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

Julia Belyakova
Northeastern University
belyakova.y@northeastern.edu

## Abstract

Semantic subtyping enables simple, set-theoretical reasoning about types by interpreting a type as the 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 explore the applicability of semantic subtyping in the context of a *dynamic language with nominal* types. Instead of static type checking, a dynamic language relies on run-time checking of type tags associated with values, so we propose using the tags for semantic interpretation. We base our work on a fragment of the Julia language and present *tag-based semantic* subtyping for nominal types, tuples, and unions, where types are interpreted set-theoretically as sets of type tags. 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 are also discussed.

***Keywords*** semantic subtyping, type tags, multiple dispatch, nominal typing, distributivity, decidability

## 1 Introduction

In static type systems, subtyping is used to determine when a value of one type can be safely used at another type. 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`" [14]. This intuition is not always correct, but, in the case of *semantic subtyping* [1, 8, 9], subtyping is defined exactly as the subset relation. Under semantic subtyping, types are interpreted as sets $[\![\tau]\!] = \{v \mid \vdash v : \tau\}$, and subtyping $\tau_1 <: \tau_2$ is defined as inclusion of the interpretations $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$.

Subtyping can also be used for run-time dispatch of function calls. For example, object-oriented languages usually support single dispatch — the ability to dispatch a method call based on the run-time type of the receiver object. A more complex form of dispatch is *multiple dispatch* (MD) [5, 6], which takes into account run-time types of *all* arguments when dispatching a function call. One way to implement MD is to interpret both function signatures and function calls as tuple types [11] and then use subtyping on these types.

Dynamic dispatch is not limited to statically typed languages, with multiple dispatch being even more widespread among *dynamically* typed ones, e.g., CLOS, Julia, Clojure. Unlike statically typed languages, which conservatively prevent type errors at compile-time, dynamic languages detect type errors at run-time: whenever an operator is restricted to certain kinds of values, the run-time system checks *type tags* associated with the operator's arguments to determine whether it can be safely executed. A type tag indicates the run-time type of a value. Thus, any class that can be instantiated induces a tag — the name of the class, whereas an abstract class or interface does not. Some structural types also give rise to tags, e.g., tuples and sums (tagged unions).

While dynamically typed languages do use subtyping, semantic subtyping is not applicable in this case, for the semantic definition refers to a static typing relation. To enable semantic reasoning in the context of dynamic languages, we propose *tag-based semantic* subtyping where a type is interpreted as a set of run-time type tags instead of values.

We define tag-based semantic subtyping for a fragment of the Julia language that includes nominal types, tuples, and unions. Tuples and unions are rather typical for semantic subtyping systems; they have a clear set-theoretic interpretation and make up an expressive subtyping relation where tuples distributes over unions. At the same time, to the best of our knowledge, the interaction of unions with *nominal* types has not been studied before in the context of semantic subtyping. This interaction introduces an unusual subtyping rule between abstract nominal types and unions, with implications for multiple dispatch. Note that the combination of unions and nominal types is not unique to Julia; for instance, it also appears in the statically typed language Ceylon.

Our contributions are as follows:

1. A definition of tag-based semantic subtyping for nominal types, tuples, and unions (Sec. 2).
2. Two syntactic definitions of subtyping, declarative (Sec. 3.1) and reductive (Sec. 3.2), along with Coq-mechanized proofs that these definitions are equivalent and coincide with the semantic definition (Sec. 4).
3. Proof of decidability of reductive subtyping (App. B).
4. Discussion of the implications of using semantic subtyping for multiple dispatch (Sec. 5).

## 2 Semantic Subtyping in MiniJl

We base our work on a small language of types MiniJl, presented in Fig. 1. Types, denoted by $\tau \in$ Type, include pairs, unions, and nominal types; *cname* denotes *concrete* nominal

, ,
2019.

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

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

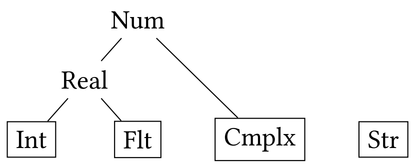


**Figure 1.** MiniJl: type grammar and nominal hierarchy

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

**Figure 2.** Value types

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

**Figure 3.** Tag-based semantic interpretation of types

types that can be instantiated, and *aname* denotes *abstract* nominal types.

We work with a particular hierarchy of nominal types (presented in Fig. 1 as a tree) instead of a generic class table to simplify the development. 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 $n_1 \triangleright n_2$ read as "$n_1$ extends $n_2$" where $n$ is either *cname* or *aname*. In the case of MiniJl, the hierarchy is defined as follows:

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

Nominal hierarchies should not have cycles, and each type can have only one parent.

***Value Types*** Only instantiatable types induce type tags, which we call **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, $\mathsf{Flt}$, $\mathsf{Int} \times \mathsf{Int}$, and $\mathsf{Str} \times (\mathsf{Int} \times \mathsf{Int})$ are all value types. Union types, like abstract nominal types, are not value types. Therefore, a type such as $\mathsf{Int} \cup \mathsf{Int}$ is not a value type despite it describing the same set of values as the value type $\mathsf{Int}$.

### 2.1 Semantic Interpretation of Types

As mentioned in Sec. 1, we interpret a type as a set of type tags (i.e. value types) instead of values and 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.

A type's interpretation states what values constitute the type: $v \in [\![\tau]\!]$ means that values $v$ tagged with $v$ (i.e. instances of $v$) belong to $\tau$. Thus, in MiniJl, a *concrete nominal* type *cname* is comprised only of its direct instances.[1] *Abstract nominal* types cannot be instantiated, but their interpretation needs to reflect the nominal hierarchy. For example, a $\mathsf{Num}$ value is either a concrete complex or real number, which in turn is either a concrete integer or a floating point value. Therefore, the set of value types $\{\mathsf{Cmplx}, \mathsf{Int}, \mathsf{Flt}\}$ describes the set of all possible values of type $\mathsf{Num}$. More generally, the interpretation of an abstract nominal type *aname* can be given as follows:

$$[\![\mathit{aname}]\!] = \{\mathit{cname} \mid \mathit{cname} \triangleright^* \mathit{aname}\},$$

where the relation $n_1 \triangleright^* n_2$ means that nominal type $n_1$ transitively extends $n_2$:

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

Finally, *pairs* and *unions* are interpreted set-theoretically like in standard semantic subtyping.

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

$$\tau_1 \overset{\text{sem}}{<:} \tau_2 \quad \overset{\text{def}}{\equiv} \quad [\![\tau_1]\!] \subseteq [\![\tau_2]\!]. \tag{1}$$

## 3 Syntactic Definitions of Subtyping

While the semantic approach does enable intuitive set-theoretic reasoning about subtyping, a subtyping relation also needs to be computable. However, the semantic definition (1) is not suitable for this purpose. Therefore, we provide an equivalent *syntactic* definition of subtyping that is straightforward to implement.

We do this in two steps. First, we give an inductive *declarative* definition that is handy to reason about and prove it equivalent to the semantic definition. Second, we provide a *reductive* syntax-directed definition of subtyping and prove it equivalent to the declarative one (and, hence, the semantic definition as well). We prove that the reductive subtyping relation is decidable, i.e. for any two types $\tau_1$ and $\tau_2$, it is possible to prove that either $\tau_1$ is a subtype of $\tau_2$ or it is not. The

[1] In the general case, the interpretation of a concrete nominal type would include the type and all its concrete subtypes.

proofs are mechanized in Coq, and since Coq logic is constructive, the decidability proof is also a subtyping algorithm. The algorithm can also be implemented as a straightforward recursive function.

### 3.1 Declarative Subtyping

The declarative syntactic definition of subtyping is provided in Fig. 4. It is mostly comprised of the standard rules of syntactic subtyping for unions and pairs: 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 (they require the left-hand side type to be syntactically equivalent to a part of the right-hand side type), transitivity allows us to derive judgments such as $\mathsf{Int} \leq (\mathsf{Str} \cup \mathsf{Real})$ via $\mathsf{Int} \leq \mathsf{Real}$ and $\mathsf{Real} \leq \mathsf{Str} \cup \mathsf{Real}$.

Note that all rules from Fig. 4 *are essential* for the definition to be equivalent to semantic subtyping. Thus, for example, the syntactic definition needs to be reflexive and transitive because so is the subset relation, which is used to define semantic subtyping. Semantic subtyping also forces us to add rules for distributing pairs over unions, SD-Distr1 and SD-Distr2. For instance, consider two types, $\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})$ and $(\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Flt})$. They have the same semantic interpretation — $\{\mathsf{Str}{\times}\mathsf{Int}, \mathsf{Str}{\times}\mathsf{Flt}\}$ — so they are equivalent. Therefore, we should also be able to derive their equivalence using the declarative definition, i.e. declarative subtyping should hold in both directions. One direction is trivial:

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

But the other direction,

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

cannot be derived without SD-Distr2 rule.

The novel part of the definition resides 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 the rules SD-RealUnion and SD-NumUnion ( highlighted in Fig. 4) are new — dictated by semantic subtyping. Thus, SD-RealUnion allows us to prove the equivalence of types $\mathsf{Int} \cup \mathsf{Flt}$ and $\mathsf{Real}$, which are both interpreted as $\{\mathsf{Int}, \mathsf{Flt}\}$.

### 3.2 Reductive Subtyping

The declarative definition is not syntax-directed. For one, it involves the SD-Trans rule, which requires "coming up" with an intermediate type $\tau_2$. For instance, in order to show

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

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

$$\dfrac{}{\tau \leq \tau}\ \text{SD-Refl} \qquad \dfrac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \text{SD-Trans}$$

$$\dfrac{\text{SD-IntReal}}{\mathsf{Int} \leq \mathsf{Real}} \qquad \dfrac{\text{SD-FltReal}}{\mathsf{Flt} \leq \mathsf{Real}}$$

$$\dfrac{\text{SD-RealNum}}{\mathsf{Real} \leq \mathsf{Num}} \qquad \dfrac{\text{SD-CmplxNum}}{\mathsf{Cmplx} \leq \mathsf{Num}}$$

$$\dfrac{\text{SD-RealUnion}}{\mathsf{Real} \leq \mathsf{Int} \cup \mathsf{Flt}} \qquad \dfrac{\text{SD-NumUnion}}{\mathsf{Num} \leq \mathsf{Real} \cup \mathsf{Cmplx}}$$

$$\dfrac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}\ \text{SD-Pair}$$

$$\dfrac{\tau_1 \leq \tau' \qquad \tau_2 \leq \tau'}{\tau_1 \cup \tau_2 \leq \tau'}\ \text{SD-UnionL}$$

$$\dfrac{\text{SD-UnionR1}}{\tau_1 \leq \tau_1 \cup \tau_2} \qquad \dfrac{\text{SD-UnionR2}}{\tau_2 \leq \tau_1 \cup \tau_2}$$

$$\dfrac{}{(\tau_{11} \cup \tau_{12}) \times \tau_2 \leq (\tau_{11} \times \tau_2) \cup (\tau_{12} \times \tau_2)}\ \text{SD-Distr1}$$

$$\dfrac{}{\tau_1 \times (\tau_{21} \cup \tau_{22}) \leq (\tau_1 \times \tau_{21}) \cup (\tau_1 \times \tau_{22})}\ \text{SD-Distr2}$$

**Figure 4.** Declarative subtyping for MiniJl

The syntax-directed reductive definition[2] of subtyping is presented in Fig. 5. Some of the inductive rules have the exact declarative counterparts, such as the rule for subtyping of pairs (SR-Pair) or the rule for a union on the left (SR-UnionL). The differing rules are highlighted . The explicit reflexivity rule SR-BaseRefl now only works with concrete nominal types, which is enough for the definition to be reflexive. General transitivity is gone as well; instead, it gets incorporated into other rules, such as subtyping of nominal types (SR-IntNum, SR-FltNum) or subtyping of a union on the right (SR-UnionR1, SR-UnionR2).

The last rule of the definition, SR-NF, is the most important. It rewrites type $\tau$ into its **normal form** $\mathrm{NF}(\tau)$ before applying other subtyping rules. This covers all useful applications of transitivity and distributivity that are possible in the declarative definition. The normalized type has the form $v_1 \cup v_2 \cup \ldots \cup v_n$, i.e. a union of value types (we omit parenthesis because union is associative). The normalization

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

$$\frac{}{\mathit{cname}\ \le_R\ \mathit{cname}}\ \text{SR-BaseRefl}$$

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

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

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

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

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

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

**Figure 5.** Reductive subtyping for MiniJl

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

**Figure 6.** Computing normal form of MiniJl types

function NF is presented in Fig. 6 (the auxiliary function `un_prs` can be found in Fig. 9, App. A). It produces a type in *disjunctive normal form* by replacing an abstract nominal type with the union of all its concrete subtypes, a pair of unions with the union of pairs of value types (each of this pairs is a value type), for instance:

$$\mathrm{NF}(\mathsf{Str} \times (\mathsf{Int} \cup \mathsf{Flt})) = (\mathsf{Str} \times \mathsf{Int}) \cup (\mathsf{Str} \times \mathsf{Flt}).$$

As we show in Sec. 4.1, a type and its normal form are equivalent in the declarative definition, an essential property for reductive subtyping being equivalent to declarative one.

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

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

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

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

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

**Figure 7.** Matching relation in MiniJl

# 4 Properties of Subtyping Relations

## 4.1 Correctness of Declarative Subtyping

We need to prove that the declarative definition of subtyping is sound and complete with respect to the semantic definition in order to show correctness of declarative subtyping. Formally, we write this as:

$$\forall \tau_1, \tau_2.\ (\tau_1\ \le\ \tau_2 \iff \tau_1 \overset{\text{sem}}{<:} \tau_2). \tag{2}$$

Instead of directly proving (2), it is more convenient to prove the equivalence of declarative subtyping to the following relation (referred to as **matching-based semantic subtyping**):

$$\tau_1\ <:\ \tau_2 \quad \overset{\text{def}}{\equiv} \quad \forall v.\ (v \prec \tau_1 \implies v \prec \tau_2). \tag{3}$$

The definition (3) relies on the relation $v \prec \tau$ (defined in Fig. 7), read "tag $v$ matches type $\tau$", which we call the **matching relation**.

Tag-based and matching-based semantic subtyping relations are equivalent:

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

To see why, recall that tag-based semantic subtyping (1) is defined as $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$ and the subset relation $X \subseteq Y$ as $\forall x.\ (x \in X \implies x \in Y)$. Therefore, the definition (1) can be rewritten as:

$$\tau_1 \overset{\text{sem}}{<:} \tau_2 \overset{\text{def}}{\equiv} \forall v.\ (v \in [\![\tau_1]\!] \implies v \in [\![\tau_2]\!]). \tag{4}$$

It is easy to show by induction on $\tau$ that the matching relation is equivalent to the belongs-to relation $v \in [\![\tau]\!]$. Therefore, the definitions (3) and (4) are also equivalent.

Since $\tau_1 \overset{\text{sem}}{<:} \tau_2$ is equivalent to $\tau_1\ <:\ \tau_2$ and the equivalence relation $\iff$ is transitive, it suffices to prove the following theorem to show (2).

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

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

The full proof of Theorem 1 is Coq-mechanized [2], so we only discuss some key aspects and leave details to the proof. First, subtyping a value type coincides with matching:

$$\forall v, \tau.\, (v \leq \tau \iff v \prec \tau). \tag{5}$$

Having that, we can prove $\tau_1 \leq \tau_2 \implies \tau_1 <: \tau_2$, i.e. the soundness direction of Theorem 1 (below, we embed the definition (3) of matching-based semantic subtyping):

$$\forall \tau_1, \tau_2.\, (\tau_1 \leq \tau_2 \implies \forall v.\, [v \prec \tau_1 \implies v \prec \tau_2]). \tag{6}$$

Knowing $\tau_1 \leq \tau_2$ and $v \prec \tau_1$, we need to show that $v \prec \tau_2$. First, by applying (5) to $v \prec \tau_1$, we get $v \leq \tau_1$. Then, $v \leq \tau_2$ follows from $v \leq \tau_1$ and $\tau_1 \leq \tau_2$ by transitivity. Finally, by applying (5) again, we get $v \prec \tau_2$. □

The other direction of Theorem 1 is more challenging:

$$\forall \tau_1, \tau_2.\, (\tau_1 <: \tau_2 \implies \tau_1 \leq \tau_2). \tag{7}$$

The key observation here is that (7) can be shown for $\tau_1$ in *normal form*, i.e. $\tau_1 \equiv v_1 \cup v_2 \cup \ldots \cup v_n$ (formally, this fact is denoted by predicate InNF($\tau_1$) defined in Fig. 8, App. A):

$$\forall \tau_1, \tau_2 \mid \mathrm{InNF}(\tau_1).\, (\tau_1 <: \tau_2 \implies \tau_1 \leq \tau_2). \tag{8}$$

In this case, in the definition (3) of $\tau_1 <: \tau_2$, the only value types $v$ that match $\tau_1$ and $\tau_2$ are $v_i$ of $\tau_1$. By (5), we know that matching implies subtyping, so we have that all $v_i \leq \tau_2$. From the latter, it is easy to show that $(v_1 \cup v_2 \cup \ldots \cup v_n) \leq \tau_2$ because, according to the SD-UnionL rule, subtyping of the left-hand side union amounts to subtyping its components. To show (7), we need several more facts in addition to (8).

- Function NF produces a type in normal form:

$$\forall \tau.\ \mathrm{InNF}(\mathrm{NF}(\tau)). \tag{9}$$

- Normalized type is equivalent to the source type:

$$\forall \tau.\ \mathrm{NF}(\tau) \leq \tau \wedge \tau \leq \mathrm{NF}(\tau). \tag{10}$$

- Normalization preserves subtyping relation:

$$\forall \tau_1, \tau_2.\, (\tau_1 <: \tau_2 \implies \mathrm{NF}(\tau_1) <: \tau_2). \tag{11}$$

To prove (7), we need to show $\tau_1 \leq \tau_2$ given $\tau_1 <: \tau_2$. For this, we first apply (11) to $\tau_1 <: \tau_2$, which gives $\mathrm{NF}(\tau_1) <: \tau_2$. Then we can apply (8) to the latter because of (9) to get $\mathrm{NF}(\tau_1) \leq \tau_2$. Finally, (10) and transitivity gives $\tau_1 \leq \tau_2$. □

### 4.2 Reductive Subtyping

Since we have already shown that declarative subtyping is equivalent to semantic subtyping, it suffices to show that reductive subtyping is equivalent to declarative subtyping:

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

$$\forall \tau_1, \tau_2.\, (\tau_1 \leq_{\mathrm{R}} \tau_2 \iff \tau_1 \leq \tau_2)$$

The proof is split into two parts: soundness and completeness. For soundness (completeness), we show that for each SR- rule (SD- rule) it is possible to build a corresponding declarative (reductive) derivation using SD-rules (SR-rules).

The soundness direction is mostly straightforward, as most SR- rules have an immediate SD- counterpart (or require one extra application of transitivity). In the case of SR-NF, the induction hypothesis of the proof, $\mathrm{NF}(\tau_1) \leq \tau_2$, and the fact that $\tau_1 \leq \mathrm{NF}(\tau_1)$ (10) allow to conclude $\tau_1 \leq \tau_2$.

The challenging part of the proof is to show completeness, as this requires proving that the reductive definition is *reflexive*, *transitive*, and *distributive* (App. B).

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

$$\forall \tau_1, \tau_2.\, (\tau_1 \leq_{\mathrm{R}} \tau_2 \quad \vee \quad \neg[\tau_1 \leq_{\mathrm{R}} \tau_2])$$

To prove the theorem, it suffices to show that reductive subtyping is decidable when $\tau_1$ is in normal form. This is done by induction on a derivation of InNF($\tau_1$). We refer the reader to App. B for more details.

## 5 Semantic Subtyping and Multiple Dynamic Dispatch

We set out to define semantic subtyping that can be useful in the context of dynamic languages, however, it appears that the semantic definition we presented has an undesired implication for dynamic dispatch. In this section, we discuss the implication and suggest a solution, using multiple dispatch as a running example.

Consider the following methods[3] of the addition function defined in a Julia syntax (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 [3], is to use subtyping on tuple types [11]. 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 the 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`

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

but `cII` $\not<:$ `mFF`, only two methods are applicable, `mII` for integers and `mUU` for mixed-type numbers.

2. Pick the most specific of the applicable methods (if one exists). For this, we check subtyping relations between all applicable methods. In this example, naturally, we would like `mII` to be called for the addition of integers. And indeed, since `mII <: mUU`, the integer addition is considered the most specific.

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

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 previous implementations of `(+)` and also:

```
+(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` defined earlier, and the call `3.14 + 5` will be dispatched to `mRR`.

There is a problem: due to the use of semantic subtyping, the semantics of the program above can change if the programmer adds a new type into the nominal hierarchy. For instance, if they add `Int8 <: Real`, type `Real` stops being equivalent to `Int∪Flt`. Therefore, if the program is re-run, the types of `mUU` and `mRR` will be different and the implementation of `mRR` will not replace `mUU`. Since `mUU <: mRR`, the call `3.14 + 5` will be dispatched to `mUU`, not `mRR`.

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.[4] To fix the discrepancy between this definition and the semantic interpretation, we can change the latter by accounting for "future nominal types", e.g. $[\![\mathsf{Real}]\!] = \{\mathsf{Int}, \mathsf{Flt}, X\}$.

## 6 Related Work

Semantic subtyping has been studied primarily in the context of *statically typed* languages with *structural* typing. For example, Hosoya and Pierce [9] defined a semantic type system for XML that incorporates unions, products, and recursive types, with a subtyping algorithm based on tree automata [10]. Frisch et al. [8] presented decidable semantic subtyping for a language with functions, products, and boolean combinators (union, intersection, negation); the decision procedure for $\tau_1 <: \tau_2$ is based on checking the emptiness of $\tau_1 \setminus \tau_2$.

[4] 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).

Dardha et al. [7] adopted semantic subtyping to objects with structural types, and Ancona and Corradi [1] proposed decidable semantic subtyping for mutable records. Unlike these works, we are 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 [4–6, 13]. There, subtyping is used for both static type checking and dynamic method resolution. In the realm of dynamic languages, Bezanson [3] employed subtyping for multiple dynamic dispatch in the Julia language. Julia has a rich language of type annotations (including, but not limited to nominal types, tuples, and unions) and a complex subtyping relation [15]. 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, while we work with only a subset of Julia types, subtyping 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 [12]. Our language of types does not have intersection types but features pair types that distribute over unions in a similar fashion.

## 7 Conclusion and Future Work

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

We found that the proposed subtyping relation, if used for 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 top and bottom types, and also invariant type constructors, e.g. $\mathsf{Ref}[\tau]$. 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. However, a naive interpretation of invariant types below 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:

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

# 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] Julia Belyakova. 2018. Coq mechanization of MiniJl. https://github.com/julbinb/ftfjp-2019/tree/master/Mechanization

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

[4] 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

[5] 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

[6] 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

[7] 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.

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

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

[15] 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

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

**Figure 8.** Normal form of types in MiniJl

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

**Figure 9.** Computing normal form of MiniJl types

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

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

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

**Figure 10.** Atomic normal form of types in MiniJl

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

**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 $\tau$ 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 $\mathsf{Real} \le \mathsf{Int} \cup \mathsf{Flt}$.

# B Appendix: Overview of Coq Proofs

In this section we give a brief overview of the Coq-mechanization [2] of the paper. When referring to a file `fname`, we mean the file `Mechanization/fname` in [2].

## B.1 Definitions

Most of the relevant definitions are in `MiniJl/BaseDefs.v`. In the table below, we show the correspondence between paper definitions (left column) and Coq definitions (middle column), possibly with syntactic sugar (right column).

| | | |
|---|---|---|
| Types | | |
| $\tau$ | `ty` | |
| $v$ | `value_type v` | |
| Relations | | |
| $v <: \tau$ | `match_ty v t` | `\|- v <$ t` |
| $\tau_1\ <:\ \tau_2$ | `sem_sub t1 t2` | `\|\|- [t1] <= [t2]` |
| $\tau_1\ \le\ \tau_2$ | `sub_d t1 t2` | `\|- t1 << t2` |
| $\tau_1\ \le_R\ \tau_2$ | `sub_r t1 t2` | `\|- t1 << t2` |
| Auxiliary definitions | | |
| $\mathrm{InNF}(\tau)$ | `in_nf t` | `InNF(t)` |
| $\mathrm{NF}(\tau)$ | `mk_nf t` | `MkNF(t)` |
| $\mathtt{un\_prs}(\tau_1, \tau_2)$ | `unite_pairs t1 t2` | |

## B.2 Basic Properties of Normalization Function

File `MiniJl/BaseProps.v` contains several simple properties that are needed for proving the major theorems discussed in the paper, in particular, the following properties of the normalization function NF:

| Statement | Ref in text | Name in Coq |
|---|---|---|
| $\mathrm{InNF}(\mathrm{NF}(\tau))$ | (9) | `mk_nf__in_nf` |
| $\mathrm{InNF}(\tau) \implies (\mathrm{NF}(\tau) \equiv \tau)$ | | `mk_nf_nf__equal` |
| $\mathrm{NF}(\mathrm{NF}(\tau)) \equiv \mathrm{NF}(\tau)$ | | `mk_nf__idempotent` |

## B.3 Basic Properties of Matching Relation

The following properties are proven in `MiniJl/PropsMatch.v`.

- Matching relation is *reflexive*, `match_valty__rflxv` (by induction on $v$):
$$\forall v.\ v \lessdot v.$$
- The only value type that a value type matches is the value type itself, `valty_match_valty__equal` (by induction on $v_1 \lessdot v_2$):
$$\forall v_1, v_2.\ (v_1 \lessdot v_2 \implies v_1 \equiv v_2).$$
- The matching relation is *decidable*, `match_ty__dcdbl` (by induction on $v$, then by induction on $\tau$):
$$\forall v, \tau.\ (v \lessdot \tau\ \vee\ \neg[v \lessdot \tau]).$$

## B.4 Correctness of Declarative Subtyping

First, we discuss some auxiliary statements that are needed for proving Theorem 1 (located in `MiniJl/DeclSubProp.v`).

One direction of (5),

$$\forall v, \tau.\ (v \prec \tau \implies v \le \tau), \tag{12}$$

is proven in `match_ty__sub_d_sound` by induction on $v \prec \tau$. The other direction,

$$\forall v, \tau.\ (v \le \tau \implies v \prec \tau),$$

is proven in `match_valty__sub_d_complete` by induction on $v \le \tau$. The transitivity case, SD-Trans, requires a helper statement, `match_valty__transitive_on_sub_d`:

$$\forall \tau_1, \tau_2, v.\ (\tau_1 \le \tau_2 \wedge v \prec \tau_1 \implies v \prec \tau_2), \tag{13}$$

which is proven by induction on $\tau_1 \le \tau_2$.

The equivalence of a type and its normal form (10) is shown by induction on $\tau$ in lemmas `mk_nf__sub_d1` ($\mathrm{NF}(\tau) \le \tau$) and `mk_nf__sub_d2` ($\tau \le \mathrm{NF}(\tau)$).

Semantic completeness of declarative subtyping for a normalized type (8),

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

is shown in `nf_sem_sub__sub_d` by induction on $\mathrm{InNF}(\tau_1)$. When $\tau_1 \equiv v$, we use (12). By definition of $v <: \tau_2$, we know that $v \prec \tau_2$ follows from $v \prec v$.
When $\tau_1 \equiv \tau_a \cup \tau_b$, we use induction hypothesis $\tau_a \le \tau_2$ and $\tau_b \le \tau_2$, SD-UnionL rule, and the fact that

$$\forall v, \tau_1, \tau_2.\ (v \prec \tau_i \implies v \prec \tau_1 \cup \tau_2).$$

Finally, soundness and completeness parts of Theorem 1 (`sub_d__semantic_sound` and `sub_d__semantic_complete`) are proven in `MiniJl/Props.v`. Note that soundness (6) is the same as transitivity of the matching relation (13). The completeness part (7) is proven as explained at the end of Sec. 4.1.

### B.5 Correctness of Reductive Subtyping

As discussed in Sec. 4.2, the soundness part of Theorem 2 (lemma `sub_r__sound` in `MiniJl/Props.v`),

$$\forall \tau_1, \tau_2.\ (\tau_1 \le_R \tau_2 \implies \tau_1 \le \tau_2),$$

is proven by induction on $\tau_1 \le_R \tau_2$. The only interesting case is the rule SR-NF where we have the induction hypothesis $\mathrm{NF}(\tau_1) \le \tau_2$ and need to show $\tau_1 \le \tau_2$. Since $\tau_1 \le \mathrm{NF}(\tau_1)$, we can use transitivity (rule SD-Trans).

The completeness part of Theorem 2 (lemma `sub_r__complete` in `MiniJl/Props.v`),

$$\forall \tau_1, \tau_2.\ (\tau_1 \le \tau_2 \implies \tau_1 \le_R \tau_2),$$

is ultimately proven by induction on $\tau_1 \le \tau_2$. However, the proof requires showing that reductive subtyping satisfies the following properties (defined in `MiniJl/RedSubProps.v`):

- *Reflexivity*, `sub_r__reflexive` (by induction on $\tau$):

$$\forall \tau.\ \tau \le_R \tau.$$

- *Transitivity*, `sub_r__transitive`:

$$\forall \tau_1, \tau_2, \tau_3.\ (\tau_1 \le_R \tau_2 \wedge \tau_2 \le_R \tau_3 \implies \tau_1 \le_R \tau_3).$$

- *Distributivity* of pairs over unions:

$$(\tau_{11} \cup \tau_{12}) \times \tau_2 \le_R (\tau_{11} \times \tau_2) \cup (\tau_{12} \times \tau_2)$$

and

$$\tau_1 \times (\tau_{21} \cup \tau_{22}) \le_R (\tau_1 \times \tau_{21}) \cup (\tau_1 \times \tau_{22}).$$

The transitivity proof is done by induction on $\tau_1 \le_R \tau_2$. In some cases it relies on the fact that subtyping a type is the same as subtyping its normal form,

$$\forall \tau.\ (\tau_1 \le_R \tau_2 \iff \mathrm{NF}(\tau_1) \le_R \tau_2). \tag{14}$$

The right-to-left part follows from SR-NF, and the left-to-right is shown by induction on $\tau_1 \le_R \tau_2$ (`sub_r__mk_nf_sub_r1`). In the SR-Pair case of the transitivity proof, we also need to perform induction on $\tau_2 \le_R \tau_3$. The last case, SR-NF, uses the two auxiliary facts:

$$\forall \tau_1, \tau_2.\ (\tau_1 \le_R \tau_2 \implies \mathrm{NF}(\tau_1) \le_R \mathrm{NF}(\tau_2)),$$

proven in `sub_r__mk_nf_sub_r` by induction on $\tau_1 \le_R \tau_2$ (uses the idempotence of NF), and $\forall \tau_1, \tau_2, \tau_3$.

$$\mathrm{InNF}(\tau_1) \wedge \mathrm{InNF}(\tau_2) \wedge (\tau_1 \le_R \tau_2) \wedge (\tau_2 \le_R \tau_3) \implies \tau_1 \le_R \tau_3,$$

proven in `sub_r_nf__transitive` by induction on $\tau_1 \le_R \tau_2$.

The distributivity proofs use the fact that

$$\forall \tau_1, \tau_2.\ (\mathrm{NF}(\tau_1) \le_R \mathrm{NF}(\tau_2) \implies \tau_1 \le_R \tau_2),$$

proven in `mk_nf_sub_r__sub_r`, and that normal forms of both types in SD-Distr* rules are in the subtyping relation:

$$\mathrm{NF}((\tau_{11} \cup \tau_{12}) \times \tau_2) \le_R \mathrm{NF}((\tau_{11} \times \tau_2) \cup (\tau_{12} \times \tau_2))$$

(`mk_nf__distr11`) and

$$\mathrm{NF}(\tau_1 \times (\tau_{21} \cup \tau_{22})) \le_R \mathrm{NF}((\tau_1 \times \tau_{21}) \cup (\tau_1 \times \tau_{22}))$$

(`mk_nf__distr21`).

### B.6 Decidability of Reductive Subtyping

The proof of Theorem 3,

$$\forall \tau_1, \tau_2.\ (\tau_1 \le_R \tau_2 \quad \vee \quad \neg[\tau_1 \le_R \tau_2]),$$

is given by `sub_r__decidable` in `MiniJl/Props.v`. It relies on the fact (discussed below) that reductive subtyping is decidable for $\tau_1$ s.t. $\mathrm{InNF}(\tau_1)$.

- Namely, if $\mathrm{NF}(\tau_1) \le_R \tau_2$, then $\tau_1 \le_R \tau_2$ by SR-NF.
- Otherwise, if $\neg[\mathrm{NF}(\tau_1) \le_R \tau_2]$, which in Coq means $\mathrm{NF}(\tau_1) \le_R \tau_2 \implies$ `False`, we can show $\neg[\tau_1 \le_R \tau_2]$ by assuming that $\tau_1 \le_R \tau_2$, applying (14) to it, and thus getting contradiction.

Decidability of subtyping of a normalized type,

$$\forall \tau_1, \tau_2 \mid \mathrm{InNF}(\tau_1).\ (\tau_1 \le_R \tau_2 \quad \vee \quad \neg[\tau_1 \le_R \tau_2]),$$

(lemma `nf_sub_r__decidable` in `MiniJl/RedSubProps.v`) is proven by induction on $\mathrm{InNF}(\tau_1)$ and uses the decidability of the matching relation, which coincides with reductive subtyping on a value type.