Num}) & = & \mathsf{Int} \cup \mathsf{Flt} \cup \mathsf{Cmplx} \\
\mathrm{NF}(\tau_1 \times \tau_2) & = & \mathtt{un_prs}(\mathrm{NF}(\tau_1), \mathrm{NF}(\tau_2)) \\
\mathrm{NF}(\tau_1 \cup \tau_2) & = & \mathrm{NF}(\tau_1) \cup \mathrm{NF}(\tau_2)
\end{array}$$

Figure 6. Computing normal form of MiniJl types

with the function NF defined in Fig. 6 (the auxiliary function `un_prs` can be found in Fig. 9, App. A). Note that abstract nominal types are unfolded into unions of all their value subtypes. A pairs gets rewritten into a union of value pairs, thus producing a type in the disjunctive normal form.

Using the fact that every type can be normalized, and that declarative subtyping is complete for normalized types, we can finally prove Lemma 2.

Lemma 3 (Properties of the Normal Form)**.**

$$\forall \tau.\ (\mathrm{InNF}(\mathrm{NF}(\tau)) \ \wedge\ \tau \ \le\ \mathrm{NF}(\tau) \ \wedge\ \mathrm{NF}(\tau) \ \le\ \tau)$$

Lemma 4 (Completeness for Normalized Types)**.**

$$\forall \tau_1, \tau_2 \mid \mathrm{InNF}(\tau_1).\ (\tau_1 \ <:\ \tau_2 \implies \tau_1 \ \le\ \tau_2)$$

Lemma 5. $\forall \tau_1, \tau_2.\ (\tau_1 \ <:\ \tau_2 \implies \mathrm{NF}(\tau_1) \ <:\ \tau_2)$

Proof (Lemma 2). We know $\tau_1 \ <:\ \tau_2$, and we need to show $\tau_1 \ \le\ \tau_2$. First, we apply Lemma 5 to $\tau_1 \ <:\ \tau_2$, and then Lemma 4, this gives us $\mathrm{NF}(\tau_1) \ \le\ \tau_2$. Using Lemma 3 and SD-Trans, we can show $\tau_1 \ \le\ \tau_2$. □

3.3 Reductive Subtyping

The declarative definition is not syntax-directed because it involves the SD-Trans rule, which requires "coming up" with a middle type τ_2. For instance, in order to show

$$\mathsf{Str} \times \mathsf{Real} \ \le\ (\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Str}) \cup (\mathsf{Str} \times \mathsf{Flt}),$$

we need to apply transitivity several times, in particular, with the middle type $\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})$.

Fig. 7 presents the syntax-directed reductive definition of subtyping, which can be easily turned into a subtyping algorithm.[4] Some of the inductive rules are similar to the declarative definition, e.g. subtyping of pairs (SR-Pair) and the left-hand side union (SR-UnionL). The differing rules are highlighted. Instead of the general reflexivity rule SD-Refl, the reductive definition has reflexivity only for concrete nominal types (SR-BaseRefl). General transitivity is also gone; instead, it gets incorporated into subtyping of nominal types (SR-IntNum, SR-FltNum) and the right-hand side union (SR-UnionR1, SR-UnionR2), as well as the rule for using the

[4]The definition is not deterministic, though. For example, there are two ways to derive $\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt}) \ \le_{\mathrm{R}}\ \mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})$: either by immediately applying SR-Pair, or by first normalizing the left-hand side with SR-NF.

$$\frac{}{cname \ \leq_R \ cname}\ \text{SR-BaseRefl}$$

$$\text{SR-IntReal}\ \frac{}{\mathsf{Int} \ \leq_R \ \mathsf{Real}} \qquad \text{SR-FltReal}\ \frac{}{\mathsf{Flt} \ \leq_R \ \mathsf{Real}}$$

$$\text{SR-CmplxNum}\ \frac{}{\mathsf{Cmplx} \ \leq_R \ \mathsf{Num}} \qquad \text{SR-IntNum}\ \frac{}{\mathsf{Int} \ \leq_R \ \mathsf{Num}} \qquad \text{SR-FltNum}\ \frac{}{\mathsf{Flt} \ \leq_R \ \mathsf{Num}}$$

$$\frac{\tau_1 \ \leq_R \ \tau_1' \qquad \tau_2 \ \leq_R \ \tau_2'}{\tau_1 \times \tau_2 \ \leq_R \ \tau_1' \times \tau_2'}\ \text{SR-Pair}$$

$$\frac{\tau_1 \ \leq_R \ \tau' \qquad \tau_2 \ \leq_R \ \tau'}{\tau_1 \cup \tau_2 \ \leq_R \ \tau'}\ \text{SR-UnionL}$$

$$\text{SR-UnionR1}\ \frac{\tau \ \leq_R \ \tau_1'}{\tau \ \leq_R \ \tau_1' \cup \tau_2'} \qquad \text{SR-UnionR2}\ \frac{\tau \ \leq_R \ \tau_2'}{\tau \ \leq_R \ \tau_1' \cup \tau_2'}$$

$$\frac{\mathrm{NF}(\tau) \ \leq_R \ \tau'}{\tau \ \leq_R \ \tau'}\ \text{SR-NF}$$

Figure 7. Reductive subtyping for MiniJl

normal form (SR-NF). Note that the last rule also takes care of distributivity.

With the reductive subtyping, the example above can be derived as follows:

$$\cfrac{\cfrac{\mathsf{Str} \times \mathsf{Int} \ \leq \ (\mathsf{Str} \times \mathsf{Int}) \ldots \qquad \mathsf{Str} \times \mathsf{Flt} \ \leq \ \ldots (\mathsf{Str} \times \mathsf{Flt})}{(\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Flt}) \ \leq \ (\mathsf{Str} \times \mathsf{Int}) \cup \ldots \cup (\mathsf{Str} \times \mathsf{Flt})}}{\mathsf{Str} \times \mathsf{Real} \ \leq \ (\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Str}) \cup (\mathsf{Str} \times \mathsf{Flt})}\ .$$

In the very bottom, we use SR-NF, and then SR-UnionL in the level above. To complete the top of derivation, we would also need to use SR-UnionR*, SR-Pair, and SR-BaseRefl.

Theorem 2 (Correctness of Reductive Subtyping)**.**

$$\forall \tau_1, \tau_2.\ (\tau_1 \ \leq_R \ \tau_2 \iff \tau_1 \ \leq \ \tau_2)$$

It is relatively easy to show by induction on a derivation of $\tau_1 \ \leq_R \ \tau_2$ that the reductive subtyping is sound: for each case we build a corresponding derivation of $\tau_1 \ \leq \ \tau_2$. Most of the reductive rules have direct declarative counterparts. In the case of SR-*Num and SR-UnionR*, we need to additionally use transitivity. Finally, in the case of SR-NF, the induction hypothesis gives us $\mathrm{NF}(\tau_1) \ \leq \ \tau_2$, so we can use Lemma 3 and SD-Trans to derive $\tau_1 \ \leq \ \tau_2$.

The challenging part of the proof is to show completeness. For this, we need to prove that the reductive definition is *reflexive*, *transitive*, and *distributive*. To prove transitivity and distributivity, we need several auxiliary statements:

1. $\tau \ \leq_R \ \tau' \implies \mathrm{NF}(\tau) \ \leq_R \ \tau'$,
2. $\mathrm{NF}(\tau_1) \ \leq_R \ \mathrm{NF}(\tau_2) \ \wedge \ \mathrm{NF}(\tau_2) \ \leq_R \ \tau_3$ $\implies \mathrm{NF}(\tau_1) \ \leq_R \ \tau_3$,
3. $\mathrm{NF}(\tau) \ \leq_R \ \mathrm{NF}(\tau') \implies \tau \ \leq_R \ \tau'$.

Having all the facts, we can prove completeness by induction on a derivation of $\mathrm{NF}(\tau_1) \ \leq \ \tau_2$. For details, the reader can refer to the full Coq-proof.

Theorem 3 (Decidability of Reductive Subtyping)**.**

$$\forall \tau_1, \tau_2.\ (\tau_1 \ \leq_R \ \tau_2 \quad \vee \quad \neg \tau_1 \ \leq_R \ \tau_2)$$

The Coq-proof is available.

4 Semantic Subtyping and Multiple Dynamic Dispatch

In this section, we describe in more details how subtyping can be used to implement multiple dynamic dispatch, and also discuss why using semantic subtyping for this might not be a good idea.

As an example, consider the following methods[5] of the addition function (we assume that function `flt` converts its argument to a float):

```
+(x::Int, y::Int) = prim_add_int(x, y)
+(x::Flt, y::Flt) = prim_add_flt(x, y)
+(x::Int∪Flt, y::Int∪Flt) = prim_add_flt(flt(x), ..)
```

and the function call `3 + 5`. How exactly does dispatch work?

One approach, adopted by some languages such as Julia [2], is to use subtyping on tuples [10]. Namely, method signatures and function calls are interpreted as tuple types, and then subtyping is used to determine applicable methods as well as pick one of them. In the example above, the three methods are interpreted as the following types (from top to bottom):

```
mII ≡ Int × Int
mFF ≡ Flt × Flt
mUU ≡ (Int∪Flt) × (Int∪Flt)
```

and the call as having type `cII ≡ Int × Int`. To resolve the call, the language run-time ought to perform two steps.

1. Find applicable methods (if any). For this, we check subtyping between the type of the call, `cII`, and the method signatures. Since `cII <: mII` and `cII <: mUU` but `cII` $\not<$`: mFF`, only two methods are applicable, `mII` for integers and `mUU` for mixed-type numbers.
2. Pick the best, most specific of the applicable methods (if there is one). For this, we check subtyping relation between all applicable methods. In this example, naturally, we would like `mII` to be called for addition of

[5]In the context of MDD, different implementations of the same function are usually called *methods*, and the set of all methods is called a *generic function*.

integers. And indeed, since `mII <: mUU`, the integer addition is considered the most specific.

Let us also consider the call `3.14 + 5`. Its type is `Flt × Int`, and there is only one applicable method that is a supertype of the call type — `mUU`, so it should be called.

It is important to understand what happens if the programmer defines several implementations with the same argument types. In the case of a static language, an error can be reported. In the case of a dynamic language, however, the second implementation simply replaces the earlier one, in the same way as reassignment to a variable replaces its previous value.

For instance, consider a program that contains the three implementations of `(+)` from above and also the following:

```
+(x::Real, y::Real) = ... # mRR
print(3.14 + 5)
```

According to the semantic subtyping relation, type `Real` is equivalent to `Int∪Flt` in MiniJl. Therefore, implementation of `mRR` will replace `mUU`, and the call `3.14 + 5` will be dispatched to `mRR`.

Unfortunately, the semantics of the program above is not stable. If the programmer adds a new type into the nominal hierarchy, e.g. `Int8 <: Real`, type `Real` is not equivalent to `Int∪Flt` anymore. Therefore, if we run the program again, types of `mUU` and `mRR` will be different, and so the implementation of `mRR` will not replace `mUU`. Since `mUU <: mRR`, the call `3.14 + 5` will be dispatched to `mUU`, not `mRR` as before.

We can gain stability by removing the rules that equate abstract nominal types with the union of their subtypes, i.e., SD-RealUnion and SD-NumUnion in the declarative definition.[6] To fix the discrepancy between this definition and the semantic interpretation, we can change the latter by accounting for "future nominal types", e.g. $[\![\mathsf{Real}]\!] = \{\mathsf{Int}, \mathsf{Flt}, X\}$. It needs to be understood whether such an interpretation provides us with a useful intuition about subtyping.

5 Related Work

Semantic subtyping has been studied a lot in the context of *statically typed* languages with *structural* typing. For example, Hosoya and Pierce [8] defined a semantic type system for XML that incorporated unions, products, and recursive types, with a subtyping algorithm based on tree automata [9]. Frisch et al. [7] presented decidable semantic subtyping for a language with functions, products, and boolean combinators (union, intersection, negation). Here, the decision procedure for $\tau_1 <: \tau_2$ is based on checking the emptiness of $\tau_1 \setminus \tau_2$. Dardha et al. [6] adopted semantic subtyping to objects with structural types, and Ancona and Corradi [1] proposed decidable semantic subtyping for mutable records. Unlike these works, we were interested in applying semantic reasoning to a *dynamic* language with *nominal* types.

Though *multiple dispatch* is more often found in dynamic languages, there has been research on safe integration of dynamic dispatch into statically typed languages [3–5, 12]. There, subtyping is used for both static type checking and dynamic method resolution. In the realm of dynamic languages, Bezanson [2] employed subtyping for multiple dynamic dispatch in the Julia language. Julia has a rich language of type annotations, including nominal types, tuples, and unions, and a complex subtyping relation [14]. However, it is not clear whether the subtyping relation is decidable or even transitive, and transitivity of subtyping is important for correct implementation of method resolution. In this paper, we work with only a subset of Julia types, but subtyping on the subset is transitive and decidable.

Recently, a framework for building transitive, distributive, and decidable subtyping of union and intersection types was proposed by Muehlboeck and Tate [11]. Our language of types does not have intersection types but features pair types that distribute over unions in a similar fashion.

6 Conclusion and Future Work

We presented decidable subtyping of nominal types, tuples, and unions, which enjoys the advantages of semantic subtyping, such as simple set-theoretic reasoning, yet can be used in the context of dynamically typed languages. Namely, we interpret types in terms of type tags, which are typical for dynamic languages, and provide a decidable syntactic subtyping relation that is equivalent to the subset relation on the interpretations (aka tag-based semantic subtyping).

We found that the proposed subtyping relation, if used for multiple dynamic dispatch, would make the semantics of dynamically typed programs unstable due to an interaction of abstract nominal types and unions. We expect that a different semantic interpretation of nominal types can fix the issue, and would like to further explore the alternative.

In future work, we plan to extend tag-based semantic subtyping to invariant type constructors, e.g. `Ref`:

$$\begin{aligned} \tau \in \textsc{Type} \quad &::= \quad \dots \mid \mathsf{Ref}[\tau] \\ v \in \textsc{ValType} \quad &::= \quad \dots \mid \mathsf{Ref}[\tau] \end{aligned}$$

As usual for invariant constructors, we would like to consider types such as $\mathsf{Ref}[\mathsf{Int}]$ and $\mathsf{Ref}[\mathsf{Int} \cup \mathsf{Int}]$ to be equivalent because Int and $\mathsf{Int} \cup \mathsf{Int}$ are equivalent. However, a naive interpretation of invariant types is not well defined:

$$[\![\mathsf{Ref}[\tau]]\!] = \{\mathsf{Ref}[\tau'] \mid v \in [\![\tau]\!] \iff v \in [\![\tau']\!]\}.$$

Our plan is to introduce an indexed interpretation,

$$[\![\mathsf{Ref}[\tau]]\!]_{k+1} = \{\mathsf{Ref}[\tau'] \mid v \in [\![\tau]\!]_k \iff v \in [\![\tau']\!]_k\},$$

and define semantic subtyping as:

$$\tau_1 \overset{\text{sem}}{<:} \tau_2 \quad \overset{\text{def}}{\equiv} \quad \forall k.\ ([\![\tau_1]\!]_k \subseteq [\![\tau_2]\!]_k).$$

[6] To get an equivalent reductive subtyping, we need to change the SR-NF rule: replace normalization function NF with $\mathrm{NF}_{\mathrm{at}}$ (Fig. 11, App. A).

References

[1] Davide Ancona and Andrea Corradi. 2016. Semantic Subtyping for Imperative Object-oriented Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 568–587. https://doi.org/10.1145/2983990.2983992

[2] Jeff Bezanson. 2015. Abstraction in technical computing.

[3] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1992. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. ACM, New York, NY, USA, 182–192. https://doi.org/10.1145/141471.141537

[4] Craig Chambers. 1992. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92)*. Springer-Verlag, Berlin, Heidelberg, 33–56. http://dl.acm.org/citation.cfm?id=646150.679216

[5] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 130–145. https://doi.org/10.1145/353171.353181

[6] Ornela Dardha, Daniele Gorla, and Daniele Varacca. 2013. Semantic Subtyping for Objects and Classes. In *Formal Techniques for Distributed Systems*, Dirk Beyer and Michele Boreale (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 66–82.

[7] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages. https://doi.org/10.1145/1391289.1391293

[8] Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Internet Technol.* 3, 2 (May 2003), 117–148. https://doi.org/10.1145/767193.767195

[9] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.* 27, 1 (Jan. 2005), 46–90. https://doi.org/10.1145/1053468.1053470

[10] Gary T. Leavens and Todd D. Millstein. 1998. Multiple Dispatch As Dispatch on Tuples. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 374–387. https://doi.org/10.1145/286936.286977

[11] Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 112 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276482

[12] Gyunghee Park, Jaemin Hong, Guy L. Steele Jr., and Sukyoung Ryu. 2019. Polymorphic Symmetric Multiple Dispatch with Variance. *Proc. ACM Program. Lang.* 3, POPL, Article 11 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290324

[13] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[14] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 113 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276483

$$\frac{}{\mathrm{InNF}(v)}\ \textsc{NF-ValType} \qquad \frac{\mathrm{InNF}(\tau_1) \quad \mathrm{InNF}(\tau_2)}{\mathrm{InNF}(\tau_1 \cup \tau_2)}\ \textsc{NF-Union}$$

Figure 8. Normal form of types in MiniJl

$$\begin{array}{rcl}
\mathrm{NF} : \textsc{Type} & \rightarrow & \textsc{Type} \\
\mathrm{NF}(\mathit{cname}) & = & \mathit{cname} \\
\mathrm{NF}(\mathsf{Real}) & = & \mathsf{Int} \cup \mathsf{Flt} \\
\mathrm{NF}(\mathsf{Num}) & = & \mathsf{Int} \cup \mathsf{Flt} \cup \mathsf{Cmplx} \\
\mathrm{NF}(\tau_1 \times \tau_2) & = & \mathtt{un_prs}(\mathrm{NF}(\tau_1),\ \mathrm{NF}(\tau_2)) \\
\mathrm{NF}(\tau_1 \cup \tau_2) & = & \mathrm{NF}(\tau_1) \cup \mathrm{NF}(\tau_2) \\
\\
\mathtt{un_prs} : \textsc{Type} \times \textsc{Type} & \rightarrow & \textsc{Type} \\
\mathtt{un_prs}(\tau_{11} \cup \tau_{12},\ \tau_2) & = & \mathtt{un_prs}(\tau_{11}, \tau_2) \cup \mathtt{un_prs}(\tau_{12}, \tau_2) \\
\mathtt{un_prs}(\tau_1,\ \tau_{21} \cup \tau_{22}) & = & \mathtt{un_prs}(\tau_1, \tau_{21}) \cup \mathtt{un_prs}(\tau_1, \tau_{22}) \\
\mathtt{un_prs}(\tau_1,\ \tau_2) & = & \tau_1 \times \tau_2
\end{array}$$

Figure 9. Computing normal form of MiniJl types

$$\frac{\textsc{Atom-CName}}{\mathrm{Atom}(\mathit{cname})} \qquad \frac{\textsc{Atom-AName}}{\mathrm{Atom}(\mathit{aname})}$$

$$\frac{\mathrm{Atom}(\tau)}{\mathrm{InNF_{at}}(\tau)}\ \textsc{NFAt-Atom}$$

$$\frac{\mathrm{InNF_{at}}(\tau_1) \quad \mathrm{InNF_{at}}(\tau_2)}{\mathrm{InNF_{at}}(\tau_1 \cup \tau_2)}\ \textsc{AtNF-Union}$$

Figure 10. Atomic normal form of types in MiniJl

$$\begin{array}{rcl}
\mathrm{NF_{at}} : \textsc{Type} & \rightarrow & \textsc{Type} \\
\mathrm{NF_{at}}(\mathit{cname}) & = & \mathit{cname} \\
\mathrm{NF_{at}}(\mathit{aname}) & = & \mathit{aname} \\
\mathrm{NF_{at}}(\tau_1 \times \tau_2) & = & \mathtt{un_prs}(\mathrm{NF_{at}}(\tau_1),\ \mathrm{NF_{at}}(\tau_2)) \\
\mathrm{NF_{at}}(\tau_1 \cup \tau_2) & = & \mathrm{NF_{at}}(\tau_1) \cup \mathrm{NF_{at}}(\tau_2)
\end{array}$$

Figure 11. Computing atomic normal form of MiniJl types

A Appendix: Normal Forms

Fig. 8 defines the predicate $\mathrm{InNF}(\tau)$, which states that type τ is in normal form. Fig. 9 contains the full definition of $\mathrm{NF}(\tau)$ function, which computes the normal form of a type.

Fig. 10 and Fig. 11 present "atomic normal form", which can be used to define reductive subtyping that disables derivations such as $\mathsf{Real} \leq_{\mathrm{R}} \mathsf{Int} \cup \mathsf{Flt}$.