On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types

Jay Ligatti, University of South Florida
Jeremy Blackburn, University of South Florida
Michael Nachtigal, University of South Florida

Well-known techniques exist for proving the soundness of subtyping relations with respect to type safety. However, completeness has not been treated with widely applicable techniques, as far as we’re aware.

This paper develops some techniques for stating and proving that a subtyping relation is complete with respect to type safety and applies the techniques to the study of iso-recursive subtyping.

The common subtyping rules for iso-recursive types—the “Amber rules”—are shown to be incomplete with respect to type safety. That is, there exist iso-recursive types \( \tau_1 \) and \( \tau_2 \) such that \( \tau_1 \) can safely be considered a subtype of \( \tau_2 \), but \( \tau_1 \leq \tau_2 \) is not derivable with the Amber rules.

This paper defines new, algorithmic rules for subtyping iso-recursive types and proves that the rules are sound and complete with respect to type safety. The fully implemented subtyping algorithm is optimized to run in \( O(mn) \) time, where \( m \) is the number of \( \mu \)-terms in the types being considered and \( n \) is the size of the types being considered.

Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics; D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms: Languages, Algorithms

Additional Key Words and Phrases: Subtyping, Completeness, Preciseness, Recursive types

1. INTRODUCTION

When defining a subtyping relation for a type-safe language, one takes into account both the soundness and the completeness of the subtyping relation with respect to type safety. Soundness alone can be satisfied by making the subtyping relation the least reflexive and transitive relation over types (i.e., \( \tau_1 \) is a subtype of \( \tau_2 \) if and only if \( \tau_1 = \tau_2 \)); completeness alone can be satisfied by making the subtyping relation the greatest reflexive and transitive relation over types (i.e., all types are subtypes of all other types). These extremes rather defeat the purpose of subtyping, which may be thought of as allowing terms of one type to stand in for terms of another type when it would be safe to do so. A standard strategy for defining a subtyping relation would be to aim for the most complete definition possible without sacrificing soundness.

Despite the importance of both soundness and completeness, completeness has not been treated as widely as soundness. Well-known techniques exist for proving the soundness of subtyping relations with respect to type safety. Standard type-safety...
proofs in languages with subtyping prove the soundness of the languages’ subtyping relations; an unsound subtyping relation would break type safety by statically allowing (via a subsumption rule in the type system) terms of some type $\tau_1$ to stand in for terms of another type $\tau_2$, when operations could be performed on $\tau_2$-type terms that aren’t defined for $\tau_1$-type terms, potentially leading to dynamically “stuck” states.

This paper develops some techniques for stating and proving that a subtyping relation is complete with respect to type safety and applies the techniques to the problem of subtyping recursive types, in particular, iso-recursive types.

Recursive types, along with product and sum types, are fundamental for typing aggregate data structures. A standard example of a recursive type would be a natural-number-list type $L \equiv \mu t. (\text{unit} + (\text{nat} \times t))$. The type variable $t$ refers to the nat-list type $(L)$ being defined. Lists of natural numbers according to this definition could be empty (i.e., have type unit) or could be a natural number (the list head) paired with another list (the tail).

Iso-recursive (also called weakly recursive) types require programmers to manually roll and unroll (also called fold and unfold) recursive types. Unrolling converts a term of type $\mu t. \tau$ to a term of type $[\mu t. \tau/t] \tau$, while rolling performs the inverse conversion (where $[\tau/t] \tau'$ is the capture-avoiding substitution of $\tau$ for $t$ in $\tau'$). For example, a programmer could create a value of type $L$ defined above by writing $\text{roll}(\text{inl} (\text{unit} + (\text{nat} \times L)))$; the inl value has type unit + (nat × L), so rolling it produces a value of type $L$. Although type checkers in languages with equi-recursive (also called strongly recursive) types automatically roll and unroll terms as needed, so programmers don’t have to, practical programming languages that support iso-recursive types, such as ML and Haskell, ease the burden of rolling and unrolling by merging this syntax with other syntax. For example, a programmer might define an iso-recursive type for natural-number lists with

$$\text{rectype } t = \text{nil of unit + cons of nat }\times\ t$$

and then just write the constructor $\text{nil}()$ to mean $\text{roll}(\text{inl} (\text{unit} + (\text{nat} \times L)))$. Hence, in practice programmers don’t explicitly roll and unroll iso-recursive types; these operations occur implicitly and automatically during constructors (rolling) and pattern matching (unrolling).

1.1. Related Work

Research into subtyping completeness has focused on proving subtyping algorithms complete with respect to definitions of subtyping relations (e.g., [Colazzo and Ghelli 2005; Pierce 1991; Hosoya et al. 1998; Tate et al. 2011]). Sekiguchi and Yonezawa also proved a type-inference algorithm sound and complete in the presence of subtyped recursive types [Sekiguchi and Yonezawa 1994].

This paper approaches subtyping from a type-safety perspective, investigating the greatest subtyping relation possible without violating type safety; however, other notions of when one type can or should be a subtype of another may be preferred in other contexts. For example, subtyping may be based on particular behaviors of objects in OOPs [Liskov and Wing 1994; Pierik and Boer 2005]. Another possibility is to consider the denotation of a type $\tau$ to be the set of terms of type $\tau$; then a subtyping relation $\leq$ is sound when $\tau_1 \leq \tau_2 \Rightarrow [\tau_1] \subseteq [\tau_2]$ and complete when $[\tau_1] \subseteq [\tau_2] \Rightarrow \tau_1 \leq \tau_2$ [Vouillon 2004]. Using these definitions, Vouillon has shown that the standard subtyping rules for function, union, and intersection types are sound and complete (under some assumptions but overall for a broad class of languages) [Vouillon 2004]. In contrast with these other approaches to subtyping, soundness and completeness in this paper are structural properties that, like normal type safety, specify relationships between languages’ static and (here, SOS-style [Plotkin 2004]) dynamic semantics.
The research on subtyping recursive types seems to have focused more on equi-recursive than iso-recursive systems. For example, Amadio and Cardelli presented rules and an algorithm for subtyping equi-recursive types [Amadio and Cardelli 1993]. The rules and algorithm are proved sound and complete with respect to type trees that result from “infinitely unrolling” equi-recursive types (i.e., the rules and algorithm determine $\tau_1 \leq \tau_2$ precisely when the type obtained by infinitely unrolling $\tau_1$ is a subtype of the type obtained by infinitely unrolling $\tau_2$). Other papers have since refined equi-recursive subtyping analyses and algorithms (e.g., [Kozen et al. 1995; Brandt and Henglein 1998; Gapeyev et al. 2002; Gauthier and Pottier 2004; Stone and Schoonmaker 2005; Colazzo and Ghelli 2005]).

For subtyping iso-recursive types, the most commonly used rules are the Amber rules:

$$S \cup \{t \leq t'\} \vdash \tau \leq \tau' \quad \text{AMBER1}$$

$$S \vdash \mu t.\tau \leq \mu t'.\tau' \quad \text{AMBER2}$$

A “recursive type $\text{rec}(t)T$ is included in a recursive type $\text{rec}(u)U$, if assuming $t$ included in $u$ implies $T$ included in $U$” [Cardelli 1986]. Note that the Amber rules assume that type variables are uniquely named, through alpha-conversion if necessary.

The Amber rules (or less-complete versions of the Amber rules tailored to specific domains, e.g., [Backes et al. 2011]) are the standard approach to defining iso-recursive subtyping (e.g., [Pierce 2002; Harper 2013; Cook et al. 1989; Simons 1994; Hosoya et al. 1998; Simons 2002; Bengtson et al. 2011]).

### 1.2. Overview and List of Contributions

Section 2 formalizes what it means for a subtyping relation to be sound, complete, and precise with respect to type safety. Intuitively, a precise (i.e., sound and complete) subtyping relation specifies that $\tau_1$ is a subtype of $\tau_2$ if and only if terms of type $\tau_1$ can always stand in for terms of type $\tau_2$ without compromising type safety. Section 2 uses evaluation contexts to formalize this intuition.

Section 3 proves that the standard subtyping system for a simply typed lambda calculus is precise with respect to type safety. The proof’s layout and techniques are rather general, so we expect them to be helpful for proving the preciseness of other inductively defined subtyping relations. Moreover, the proof of completeness uses a new (as far as we’re aware) technique, induction over failing derivations [Ligatti 2013a].

With the paper’s primary contributions completed in Sections 2 and 3, Sections 4 and 5 move to the secondary contributions, which involve applying the new definitions and techniques to the study of iso-recursive types.

Section 4 shows that the Amber rules are incomplete for subtyping iso-recursive types. First, they violate reflexivity in some ways; they can’t derive that $\mu t.(t \rightarrow \text{nat})$ is a subtype of itself, due to complications with subtyping in contravariant positions. Second, due to complications with type unrolling, they can’t derive that types like $\mu a.((\mu b.((b \rightarrow \text{nat}) + a)) + \text{nat}) + a$ are subtypes of types like $\mu c.((c + \text{real}) + c)$, though it’s always safe for expressions of the former type to stand in for expressions of the latter type.

Given the incompleteness of the Amber rules, Section 5 presents new subtyping rules for iso-recursive types and proves them precise with respect to type safety. As far as we’re aware, this is the first proof that iso-recursive subtyping rules are in some way complete. The main finding here is that, for the sake of completeness (and reflexivity),
the following rules can be used:

\[
S \cup \{\mu.\tau \leq \mu'.\tau'\} \vdash [\mu.\tau/t]\tau \leq [\mu'.\tau'/t']\tau' \\
S \vdash \mu.\tau \leq \mu'.\tau' \quad S-REC1
\]

\[
S \cup \{\mu.\tau \leq \mu'.\tau'\} \vdash \mu.\tau \leq \mu'.\tau' \quad S-REC2
\]

These new rules simultaneously unroll the iso-recursive types under consideration, matching the types obtained when recursive-type values are eliminated (using unroll expressions).

Section 5 also presents a deterministic algorithm for subtyping iso-recursive types and shows that the algorithm runs in \(O(mn)\) time, where \(m\) is the number of \(\mu\)-terms in the types being considered and \(n\) is the size of the types being considered. Because the \(m\) variable is independent from, and guaranteed to be smaller than, the \(n\) variable, the \(O(mn)\) bound is an improvement over the best-known bound of \(O(n^2)\) for subtyping equi-recursive types.

Section 6 contains further discussion of this paper’s definitions and results.

2. BASIC DEFINITIONS

This section formalizes the soundness, completeness, and preciseness of subtyping relations, with respect to type safety.

Intuitively, we wish for a language’s subtyping relation to define \(\tau_1 \leq \tau_2\) precisely when such a definition could not compromise type safety. By the principle of subsumption, which states that a term of type \(\tau\) when such a definition could not compromise type safety. By the principle of subsumption with respect to type safety.

This section formalizes the soundness, completeness, and preciseness of subtyping relations. The following definition formalizes this requirement that \(\tau_1 \leq \tau_2\) if and only if \(\tau_2\)-type expressions can—in any context—be replaced by \(\tau_1\)-type expressions without causing well-typed programs to “get stuck.” The definition assumes typing judgments of the form \(e:\tau\) and SOS-style single- and multi-step judgments \(e \mapsto e'\) and \(e \mapsto^* e'\), with the usual meanings. The definition also uses evaluation contexts in the standard way; an evaluation context is an expression with a “hole” that can be filled by a subexpression. The judgment form \(E[\tau]:\tau\) means that filling evaluation context \(E\)'s hole with a \(\tau\)-type expression produces a \(\tau\)-type expression (formally, \(E[\tau]:\tau \iff \{x:\tau\} \vdash E[x]:\tau\), where \(x\) is a “fresh” variable, not appearing in \(E\)).

Definition 1 (Preciseness, Soundness, and Completeness). Let metavariables \(E, e,\) and \(\tau\) respectively range over evaluation contexts, expressions, and types. Then a subtyping relation \(\leq\) (i.e., a reflexive, transitive, binary relation on types) is precise with respect to type safety when, for all types \(\tau_1\) and \(\tau_2\):

\[
\tau_1 \leq \tau_2 \iff \neg \exists E, \tau, e, e' : E[\tau_2]:\tau \land e:\tau_1 \land E[e] \mapsto^* e' \land \text{stuck}(e')
\]

When the only-if direction \(\Rightarrow\) of this formula holds, we say that the subtyping relation is sound with respect to type safety; when the if direction \(\Leftarrow\) holds, we say that the subtyping relation is complete with respect to type safety.

Definition 1 stipulates that precise subtyping relations allow \(\tau_1 \leq \tau_2\) exactly when it’s impossible to reach a “stuck” state by replacing an evaluable \(\tau_2\)-type expression in a well-typed program with a \(\tau_1\)-type expression and evaluating the result. That is, \(\tau_1 \leq \tau_2\) means that replacing \(\tau_2\)-type expressions with \(\tau_1\)-type expressions can’t break type safety.
On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types

Types \( \tau ::= \text{nat} | \text{real} | \tau_1 \to \tau_2 \)

Expressions \( e ::= n | r | \text{succ}(e) | \text{neg}(e) | \lambda x : \tau. e | e_1(e_2) | x \)

Evaluation contexts \( E ::= [] | \text{succ}(E) | \text{neg}(E) | E(e) | v(E) \)

Values \( v ::= n | r | \lambda x : \tau. e \)

3. AN INTRODUCTORY PROOF OF PRECISENESS

To more concretely understand and apply these definitions, let's consider a simple language \( \lambda \), a simply typed lambda calculus with base types \( \text{nat} \) (natural numbers) and \( \text{real} \) (real numbers), the idea being that \( \text{nat} \leq \text{real} \). Figure 1 presents the syntax and static and dynamic semantics. All the notation in Figure 1 is intended to have the usual meanings, with the usual assumptions being made. For example, variables are consistently renamed, through alpha-conversion, whenever necessary to avoid reintroducing variables into contexts, and empty contexts are normally omitted from judgment forms (e.g., \( e : \tau \) means \( \emptyset \vdash e : \tau \)).
The expressions in $\lambda$ are natural and real numbers, successor and negation operations, anonymous functions, applications, and variables. The negation operation is defined on natural and real numbers, while the successor operation is defined on natural, but not real, numbers.

The typing and operational rules for $\lambda$ are standard. Figure 1 uses evaluation contexts to define the operational semantics. Evaluation contexts mark where beta-reductions may occur; contexts here specify a left-to-right evaluation order and a call-by-value evaluation strategy. Section 6.2 discusses subtyping with other evaluation strategies.

3.1. Proof that $\lambda$’s Subtyping Relation is Sound

We wish to prove that the subtyping relation in $\lambda$ is precise with respect to type safety. We’ll prove soundness and then completeness, but let’s begin with a few standard lemmas (Lemmas 2–5). Because these lemmas, and their proofs, are standard, we don’t supply proof details. Our focus is on proving the preciseness of the subtyping relation.

**Lemma 2.** Weakening.

$$\forall \Gamma, e, \tau, \Gamma' \supseteq \Gamma : (\Gamma \vdash e : \tau \Rightarrow \Gamma' \vdash e : \tau)$$

**Proof.** By induction on the derivation of $\Gamma \vdash e : \tau$. 

**Lemma 3.** Universal Value-Inhabitation.

$$\forall \tau \exists v : (v : \tau)$$

**Proof.** By induction on the structure of $\tau$. 

**Lemma 4.** Variable Substitution.

$$\forall \Gamma, x, \tau', e, \tau, e' : ((\Gamma \cup \{x : \tau'\}) \vdash e : \tau \land \Gamma \vdash e' : \tau') \Rightarrow \Gamma \vdash [e'/x]e : \tau$$

**Proof.** By induction on the derivation of $\Gamma \cup \{x:\tau'\} \vdash e : \tau$. 

**Lemma 5.** Type Safety.

$$\forall e, \tau, e' : ((e : \tau \land e \mapsto^* e') \Rightarrow \neg\text{stuck}(e'))$$

**Proof.** By induction on the derivation of $e \mapsto^* e'$, using Progress and Preservation in the usual way.

The soundness of the subtyping relation now follows from the fact that the language is indeed type safe.

**Lemma 6.** Soundness.

$$\forall \tau_1, \tau_2 : (\tau_1 \leq \tau_2 \Rightarrow \neg\exists E, \tau, e, e' : (E[\tau_2] : \tau \land e : \tau_1 \land E[e] \mapsto^* e' \land \text{stuck}(e')))$$

**Proof.** Assume for the sake of obtaining a contradiction that $\tau_1 \leq \tau_2$ and there exist $E, \tau, e, \text{ and } e'$ such that $E[\tau_2] : \tau, e : \tau_1, E[e] \mapsto^* e'$, and stuck($e'$). Because $\tau_1 \leq \tau_2$ and $e : \tau_1$, we have $e : \tau_2$ by rule T-SUBSUME. Then because $E[\tau_2] : \tau$, we have $\{x:\tau_2\} + E[x] : \tau$, which combines with $e : \tau_2$ and Lemma 4 to imply that $E[e] : \tau$. Given that $E[e] : \tau$ and $E[e] \mapsto^* e'$, Lemma 5 ensures that $\neg\text{stuck}(e')$, which contradicts the assumption that stuck($e'$). Our original assumption was therefore false, so the lemma holds.

3.2. Induction on Failing Derivations

This paper’s completeness proofs use a technique that we call induction on failing derivations [Ligatti 2013a].

Standard induction on derivations is a form of tree induction; to prove that some property $P$ holds on all roots of derivation trees (i.e., valid judgments), one may prove
that \( P \) holds on all leaves of the trees (base cases) and on the internal nodes (inductive cases). When proving that \( P \) holds on the internal nodes, one may inductively assume that the property holds on all the children (i.e., subtrees) of that node, each of which must also be a valid derivation tree.

Dually, let’s consider failing derivation trees, which are finite trees in which a derivation “gets stuck” at one or more points. These points of failure are leaves in failing derivation trees. For example, trying to derive the invalid judgment

\[
(real \rightarrow real) \rightarrow (nat \rightarrow real) \leq (real \rightarrow nat) \rightarrow (real \rightarrow real)
\]

using the rules for subtyping in \( \lambda \) produces the following failing derivation.

\[
\begin{array}{cccc}
\text{real} \leq \text{real} & \text{nat} \leq \text{real} & \text{real} \leq \text{real} & \text{real} \leq \text{real} \\
\text{real} \rightarrow \text{nat} \leq \text{real} \rightarrow \text{real} & \text{nat} \rightarrow \text{real} \leq \text{real} \rightarrow \text{real} & \\
\hline
\text{(real} \rightarrow \text{real}) \rightarrow (\text{nat} \rightarrow \text{real}) \leq (\text{real} \rightarrow \text{nat}) \rightarrow (\text{real} \rightarrow \text{real})
\end{array}
\]

In \( \lambda \)'s subtyping system, every undervariable \( \tau \leq \tau' \) judgment has a finite failing derivation, in which at least one leaf judgment in the derivation tree fails (because it can’t be the conclusion of any inference rule).

Notice that all judgments between a failing leaf and the root of the derivation tree must also be failing. Proofs by induction on failing derivations trace the failure along this path, showing that some property holds on every (undervariable) judgment along the way. More specifically, proofs by induction on failing derivations show as base cases that the property of interest holds on all possible failing leaf judgments and then, while inductively assuming that the property holds on the failing premise(s) of a failing internal judgment \( J \), show that the property holds on \( J \) as well.

Notice also that the inductive hypothesis with regular proofs by induction on derivations applies universally to all premises of the internal judgment being considered, because all its premises must be derivable. In contrast, the inductive hypothesis with proofs by induction on failing derivations applies existentially to one or more premises of the internal judgment being considered, because at least one of its premises must be undervariable.

As an example, let’s consider proving a property \( P \) on undervariable \( \tau \leq \tau' \) judgments in \( \lambda \), by induction on failing derivations. The leaf nodes in a failing \( \tau \leq \tau' \) derivation can only occur when \( \tau = \text{real} \) and \( \tau' = \text{nat} \), or when exactly one of \( \tau \) and \( \tau' \) is a function type; hence, the base cases of the proof must show that \( P \) holds on all such judgments. The inductive case occurs when subtyping function types; all internal nodes in this system’s failing derivation trees must be attempts to subtype function types. Hence, the proof must show, while inductively assuming that \( P \) holds on the rule’s failing premise (i.e., subtyping the argument or return type), that \( P \) also holds on the failing conclusion.

Proof by induction on failing derivations is useful for establishing the completeness of a subtyping relation. Recall from Definition 1 that completeness requires: for all types \( \tau_1 \) and \( \tau_2 \), if there do not exist \( E, \tau, c, \) and \( c' \) such that \( E[\tau_2]:\tau, c:\tau_1, E[c] \mapsto c' \), and \( \text{stuck}(c') \), then \( \tau_1 \leq \tau_2 \). Although it may not be obvious how to prove this property directly, we can approach its contrapositive neatly by induction on the failing derivation of \( \tau_1 \leq \tau_2 \).

This article only considers subtyping systems in which, for all undervariable judgments \( J \), there exists a finite failing derivation tree rooted at \( J \). Hence, proof by induction on failing derivations will always be a viable technique here.
3.3. Proof that \( \lambda \)'s Subtyping Relation is Complete

Lemma 7 uses induction on failing derivations to prove a slightly stronger version of completeness. The proof is constructive; given any \( \tau_1 \) and \( \tau_2 \) such that \( \tau_1 \leq \tau_2 \) is not derivable, the proof shows how to (inductively) construct a well-typed program that gets stuck when its \( \tau_2 \)-type subexpression is replaced by a \( \tau_1 \)-type value.

**LEMMA 7. Strong Completeness.**

\[ \forall \tau_1, \tau_2 : (\tau_1 \leq \tau_2 \text{ not derivable} \Rightarrow \exists E, \tau, v, e : (E[\tau_2] : \tau \land \tau \land E[v] \rightarrow^* e \land \text{stuck}(e))) \]

**PROOF.** By induction on the failing derivation of \( \tau_1 \leq \tau_2 \). This derivation can fail when \( \tau_1 = \text{real} \) and \( \tau_2 = \text{nat} \), or when exactly one of \( \tau_1 \) and \( \tau_2 \) is a function type. These are the only base cases of a failing derivation of \( \tau_1 \leq \tau_2 \) (i.e., they're the only possible leaf nodes in a failing derivation tree rooted at \( \tau_1 \leq \tau_2 \)). We first prove the lemma for these base cases.

— Case \( \tau_1 = \text{real} \) and \( \tau_2 = \text{nat} \):

Define:

- \( E = \text{succ}([[]]) \)
- \( \tau = \text{nat} \)
- \( v = 2.718 \)
- \( e = \text{succ}(2.718) \)

Then:

- \( E[\tau_2] : \tau \) (by rules T-CTXT, T-SUC, and T-VAR)
- \( v : \tau_1 \) (by T-REAL)
- \( E[v] \rightarrow^* e \) (by the reflexive multistep rule)
- \( \text{stuck}(e) \) (by the definitions of stuck and \( e \))

— Case \( \tau_1 = \tau'_1 \rightarrow \tau''_1 \) and \( \tau_2 \neq \tau'_2 \rightarrow \tau''_2 \):

Because \( \tau_2 \) isn't a function type, it must be \( \text{nat} \) or \( \text{real} \). Also, by Lemma 3 there exists a \( v'_2 \) such that \( v'_2 : \tau''_2 \). Note that it is straightforward to prove Lemma 3 constructively, so we can construct this \( v'_2 \). Now define:

- \( E = \text{neg}([[]]) \)
- \( \tau = \text{real} \)
- \( v = \lambda x : \tau'_1, v'_2 \)
- \( e = \text{neg}((\lambda x : \tau'_1, v'_2) \)

Then:

- \( E[\tau_2] : \tau \) (by rules T-CTXT, T-NEG, T-VAR, and T-SUBSUME when \( \tau_2 = \text{nat} \))
- \( v : \tau_1 \) (by T-LAM)
- \( E[v] \rightarrow^* e \) (by the reflexive multistep rule)
- \( \text{stuck}(e) \) (by the definitions of stuck and \( e \))

— Case \( \tau_1 \neq \tau'_1 \rightarrow \tau''_1 \) and \( \tau_2 = \tau'_2 \rightarrow \tau''_2 \):

Because \( \tau_1 \) isn't a function type, it must be \( \text{nat} \) or \( \text{real} \). Also, by Lemma 3 there exists a \( v'_2 \) such that \( v'_2 : \tau'_2 \). Now define:

- \( E = ([[]])(v'_2) \)
- \( \tau = \tau'_2 \)
- \( v = 0 \)
- \( e = 0(v'_2) \)

Then:

- \( E[\tau_2] : \tau \) (by rules T-CTXT, T-APP, and T-VAR)
- \( v : \tau_1 \) (by the fact that \( \tau_1 \) is \( \text{nat} \) or \( \text{real} \))
- \( E[v] \rightarrow^* e \) (by the reflexive multistep rule)
- \( \text{stuck}(e) \) (by the definitions of stuck and \( e \))
The only inductive case of a failing $\tau_1 \leq \tau_2$ derivation (i.e., internal node in a failing derivation tree) is a use of the rule for subtyping function types. By assumption, the conclusion of this rule is underivable, so at least one of its two premises must also be underivable. That is, the contravariant-argument-type judgment is underivable or the covariant-return-type judgment is underivable. Let’s examine each of these two subcases.

— Case $\tau_1 = \tau'_1 \rightarrow \tau'_2$, $\tau_2 = \tau''_2 \rightarrow \tau''_2$, and the $\tau'_2 \leq \tau'_1$ premise is underivable:
By Lemma 3 there exists a $v''_1$ such that $v''_1 : \tau''_1$. Also, by the inductive hypothesis (where $\tau''_2 \leq \tau''_1$ is underivable), there exist $E'$, $\tau'$, $v'$, and $e'$ such that:
- $E'[\tau'_1] : \tau'$
- $v' : \tau'_2$
- $E'[v'] \mapsto^* e'$
- stuck($e'$)

Now define:
- $E = [\cdot](v')$
- $\tau = \tau''_2$
- $v = \lambda x : \tau'_1, ((\lambda y : \tau', v''_1)(E'[x]))$
- $e = (\lambda y : \tau', v''_1)(e')$

Then:
- $E[\tau_2] : \tau$ (by T-CTX, T-APP, T-VAR, and Lemma 2, where $\tau_2 = \tau''_2 \rightarrow \tau''_2$ and $v' : \tau'_2$
- $v : \tau_1$ (by T-LAM, T-APP, and Lemma 2, where $\tau_1 = \tau'_1 \rightarrow \tau''_1$, $v'_1 : \tau''_1$, and $E'[\tau'_1] : \tau'$
- $E[v] \mapsto e$ (because $E[v] \mapsto (\lambda y : \tau', v''_1)(E'[v'])$, where $E'[v'] \mapsto^* e'$
- stuck($e'$) (because stuck($e'$))

— Case $\tau_1 = \tau'_1 \rightarrow \tau'_2$, $\tau_2 = \tau''_2 \rightarrow \tau''_2$, and the $\tau''_2 \leq \tau''_1$ premise is underivable:
By Lemma 3 there exists a $v''_2$ such that $v''_2 : \tau''_2$. Also, by the inductive hypothesis (where $\tau''_2 \leq \tau''_1$ is underivable), there exist $E'$, $\tau'$, $v'$, and $e'$ such that:
- $E'[\tau''_2] : \tau'$
- $v' : \tau''_2$
- $E'[v'] \mapsto^* e'$
- stuck($e'$)

Now define:
- $E = E'[\cdot](v''_2)$ (i.e., build $E$ by filling the hole of $E'$ with $\cdot(v''_2)$)
- $\tau = \tau''_2$
- $v = \lambda x : \tau'_1, v'$
- $e = e'$

Then:
- $E[\tau_2] : \tau$ (because $E'[\tau''_2] : \tau'$ means that $\{y : \tau''_2\} \vdash E'[y] : \tau'$, which implies by Lemma 2 that $\{z : \tau_2, y : \tau''_2\} \vdash E'[y] : \tau'$; then because $\{z : \tau_2\} \vdash E'[v'_2] : \tau''_1$, Lemma 4 ensures that $\{z : \tau_2\} \vdash E'[\tau_2(v'_2)] : \tau''_1$, which means that $E'[\tau_2(v''_2)] : \tau'$)
- $v : \tau_1$ (by T-LAM and Lemma 2, where $\tau_1 = \tau'_1 \rightarrow \tau'_1$ and $v' : \tau'_1$
- $E[v] \mapsto e$ (because $E[v] \mapsto E'[v']$, where $E'[v'] \mapsto^* e'$
- stuck($e'$) (because stuck($e'$))

Hence, in all cases, the requisite $E$, $\tau$, $v$, and $e$ can be constructed to satisfy the lemma. □

The completeness of $\lambda$’s subtyping relation follows immediately from Lemma 7. By combining this completeness result with the soundness established in Lemma 6, preciseness follows as a corollary.

ACM Journal Name, Vol. V, No. N, Article A, Publication date: January YYYY.
4. INCOMPLETENESS WITH THE AMBER RULES, FOR SUBTypING ISO-RECURSIVE TYPES

Let’s focus now on subtyping iso-recursive types.

The Amber rules have at least two sources of incompleteness, one stemming from
contravariant subtyping and another from incomparability between type variables and
recursive types.

4.1. A First Source of Incompleteness: Complications with Contravariance

Suppose that $\lambda$ contains recursive types and the Amber subtyping rules (as stated
in Section 1.1). Also suppose that all the premises and conclusions of the existing
subtyping rules, shown in Figure 1, have $S \vdash$ prepended to them, so that subtyping-
assumption sets $S$ get carried through derivations. Then we can derive some reflexive
relationships, like $\mu \tau. (\text{nat} \rightarrow t) \leq \mu \tau'. (\text{nat} \rightarrow t')$:

\[
\begin{array}{c}
\frac{(t \leq t') \vdash \text{nat} \leq \text{nat} \quad (t \leq t') \vdash t \leq t'}{\mu \tau. (\text{nat} \rightarrow t) \leq \mu \tau'. (\text{nat} \rightarrow t')}
\end{array}
\]

But we can’t derive other reflexive relationships, like $\mu \tau. (t \rightarrow \text{nat}) \leq \mu \tau'. (t' \rightarrow \text{nat})$:

\[
\downarrow \text{Derivation fails here} \downarrow
\]

\[
\frac{(t \leq t') \vdash t' \leq t \quad (t \leq t') \vdash \text{nat} \leq \text{nat}}{(t \leq t') \vdash t \rightarrow \text{nat} \leq t' \rightarrow \text{nat}}
\]

\[
\mu \tau. (t \rightarrow \text{nat}) \leq \mu \tau'. (t' \rightarrow \text{nat})
\]

This lack of reflexivity stems from a key underlying problem: the rules can’t subtype
variables defined in covariant positions but used in contravariant positions (and vice
versa).

We could try to fix this problem by reversing the order of subtyping assumptions
when subtyping in contravariant positions, resulting in the following rule:

\[
\frac{(t' \leq t \mid t \leq t' \in S) \vdash t'_1 \leq \tau_1 \quad S \vdash \tau_2 \leq t_2'}{S \vdash \tau_1 \rightarrow \tau_2 \leq t'_1 \rightarrow t_2'}
\]

However, such a rule would unsoundly allow $\mu \tau. (t \rightarrow \text{nat}) \leq \mu \tau'. (t' \rightarrow \text{real})$. To see why
$\tau = \mu \tau. (t \rightarrow \text{nat})$ can’t be a subtype of $\tau' = \mu \tau'. (t' \rightarrow \text{real})$, suppose $\tau \leq \tau'$ and define $f$ and
g as follows.

\[ f = \lambda x : \tau . \text{succ} (\text{unroll}(x) \ x) \]
\[ g = \lambda y : \tau'. (2.718) \]

Then roll($f$) would have type $\tau$ and (by subsumption) $\tau'$, which means that
(roll(roll($f$)))(roll($g$)) would have type real but evaluates to the stuck expression succ(2.718).

Another way to try to fix the problem would be to allow the same type variables to
appear on both sides of the $\leq$ symbol. Then we could derive $\mu \tau. (t \rightarrow \text{nat}) \leq \mu \tau. (t \rightarrow \text{nat})$, as desired, but we could also derive that $\mu \tau. (t \rightarrow \text{nat}) \leq \mu \tau. (t \rightarrow \text{real})$, which we just
showed is unsound.

The only other approach we can think of to make the Amber rules reflexive, so we
can derive that types like $\mu \tau. (t \rightarrow \text{nat})$ are subtypes of themselves, is to add a rule explicitly saying so. To create such a rule, let’s focus on the core problem here: when
the Amber rules reach a judgment $\mu \tau. \tau \leq \mu \tau'. \tau'$, they attempt to derive $\tau \leq \tau'$ while assuming $t \leq t'$ but are unable to derive that $t \leq t$. Hence, the problem arises with non-
antisymmetric recursive types, where we have $\mu \tau. \tau \leq \mu \tau'. \tau'$ and $\mu \tau'. \tau' \leq \mu \tau$ (in which
On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types

Given that the core problem relates to non-antisymmetric recursive types, we can't fix the problem just by adding a rule to say that if \( \tau \) is alpha-equivalent to \( \tau' \) then \( \tau \leq \tau' \). Such a rule addresses one source of non-antisymmetry in subtyping relations (i.e., alpha-equivalence) but doesn't address others, such as permutations of record or variant fields. For example, such a rule still wouldn't allow us to derive that

\[
\mu t.\{a:t,b:nat\}\rightarrow nat \leq \mu t'.\{(b:nat,a:t')\}\rightarrow nat.
\]

To fix the general problem, then, the new rule would have to allow \( \tau \leq \tau' \) exactly when \( \tau \) and \( \tau' \) exhibit non-antisymmetry, that is, when \( \tau \) and \( \tau' \) are subtypes of each other (but not equal). Let's define \( \tau \) and \( \tau' \) to be equivalent iff they subtype each other. Then our final rule to fix the contravariance problem would say that if \( \tau \) is equivalent to \( \tau' \) then \( \tau \leq \tau' \). But because equivalence of \( \tau \) and \( \tau' \) requires \( \tau \leq \tau' \), such a rule is circular and not immediately helpful. Nonetheless, this rule could be helpful in cases where type equivalence can be defined using some equivalent alternative rules (e.g., in terms of alpha-equivalence and field permutations), at the cost of complicating the subtyping rules and algorithm with these alternative rules.

4.2. A Second Source of Incompleteness: Complications with Unrolling

Besides the contravariance problem, the Amber rules are incomplete in other ways. For example, consider the recursive types \( \tau' \) and \( \tau \) defined as follows.

\[
\begin{align*}
\tau' &= \mu i.\{\text{add}:i\rightarrow \text{unit}\} \\
\tau &= \mu l.\{\mu i'.\{\text{add}:i'\rightarrow \text{unit}\}\rightarrow \text{unit}, \text{min}:\text{unit}\rightarrow \text{int}\}
\end{align*}
\]

These types arise naturally when encoding the following OOPL classes into a language like \( \lambda \) (extended to have record, unit, int, and recursive types).

```java
class Integer {
    public Integer(int i) {n=i}
    public void add(Integer i) {n = n + i.n}
    protected int n=0
}

class LowBoundNat extends Integer {
    public LowBoundNat(int lowBound, int i) {if 0<=lowBound<=i then (min=lowBound; n=i)}
    public void add(Integer i) {if ((n + i.n) < min) then n=min else super.add(i)}
    public int min() {min}
    protected int min=0
}
```

The Integer type may be encoded as \( \tau' \), and the LowBoundNat type as \( \tau \). One would expect \( \tau \leq \tau' \) in an iso-recursive system because the only way a \( \tau' \)-type expression can be eliminated is by unrolling it, to produce an expression of type \( \{\text{add}:\tau'\rightarrow \text{unit}\} \), while unrolling a \( \tau \)-type expression produces an expression of type \( \{\text{add}:\mu i'.\{\text{add}:i'\rightarrow \text{unit}\}\rightarrow \text{unit}, \text{min}:\text{unit}\rightarrow \text{int}\} \), which is a subtype of \( \{\text{add}:\tau'\rightarrow \text{unit}\} \). Thus, it's always safe for a \( \tau \)-type expression to stand in for a \( \tau' \)-type expression.

However, the Amber rules (in conjunction with standard subtyping rules for records and functions) can't derive \( \tau \leq \tau' \), as Figure 2 illustrates.

For another example, let's redefine \( \tau \) and \( \tau' \) as follows.

\[
\begin{align*}
\tau' &= \mu c.((c + \text{real}) + c) \\
\tau &= \mu a.((\mu b.((b + \text{nat}) + a)) + \text{nat} + a)
\end{align*}
\]
A:12

Jay Ligatti et al.

\[ \begin{array}{c}
\downarrow \text{Derivation fails here} \downarrow \\
\{l \leq i\} \vdash i \leq \mu l. (\text{add} \cdot i \vdash \text{unit}) \\
\{l \leq i\} \vdash \text{unit} \leq \text{unit} \\
\{l \leq i\} \vdash (\mu l. (\text{add} \cdot i \vdash \text{unit})) \vdash \text{unit} \leq i \vdash \text{unit}
\end{array} \]

This may be a more interesting example because all the declared type variables get used (unlike the type variable \( \tau \) in the previous example’s \( \tau \)). Again, the Amber rules (in conjunction with the standard subtyping rule for binary sums) can’t be used to derive \( \tau \leq \tau' \), as shown in Figure 3. We prove that it’s safe to consider \( \tau \leq \tau' \) in two steps: first, Section 5 shows that \( \tau \leq \tau' \) is derivable using new subtyping rules; second, Appendix A shows that the new subtyping rules are indeed sound with respect to type safety.

Notice that, in both Figures 2 and 3, the inability to derive a valid subtyping judgment stems from the rules’ inability to distinguish type variables from the recursive types they represent. Additional or alternative rules are again needed.

5. A PRECISE SYSTEM FOR SUBTYING ISO-RECURSIVE TYPES

This section defines new rules, and an algorithm, for subtyping iso-recursive types. Appendix A contains a proof that the new subtyping relation is precise with respect to type safety.

5.1. A Language with Aggregate Data Types, \( L_{+x}^{-\mu} \)

Let’s define a new language called \( L_{+x}^{-\mu} \) having function types, binary (disjoint) sum and product types, and iso-recursive types. Figures 4–6 present the syntax and static and dynamic semantics. Again, all the notation is intended to have the usual meanings, with the usual assumptions being made.
There are two categories of types in $L_{+\times}^\mu$, one for closed types $\tau$ (having no free type variables) and the other for possibly open types $\tau$ (possibly having free type variables). This paper assumes that closed types never use undeclared variables. Note that unrolling a closed recursive type $\tau = \mu t. \tau$ produces another closed type, $[\mu t. \tau]$. The base types in $L_{+\times}^\mu$ remain nat and real, with successor and negation operations defined as in $\lambda$. Functions in $L_{+\times}^\mu$ are named and may be recursive. The decision to allow recursive functions was made because real languages sophisticated enough to have iso-recursive types and subtyping seem likely to also have recursive functions.

The typing rules for $L_{+\times}^\mu$, shown in Figure 5, are standard. Section 5.2 will define the subtyping judgment used by rule T-SUBSUME.

The operational rules for $L_{+\times}^\mu$, shown in Figure 6, are also standard. Note that the evaluation contexts in $L_{+\times}^\mu$ again specify left-to-right, call-by-value evaluation. Section 6.2 discusses other evaluation strategies.
Evaluation contexts $E ::= [] | succ(E) | neg(E) | E(e) | v(E) | (E,e) | (v,E) |
E.fst | E.snd | inl,E | inr,E | unroll(E) | roll(E) |
case $E$ of $\text{inl} x \Rightarrow e_1$ else $\text{inr} y \Rightarrow e_2$

Values $v ::= n | r | \text{fun } f(x : \tau_1) : \tau_2 = e | (v_1, v_2) | \text{inl}_r(v) | \text{inr}_r(v) | \text{roll}(v)$

$$\begin{align*}
e &\rightsquigarrow e' \\
e &\rightsquigarrow _\beta e' \\
\text{stuck}(e) &\quad \neg \exists v : (e = v) \\
\text{unroll}(e) &\quad \neg \exists e' : (e \mapsto e')
\end{align*}$$

Fig. 6. Dynamic semantics of $L_{+\mu}^{-\mu}$.

5.2. The Subtyping Rules for $L_{+\mu}^{-\mu}$

Incompleteness in the Amber rules (for subtyping iso-recursive types) ultimately stems from their lack of considering unrolled types. Iso-recursive types get eliminated by unrolling, so type $\mu t.\tau$ should be a subtype of $\mu t'.\tau'$ if the unrolled version of $\mu t.\tau$ is a subtype of the unrolled version of $\mu t'.\tau'$. When considering whether these unrolled versions are in a subtype relationship (i.e., whether $[\mu t.\tau / t]\tau \leq [\mu t'.\tau' / t']\tau'$), one can assume that $\mu t.\tau \leq \mu t'.\tau'$ because any expressions of types $\mu t.\tau$ and $\mu t'.\tau'$ encountered by unrolling expressions of types $\mu t.\tau$ and $\mu t'.\tau'$ can be unrolled and manipulated in the same ways again.

This discussion leads to the following subtyping rules for iso-recursive types:

\[
\frac{S \cup \{\mu t.\tau \leq \mu t'.\tau'\} \vdash [\mu t.\tau / t]\tau \leq [\mu t'.\tau' / t']\tau'}{S \vdash \mu t.\tau \leq \mu t'.\tau'} \quad \text{S-Rec1}
\]

\[
\frac{S \cup \{\mu t.\tau \leq \mu t'.\tau'\} \vdash [\mu t.\tau / t]\tau \leq [\mu t'.\tau' / t']\tau'}{S \vdash \mu t.\tau \leq \mu t'.\tau'} \quad \text{S-Rec2}
\]
EC — Rules S-R

— Other systems have used rules similar to S-R

— As with other judgment forms that use contexts, we abbreviate judgments of the form $F \vdash \tau \leq \mu t$. Derivation of $L \leq I$ using the new subtyping rules S-Rec1 and S-Rec2, where $L=\mu t.((\mu b.((b + nat) + a)) + nat) + a)$, $I=\mu t.((b + nat) + A)$, $I'=\mu t'.(\mu b.((b + nat) + A) + C=\mu c.((c + real) + c)$, and $S = \{A \leq C, B \leq C\}$.

<table>
<thead>
<tr>
<th>$F \vdash t \leq t'$</th>
<th>$F \vdash \text{unit} \leq \text{unit}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \vdash t' \rightarrow \text{unit} \leq I \rightarrow \text{unit}$</td>
<td>$F \vdash {\text{add} : I \rightarrow \text{unit}} \leq {\text{add} : I \rightarrow \text{unit}}$</td>
</tr>
<tr>
<td>${L \leq I, I \leq I'} \vdash I \leq \mu t. \text{unit} \leq \mu t. \text{unit}$</td>
<td>${L \leq I, I \leq I'} \vdash \mu t. \text{unit} \leq \mu t. \text{unit}$</td>
</tr>
<tr>
<td>${L \leq I, I \leq I'} \vdash I \rightarrow \text{unit} \leq I \rightarrow \text{unit}$</td>
<td>${L \leq I} \vdash I \leq I'$</td>
</tr>
</tbody>
</table>
| $\{L \leq I\} \vdash \{\text{add} : I \rightarrow \text{unit}\} \leq \{\text{add} : I \rightarrow \text{unit}\}$ | $\{L \leq I\} \vdash \text{unit} \leq \mu t$. Derivation of $Fig. 7$ using the new subtyping rules S-Rec1 and S-Rec2, where $L=\mu t.((\mu b.((b + nat) + a)) + nat) + a)$, $I=\mu t.((b + nat) + A)$, $I'=\mu t'.(\mu b.((b + nat) + A) + C=\mu c.((c + real) + c)$, and $S = \{A \leq C, B \leq C\}$.

<table>
<thead>
<tr>
<th>$S \vdash B \leq C$</th>
<th>$S \vdash \text{nat} \leq \text{real}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \vdash B + \text{nat} \leq C + \text{real}$</td>
<td>$S \vdash A \leq C$</td>
</tr>
<tr>
<td>$S \vdash (B + \text{nat}) + A \leq (C + \text{real}) + C$</td>
<td>${A \leq C} \vdash B \leq C$</td>
</tr>
<tr>
<td>${A \leq C} \vdash \text{nat} \leq \text{real}$</td>
<td>${A \leq C} \vdash A \leq C$</td>
</tr>
<tr>
<td>${A \leq C} \vdash (B + \text{nat}) + A \leq (C + \text{real}) + C$</td>
<td>${A \leq C} \vdash A \leq C$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$A \leq C$</th>
</tr>
</thead>
</table>

— As with other judgment forms that use contexts, we abbreviate judgments of the form $\emptyset \vdash \tau_1 \leq \tau_2$ as $\tau_1 \leq \tau_2$.

— Rules S-Rec1 and S-Rec2 maintain the invariant that only closed types are being considered; unrolling a closed type produces another closed type.

— Other systems have used rules similar to S-Rec1 and S-Rec2 to define equivalence, rather than subtyping, relations on iso-recursive types [League and Shao 1998; Van-derwaart et al. 2003].

Rules S-Rec1 and S-Rec2 enable derivations of all the subtyping judgments that Section 4 showed were sources of Amber-rule incompleteness. For example, $\mu t.(t \rightarrow \text{nat}) \leq \mu t.(t \rightarrow \text{nat})$ and $\mu t.(t \rightarrow \text{nat}) \leq \mu t'.(t' \rightarrow \text{nat})$ are now derivable, while $\mu t.(t \rightarrow \text{real}) \leq \mu t.(t \rightarrow \text{real})$ and $\mu t.(t \rightarrow \text{nat}) \leq \mu t'.(t' \rightarrow \text{real})$ are underviable (as is required for soundness). Recall that Figures 2–3 showed that two other subtyping judgments are underviable with the Amber rules; now Figures 7–8 show that the same judgments are derivable with S-Rec1 and S-Rec2.

Unfortunately, rules S-Rec1 and S-Rec2 are insufficient for making the subtyping relation complete (as we learned by attempting an early proof of completeness). Because $L^\mu_{x+}$ has recursive functions, all types are inhabited (e.g., the expression $\text{fun f(x:nat):}(x \rightarrow f(x))(0)$ has type $\tau$ for any $\tau$). However, because $L^\mu_{x+}$ has recursive types, some types are value-inhabited (i.e., inhabited only by nonterminating ex-
pressions). For example, the type $\mu t.t$ is uninhabited by (normal-form) values; writing a value of type $\mu t.t$ would require already having a value of type $\mu t.t$ to roll. Hence, every expression of type $\mu t.t$ must diverge.

We can treat any type inhabited only by diverging expressions, such as $\mu t.t$, as being equivalent to a $\perp$ type. If all expressions of a type $\tau$ diverge, then any $\tau$-type expression can substitute for any expression of any type; such a substitution won’t compromise type safety because the $\tau$-type expression would have to be evaluated to a value before it could be used in an unsafe way.

Moreover, any expression can substitute for a function whose argument type is uninhabited by values (e.g., $\mu t.t$), without compromising type safety. Intuitively, such a function can never be applied because the call-by-value semantics requires the argument to be evaluated to a value, something guaranteed to never happen. Because such a function, when part of a well-typed program, can never be applied, we can substitute any expression—of any type—for the function.

Based on the preceding discussion, we add the following rules to the definition of subtyping in $L_{+\times}^\mu$.

$$\text{val}(\tau) = \emptyset$$
$$S \vdash \tau \leq \tau' \quad \text{S-}\perp$$
$$\text{val}(\tau'_1) = \emptyset$$
$$S \vdash \tau \leq \tau'_1 \rightarrow \tau'_2 \quad \text{S-}\top$$

These rules use an auxiliary value-uninhabitation judgment of the form $\text{val}(\tau) = \emptyset$ to indicate that $\tau$ is value-uninhabited. Although S-\perp and S-\top are similar to rules described by Vouillon [Vouillon 2004], we’re not aware of existing rules or an algorithm for deciding value-uninhabitation. The related problem of deciding expression-uninhabitation (also called the emptiness problem) has been well studied (e.g., [Urzyczyn 1994]). The closest existing rules we know for deciding value-uninhabitation are for contractive types (e.g., [Im et al. 2013]), but contractive-type rules are incomplete—they reject not only value-uninhabited types, but also many value-inhabited types like $(\mu t.t) \rightarrow \text{nat}$. For this paper’s subtyping system to be precise with respect to type safety, we’ll need rules for precisely deciding value-uninhabitation.

Combining rules S-REC1, S-REC2, S-\perp, and S-\top with the standard rules for subtyping nat, real, function, sum, and product types produces the subtyping system shown in Figure 9. This is the full definition of the subtyping relation for $L_{+\times}^\mu$.

Figure 9 does contain rules for deciding value-uninhabitation. Note that nat, real, and function types are always value-inhabited. Sum type $\tau_1 + \tau_2$ is value-uninhabited when both $\tau_1$ and $\tau_2$ are value-uninhabited, and product type $\tau_1 \times \tau_2$ is value-uninhabited when $\tau_1$ or $\tau_2$ is value-uninhabited. Finally, recursive type $\tau$ is value-uninhabited when the unrolled version of $\tau$ is value-uninhabited under the assumption that $\tau$ is value-uninhabited (because we can’t make a value of type $\tau$ by relying on already having one).

The subtyping system defined here makes the following precedence assumptions, which dictate the order in which Figure 9’s rules must be applied.

— S-\perp has the highest precedence, then S-\top and S-REC2; after that at most one subtyping rule applies.
— U-PROD1 and U-REC2 have the highest precedence; after that at most one value-uninhabitation rule applies.

For example, rule S-REC1 is only used when rule S-REC2 doesn’t apply. The derivations in Figures 7–8 remain valid under these precedence assumptions.

Note that rules that take precedence over other rules can’t appear in failing derivations because when such rules fail, the lower-precedence rules get used instead. Hence, rules S-\perp, S-\top, S-REC2, U-PROD1, and U-REC2 never appear in failing derivations.
On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types

We’re thus left with $S-F$.

On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types

5.3. A Subtyping Algorithm

With the precedence assumptions stated above and beginning with an empty $S$ context, the subtyping rules in Figure 9 are deterministic and algorithmic: there’s always at most one next rule to try, and every derivation fails or succeeds at a finite height. This

finite-height (termination) property holds because all the rules’ premises decrease the sizes of the types under consideration, except that recursive types may be unrolled a limited number of times (e.g., the $U\vdash \text{val}(\tau) = \emptyset$ rules may unroll every recursive type in $\tau$ at most once). Hence, a simple algorithm for deciding whether $\tau_1 \leq \tau_2$ is to traverse the (possibly failing) derivation of $\tau_1 \leq \tau_2$ and reject iff the derivation fails at some point (because one of the traversed judgments can’t be the conclusion of any inference rule).

This basic subtyping algorithm can be optimized to prevent redundant computations. Figures 10–12 present one such implementation in Standard ML. This imple-
mentation is a complete but slightly edited version of the actual implementation posted online [Ligatti 2013b]. The actual implementation is 73 lines of code, not counting whitespace and comments.

The code in Figures 10–12 is heavily commented, to explain the optimized algorithm’s operation and correctness.

**Analysis of Running Time.** The optimized algorithm decides whether $\tau_1 \leq \tau_2$ in $O(mn)$ time, where:

- $m$ is the number of $\mu$-terms (i.e., variable declarations) in $\tau_1$ or $\tau_2$, whichever is greater (or 1 if neither contain $\mu$-terms)
- $n$ is the total size of $\tau_1$ or $\tau_2$, whichever is greater.

The main subtyping function, sub in Figure 12, first counts the number of variable declarations in its argument types $t_1$ and $t_2$, and then allocates tables ($UT_1$, $UT_2$, $U_1$, and $U_2$) of these sizes, all in $O(n)$ time. The sub function next calls init (Figure 11) on each of $t_1$ and $t_2$, in order to (1) build CEtyp-versions of $t_1$ and $t_2$, and (2) properly initialize the previously allocated tables.

The init function implements the $\text{val}(\tau) = \emptyset$ judgment on all $n$ component types of $\tau$ and runs in $O(mn)$ time. This function traverses a given type tree and commits to the value-uninhabitation of each of its recursive types in turn, where each of the $m$ commits requires traversing the recursive type's subtree in $O(n)$ time (all the other cases of init, which don't involve committing to the value-uninhabitation of a recursive type, run in time that's a constant plus the time required to init subtrees, for a total time that's proportional to the size of the subtree being considered). Note that init runs in $O(mn)$ time because it initializes tables for all the component types of its type argument; in applications where we only care to test whether one overall type is value-inhabited, we could simply call init with the final $b$ argument set to true, in which case init decides value-inhabitation in $O(n)$ time.

After init has completed, all value-uninhabitation checks and recursive-type unrolling can be performed in constant time. At this point, sub allocates and initializes two tables ($S_1$ and $S_2$) for storing recursive-type subtyping assumptions, in $O(m^2)$ time.

Finally, sub invokes its helper function subh, which implements the $\tau_1 \leq \tau_2$ judgment. All cases of subh run in time that's a constant plus the time required to do other subtyping comparisons (i.e., recursive calls to subh, if any). Hence, subh runs in time that's proportional to the number of subtyping comparisons made. Every type outside of $\mu$-terms may be involved in at most one subtyping comparison, and every type within a $\mu$-term (of which there are $O(n)$) may be compared to at most one corresponding type in each of the other side’s $\mu$-terms (of which there are $O(m)$), as pairs of recursive types are compared. The total number of subtyping comparisons is therefore $O(mn)$, so subh runs in $O(mn)$ time.

Thus, the total running time of sub is $O(n)$ (to allocate $UT_1$, $UT_2$, $U_1$, and $U_2$) plus $O(mn)$ (to run init) plus $O(m^2)$ (to allocate $S_1$ and $S_2$) plus $O(mn)$ (to run subh). Because $0 < m < n$, the total running time of the subtyping algorithm is $O(mn)$.
(* Constructors for types. Type variables are represented as *)

(* integers, which are assumed to be named 0, 1, etc., so for all *)

(* types T passed as arguments to the subtype-testing function *)

(* sub, the set of type variables in T is \{0..n\} for some n. *)

(* We also assume that T never uses undeclared variables and has *)

(* been alpha-converted to ensure the uniqueness of every *)

(* declared variable. *)

(* As an example, the type A from Figure 8 could be encoded as: *)

(* Rec(0,Sum(Sum(Rec(1,Sum(Sum(Var(1),Nat),Var(0))),Nat),Var(0))) *)

(*

datatype typ = Nat | Real | Prod of typ * typ | Sum of typ * typ |

| Fun of typ * typ | Rec of int * typ | Var of int |

(* A CEtyp is a ‘compressed’ and ‘extended’ type. *)

(* ‘compressed’ means that all recursive types \(\mu n.t\) have been *)

(* replaced by just the type variable n. We’ll still be able *)

(* to look up the type to which n refers in an ‘unroll table’, *)

(* an array that maps n to (the CEtyp-version of) t. *)

(* Hence, CEtyp has no case for recursive types. *)

(* ‘extended’ means that the structure carries extra boolean *)

(* flags to memoize whether types are value-uninhabited. *)

(* Nat, real, and function types in this language are always *)

(* value-uninhabited, so their cases of CEtyp don’t need the *)

(* extra flag. Variable types also don’t need the flag; we’ll *)

(* instead use a separate array U to map type variables to *)

(* bools indicating value-uninhabitation. *)

(*

datatype CEtyp = CENat | CEReal | CEFun of CEtyp * CEtyp |

| CEVar of int | CEProd of CEtyp * CEtyp * bool |

| CESum of CEtyp * CEtyp * bool |

(* Returns the number of variables defined in a type. *)

fun numVars (Sum(t1,t2)) = numVars(t1) + numVars(t2) |

| numVars (Prod(t1,t2)) = numVars(t1) + numVars(t2) |

| numVars (Fun(t1,t2)) = numVars(t1) + numVars(t2) |

| numVars (Rec(_,t1)) = numVars(t1) + 1 |

| numVars _ = 0; |

(* Returns a bool indicating whether a given CEtyp is *)

(* value-uninhabited. The second parameter is an array *)

(* mapping type variables to value-uninhabitation flags. *)

(* That is, U[n] iff the recursive type to which *)

(* type-variable n refers is value-uninhabited. *)

(*

fun isUninhabited (CEProd(_,_,b)) _ = b |

| isUninhabited (CESum(_,_,b)) _ = b |

| isUninhabited (CEVar(n)) U = Array.sub(U,n) |

(* nat, real, and function types are value-inhabited *)

| isUninhabited _ = false; |

Fig. 10. Auxiliary definitions for the optimized subtyping algorithm. |
This initialization function has 4 parameters:

1. A type \( t \)
2. An unroll table \( UT \)
3. An array \( U \) mapping type variables to value-uninhabitation flags
4. A boolean \( b \) indicating whether we're trying to commit to the value-(un)inhabitation of some previously seen recursive type.

This function returns the CEtyp-version of \( t \) and properly initializes the \( UT \) and \( U \) arrays (as side effects).

```plaintext
fun init Nat _ _ = CENat
| init Real _ _ = CEReal
| init (Fun( t1 , t2 ) ) UT U b = CEFun(init t1 UT U b , init t2 UT U b)
| init (Sum( t1 , t2 ) ) UT U b =
  let val CEt1 = init t1 UT U b
  val CEt2 = init t2 UT U b
  in (* set the value-uninhabited flag based on rule U-Sum *)
    CESum(CEt1 , CEt2 , isUninhabited CEt1 U andalso isUninhabited CEt2 U)
  end
| init (Prod(t1,t2)) UT U b =
  let val CEt1 = init t1 UT U b
  val CEt2 = init t2 UT U b
  in (* set the flag based on rules U-Prod1 and U-Prod2 *)
    CEProd(CEt1 , CEt2 ,
      isUninhabited CEt1 U orelse isUninhabited CEt2 U)
  end
| init (Rec(n , t ) ) UT U b =
  (* Recursive type \( n \) is value-uninhabited iff \( t \) is
    value-uninhabited under the assumption that \( n \) is
    value-uninhabited (U-Rec1 and U-Rec2). Once we know
    whether \( t \) is value-inhabited, we can properly set \( U[n]\).
    Finally, if \( b=false \) then we're now committed to \( U[n]\) and
    can move on to processing \( t \), after which we can properly
    set \( UT[n]\) and return the compressed version of \( \mu n.t \),
    which is just the variable \( n \). *)
  (Array.update(U,n , true);
   Array.update(U,n , isUninhabited (init t UT U true ) U);
   if b then () else Array.update(UT,n , init t UT U false );
   CEVar(n))
  (* We commit to the value-uninhabitation of recursive types
    in this outer-to-inner fashion to properly handle types
    like \( \mu n.0.((\mu m.1.0) + \tau) \), in which the value-
    uninhabitation of an outer type (here, 0) determines the
    value-uninhabitation of an inner type (here, 1). *)
| init (Var(n)) _ _ = CEVar(n);
```

Fig. 11. Computation of value-uninhabitation in the optimized subtyping algorithm.
fun sub t1 t2 =
  let
  (* Allocate and initialize the unroll tables UT1 and UT2,
    * the uninhabitation arrays U1 and U2, and the compressed
    * and extended types CEt1 and CEt2 *)
  val m = numVars t1
  val n = numVars t2
  val UT1 = Array.array(m,CENat)
  val UT2 = Array.array(n,CENat)
  val U1 = Array.array(m,false)
  val U2 = Array.array(n,false)
  val CEt1 = init t1 UT1 U1 false
  val CEt2 = init t2 UT2 U2 false
  (* Now create arrays for storing subtyping assumptions.
    * S1[m][n] iff recursive type m in t1 is assumed to subtype
    * recursive type n in t2; similarly, S2[n][m] iff recursive
    * type n in t2 is assumed to subtype recursive type m in t1. *)
  val S1 = Array2.array(m,n,false)
  val S2 = Array2.array(n,m,false)
  (* The following helper subtyping function operates on CEtyp’s.
    * isContra indicates whether we’re subtyping contravariantly,
    * i.e., CEt1 is a type in t2 and CEt2 is a type in t1. *)
  fun subh CEt1 CEt2 isContra =
    let
      (* Normalize table names for contravariant subtyping *)
      val (S,UT1,UT2,U1,U2)=
        if isContra
        then (S2,UT2,UT1,U2,U1)
        else (S1,UT1,UT2,U1,U2)
    in
      isUninhabited CEt1 U1 (* rule S-Bottom *)
      orelse (case CEt2 of
        CFun(CEt2',..) => isUninhabited CEt2' U2 (* S-Top *)
      | _ => false)
      orelse case (CEt1, CEt2) of
        (CENat, CEReal) => true (* S-Base *)
        | (CENat, CENat) => true (* S-Nat *)
        | (CEReal, CEReal) => true (* S-Real *)
        | (CFun(t1,t2), CFun(t1',t2')) => (* S-Fun *)
          (subh t1' t1 (not isContra)) andalso
          (subh t2 t2' isContra)
        | (CESum(t1,t2,..), CESum(t1',t2',..)) => (* S-Sum *)
          (subh t1 t1' isContra) andalso
          (subh t2 t2' isContra)
        | (CEProd(t1,t2,..), CEProd(t1',t2',..)) => (* S-Prod *)
          (subh t1 t1' isContra) andalso
          (subh t2 t2' isContra)
        | (CEVar(m), CEVar(n)) => (* Return true if m is assumed to subtype n (S-Rec2);
          * otherwise, assume m subtypes n and return whether
          * m unrolled subtypes n unrolled (S-Rec1). *)
          Array2.sub(S,m,n) orelse (Array2.update(S,m,n,true);
          subh (Array.sub(UT1,m)) (Array.sub(UT2,n)) isContra)
      | _ => false
    end
  in subh CEt1 CEt2 false end;

Fig. 12. The main function of the optimized subtyping algorithm.
6. DISCUSSION
A few remaining points are worth discussing.

6.1. Evaluation Contexts vs. General Contexts in the Definition of Preciseness
Definition 1 is based on evaluation contexts $E$ rather than general (arbitrary-subexpression) contexts $G$. General contexts are the evaluation contexts used with the full-$\beta$ evaluation strategy. For example, $G$ is defined for $\lambda$ as follows.

$$G ::= [] | \text{succ}(G) | \text{neg}(G) | \lambda x: \tau. G | G(e) | e(G)$$

One may wish to consider an alternative definition of subtyping-relation preciseness, based on $G$ rather than $E$. The following proposition shows that preciseness according to Definition 1 implies preciseness according to this alternative version of Definition 1.

**Proposition 8.** Evaluation Preciseness Implies General Preciseness.
Let $L$ be a language that:

— is type safe,
— has a subtyping relation $\leq$ that’s precise according to Definition 1,
— allows the standard subsumption typing rule (T-SUBSUME in Figures 1 and 5), and
— obeys the standard variable-substitution lemma (Lemma 4).

Then $\leq$ is also precise according to the alternative version of Definition 1, in which evaluation context $E$ is replaced with general context $G$.

**Proof.** Using general contexts instead of evaluation contexts does not affect the proof of soundness (Lemma 6), which relies only on the existence of rule T-SUBSUME and the variable-substitution and type-safety lemmas. Hence, $\leq$ is sound according to the alternative version of Definition 1. Moreover, because $\leq$ is complete according to Definition 1, we have that if $\tau_1 \leq \tau_2$ isn’t derivable then there exist $E$, $\tau$, $e$, and $e'$ such that $E[\tau_2]: \tau$, $e: \tau_1$, $E[e] \Rightarrow^* e'$, and stuck($e'$). Because every $E$ is also a $G$, if $\tau_1 \leq \tau_2$ isn’t derivable then there exist $G$, $\tau$, $e$, and $e'$ such that $G[\tau_2]: \tau$, $e: \tau_1$, $G[e] \Rightarrow^* e'$, and stuck($e'$). Hence, $\leq$ is complete according to the alternative version of Definition 1.

Languages $\lambda$ and $L_{\mu, x}^-$ satisfy the requirements of Proposition 8 and are therefore precise according to the general-context version of Definition 1.

6.2. Subtyping with Strict vs. Nonstrict Evaluation Strategies
The evaluation strategy remains fixed in Proposition 8; the proposition does not imply that a subtyping relation that’s precise with one evaluation strategy will be precise with another. On the contrary, the choice of evaluation strategy may affect subtyping.

This paper has proved two subtyping relations precise, both in call-by-value languages (i.e., languages with strict evaluation). The completeness proofs have relied on the ability to “force” some unsafe computation to occur before performing unrelated, safe operations. This ability has been needed in exactly one subcase of each completeness proof: when the contravariant subtyping judgment for function arguments is undervariable.

Complications arise in nonstrict languages. As just eluded to, the complications relate to function-argument subtyping. For an example, let’s consider the call-by-name version of $\lambda$ from Section 3, called $\lambda_{CBN}$. In this call-by-name calculus, we could safely allow real $\rightarrow$ nat to be a subtype of $\tau \rightarrow$ nat, for all types $\tau$. Although such a rule would break type safety in the call-by-value version of $\lambda$, allowing real $\rightarrow$ nat to subtype $\tau \rightarrow$ nat cannot cause well-typed $\lambda_{CBN}$ programs to get stuck. It’s always safe to substitute a function $f$ of type real $\rightarrow$ nat in place of any function that returns a nat (or real) in $\lambda_{CBN}$ because it’s impossible for $f$ to actually compute its real-type argument.
expression. No primitive operations exist to convert a `real` into a `nat`, so there’s no way for `f` to use its argument to compute its result, and the call-by-name semantics prevents `f` from computing its argument expression just to “throw away” the result.

Subtyping in nonstrict languages thus depends on which primitives are present in the language, sometimes in non-orthogonal ways. For example, the subtyping rule for functions in $\lambda_{CBN}$ not only depends on how functions operate, but also on the types used and returned by the `succ` and `neg` operations. Suppose we added a new kind of expression to $\lambda_{CBN}$, called `floorAbs(e)`. Statically `floorAbs(e)` requires `e` to have type `real`; when it does, `floorAbs(e)` has type `nat`. Dynamically, if `e` evaluates to `r` (where `r` may be a real or natural number), then `floorAbs(e)` evaluates to the `n` such that `n = |r|`. With this new `floorAbs` operation, which on the surface has nothing to do with functions, we have to change the subtyping rule for function types, because it’s now unsound to allow `real` to `nat` to be a subtype of `τ`—`nat`; otherwise, the expression $(\lambda x:real.floorAbs(x))(\lambda z:nat.0)$ would be well typed but gets stuck. Again, without `floorAbs`, there’s no way for a function of type `real` to `nat` to get stuck, regardless of its actual argument, so precisely subtyping function types in $\lambda_{CBN}$ depends on the other operations available in the language.

Although it’s sound with respect to type safety to allow `real` to `nat` to subtype every type `τ`—`nat` in $\lambda_{CBN}$, such a subtyping violates the preservation property of $\lambda_{CBN}$. For example, if `real` is a subtype of `(nat→nat)`—`nat` then the expression $(\lambda x:real.(\lambda y:real.0)(\text{neg}(x)))(\lambda z:nat.0)$ has type `nat` but takes a step to $(\lambda x:real.0)(\text{neg}(\lambda z:nat.0))$, which is ill typed (but does not get stuck; getting stuck would be impossible per the discussion above). Hence, establishing type safety for the version of $\lambda_{CBN}$ that allows `real` to `nat` to subtype every type `τ`—`nat`—and such an allowance must be made for the subtyping relation to be complete—would require using some non-preservation-based technique.

Similar analysis would show the same complications with other nonstrict evaluation strategies, such as the full-\(\beta\) strategy. Due to these complications, it may be difficult in practice to design and maintain precise subtyping systems for large-scale, nonstrict languages.

6.3. Iso-recursive vs. Equi-recursive Subtyping

Finally, let’s compare this paper’s techniques for subtyping iso-recursive types with existing techniques for subtyping equi-recursive types. The most basic difference stems from the fact that types in equi-recursive systems are always equivalent to their rolled and unrolled versions, so for example $\mu t.nat \leq real$ in an equi-recursive, but not iso-recursive, system.

A second difference stems from how soundness and completeness have been defined in equi-recursive systems, versus how they’re defined here. In equi-recursive systems, a subtyping relation is sound and complete when $\tau_1 \leq \tau_2$ means that “infinitely unrolling” $\tau_1$ produces a type tree that’s a subtype of the tree produced by “infinitely unrolling” $\tau_2$. In contrast, this paper’s definitions of soundness and completeness are tied not to recursive types and unrolling, but to maximizing the subtyping relation without compromising type safety. This difference in definitions has a major impact on the subtyping rules: the rules here have to account for value-uninhabitation. Concretely, the rules here derive relationships like $\mu t.(t + t) \leq real \leq (\mu t.t)\rightarrow nat$, which are not generally derivable in equi-recursive systems. Because equi-recursive systems focus on type trees, they normally limit the syntax of recursive types to ensure that the trees are well formed; for example, equi-recursive types may have to have the form $\mu t.(\tau_1 \rightarrow \tau_2)$ rather than the more general form $\mu t.\tau$ [Im et al. 2013]. With such limitations, equi-recursive systems forbid from consideration not only all value-uninhabited types, but also infinitely many value-inhabited types (such as $(\mu t.t)\rightarrow nat$).
A third difference lies in the efficiency of known subtyping algorithms. Although the subtyping-helper function \texttt{subh} in Figure 12 is similar to Brandt and Henglein’s algorithm for subtyping equi-recursive types [Brandt and Henglein 1998], the most efficient known equi-recursive subtyping algorithms have \(O(n^2)\) running time [Kozen et al. 1995; Brandt and Henglein 1998], while this paper’s iso-recursive subtyping algorithm has \(O(mn)\) running time. The \(O(mn)\) bound is an improvement over \(O(n^2)\) because the \(m\) variable is independent from, and guaranteed to be smaller than, the \(n\) variable. In fact, we expect \(m\) to be much smaller than \(n\) in practice. For example, the code in Figures 10–12 defines two recursive types, \texttt{typ} and \texttt{CEtyp}. Recursive type \texttt{typ} has a size of at least 17 (its precise size depends on how variant types are represented) but only contains a single \(\mu\)-term. Similarly, recursive type \texttt{CEtyp} has \(n \geq 18\) but \(m=1\).

This paper’s subtyping-helper function \texttt{subh} runs in \(O(mn)\) time because every type in a \(\mu\)-term on one side of the \(\leq\) may be compared to at most one corresponding type in each of the other side’s \(\mu\)-terms. However, equi-recursive subtyping systems have to consider unrolling only one side of the \(\leq\) (e.g., because \(\mu.t.\text{nat}\leq\text{real}\)). This asynchronous unrolling affects the equi-recursive subtyping algorithm’s performance: every type in a \(\mu\)-term on one side of the \(\leq\) may be compared to at most one type \(\tau\) on the other side, where \(\tau\) is not limited to being a corresponding type within a \(\mu\)-term. Hence, asynchronous type unrolling causes the \(O(n^2)\) running time in existing equi-recursive subtyping algorithms. Because none of this paper’s techniques address asynchronous type unrolling, we believe that none of this paper’s techniques could be used to improve the \(O(n^2)\) bound for subtyping equi-recursive types.

REFERENCES


Hyeonseung Im, Keiko Nakata, and Sungwoo Park. 2013. Contractive Signatures with Recursive Types, Type Parameters, and Abstract Types. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*.


On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types


A. PROOF OF PRECISENESS FOR THE SUBTYPING RELATION IN \( L^{\mu}_{k,\times} \)

The following proof shows that the subtyping relation \( \leq \) defined in Figure 9 is precise with respect to type safety. Along the way, we’ll also see that \( \leq \) is reflexive and transitive, and that the val(\( \tau \))=\( \emptyset \) rules precisely decide whether \( \tau \) is value-uninhabited.

A.1. Basic Properties of the Value-Uninhabitation and Subtyping Relations

The proof begins with many “sanity checks” on the val and \( \leq \) relations (from Lemma 9 to Corollary 18). The first two lemmas are simple context-weakening results.

**Lemma 9.** Value-Uninhabitation Weakening.

\[ \forall U, \tau, U' \supseteq U : (U \vdash \text{val}(\tau) = \emptyset \Rightarrow U' \vdash \text{val}(\tau) = \emptyset) \]

**Proof.** By straightforward induction on the derivation of \( U \vdash \text{val}(\tau) = \emptyset \). □

**Lemma 10.** Subtype Weakening.

\[ \forall S, \tau_1, \tau_2, S' \supseteq S : (S \vdash \tau_1 \leq \tau_2 \Rightarrow S' \vdash \tau_1 \leq \tau_2) \]

**Proof.** By straightforward induction on the derivation of \( S \vdash \tau_1 \leq \tau_2 \). □

The next two lemmas show that properties of recursive types imply properties of their unrolled versions.

**Lemma 11.** Unrolled Value-Uninhabitation.

\[ \forall t, \tau : (\text{val}(\mu t.\tau)=\emptyset \Rightarrow \text{val}([\mu t.\tau]t)\tau)=\emptyset) \]

**Proof.** The only rule deriving \( \text{val}(\mu t.\tau)=\emptyset \) is U-REC1, so by inversion of that rule we have \( \{\mu t.\tau\} \vdash \text{val}([\mu t.\tau]t)\tau)=\emptyset \). Hence, by Lemma 9, for all \( U \) there exists
a derivation forest $D_U$ such that \[
\frac{D_U}{U \cup \{\mu_t.\tau\} \vdash \text{val}([\mu_t.\tau/t]\tau) = \emptyset}
\] is a valid derivation.

Now construct a new derivation forest $D' = D_\emptyset$, except that $D'$ (1) first removes all $\mu_t.\tau$ value-uninhabitation assumptions from $D_\emptyset$, and then (2) replaces all leaf-node judgments of the form $U \cup \{\mu_t.\tau\} \vdash \text{val}([\mu_t.\tau/t]\tau) = \emptyset$ in $D_\emptyset$ with the derivation tree
\[
\frac{D_U}{U \vdash \text{val}([\mu_t.\tau/t]\tau) = \emptyset}.
\]

Then $D'$ (in the case where $\text{val}([\mu_t.\tau/t]\tau) = \emptyset$) is a valid derivation tree because $D'$ derives as does $D_\emptyset$, but without requiring an initial $\mu_t.\tau$ value-uninhabitation assumption. □

**Lemma 12. Unrolled Subtyping.**

$$\forall t_1, t_2, \tau_1, \tau_2 : (\mu_t.\tau_1 \leq \mu_t.\tau_2 \Rightarrow [\mu_t.\tau_1/t]\tau_1 \leq [\mu_t.\tau_2/t]\tau_2)$$

**Proof.** Let $\tau_1 = \mu t.\tau_1$, $\tau_2 = \mu t.\tau_2$, $\tau_{1u} = [\tau_1/t]\tau_1$, and $\tau_{2u} = [\tau_2/t]\tau_2$. The only rules for deriving $\tau_1 \leq \tau_2$ are S-\bot and S-REC1. In the S-\bot case, $\text{val}(\tau_1) = \emptyset$, so by Lemma 11, $\text{val}(\tau_{1u}) = \emptyset$, implying by S-\bot that $\tau_{1u} \leq \tau_{2u}$, as required. In the S-REC1 case, we assume $(\tau_1 \leq \tau_2) \vdash \tau_{1u} \leq \tau_{2u}$, so by Lemma 10, for all $S$ there exists a derivation-forest $D_S$ such that $\frac{S \cup \{\tau_1 \leq \tau_2\} \vdash \tau_{1u} \leq \tau_{2u}}{S \vdash \tau_{1u} \leq \tau_{2u}}$. Then $D'$ (in the case where $\text{val}([\mu_t.\tau/t]\tau) = \emptyset$) is a valid derivation tree because $D'$ derives as does $D_\emptyset$, but without requiring an initial $\tau_1 \leq \tau_2$ subtyping assumption. □

**A.2. Correctness of the Value-Uninhabitation Rules**

Lemma 13 shows that subtypes of value-unhabited types must be value-unhabited. In other words, the bottom type truly is bottom.

**Lemma 13. Value-Uninhabitation is Closed Under Subtyping.**

$$\forall \tau_1, \tau_2, U : ((U \vdash \text{val}(\tau_1) = \emptyset \ not \ derivable \wedge \tau_1 \leq \tau_2) \Rightarrow (\text{val}(\tau_2) = \emptyset \ not \ derivable))$$

**Proof.** By induction on the failing derivation of $U \vdash \text{val}(\tau_1) = \emptyset$. In all cases, $U \vdash \text{val}(\tau_1) = \emptyset$ not being derivable implies by the contrapositive of Lemma 9 that $\text{val}(\tau_1) = \emptyset$ is not derivable, so $\tau_1 \leq \tau_2$ can’t be derived with rule S-\bot.

The leaf nodes in a failing derivation of $U \vdash \text{val}(\tau_1) = \emptyset$ occur when $\tau_1$ is a nat, real, or function type. In these cases $\tau_1 \leq \tau_2$ may be derived with rules S-BASE, S-NAT, S-REAL, S-T, or S-FUN, implying that $\tau_2$ must be nat, real, or a function type, so $\text{val}(\tau_2) = \emptyset$ is not derivable.

The inner nodes in a failing derivation of $U \vdash \text{val}(\tau_1) = \emptyset$ occur with the rules U-SUM, U-PROD2, and U-REC1 (derivations may not fail on rules U-PROD1 or U-REC2 because then rules U-PROD2 or U-REC1 would be used instead). We consider each of these three inductive cases.

\[
\begin{array}{c}
\text{Case} & U \vdash \text{val}(\tau_1) = \emptyset & U \vdash \text{val}(\tau_1') = \emptyset & U \vdash \text{val}(\tau_1''') = \emptyset & \text{U-SUM} \\
\end{array}
\]

By assumption, $\tau_1 = \tau_1' + \tau_1''$, $\tau_1 \leq \tau_2$, and at least one of the U-SUM premises is not derivable (i.e., $U \vdash \text{val}(\tau_1') = \emptyset$ is not derivable or $U \vdash \text{val}(\tau_1'') = \emptyset$ is not derivable). Because
$\tau_1 = \tau_1' + \tau_1''$, $\tau_1 \leq \tau_2$ may be derived with rule S-T or S-SUM. In the S-T subcase, $\tau_2$ is a function type, so $\text{val}(\tau_2) = \emptyset$ is not derivable, as required. In the S-SUM subcase, we have $\tau_2 = \tau_2' + \tau_2''$, $\tau_1' \leq \tau_2'$, and $\tau_1'' \leq \tau_2''$. Because $\tau_1' \leq \tau_2'$, $\tau_1'' \leq \tau_2''$, and at least one of the U-SUM premises is underivable, the inductive hypothesis implies that $\text{val}(\tau_2') = \emptyset$ is not derivable or $\text{val}(\tau_2'') = \emptyset$ is not derivable. From U-SUM, then, $\text{val}(\tau_2) = \emptyset$ is not derivable, as required.

\[
\begin{array}{c}
\text{Case} \quad U \vdash \text{val}(\tau_1') = \emptyset \\
U \vdash \text{val}(\tau_1' + \tau_1'') = \emptyset \\
\text{U-PROD2}
\end{array}
\]

In this case, $\tau_1 = \tau_1' + \tau_1''$, $\tau_1 \leq \tau_2$, U-PROD2 fails due to $U \vdash \text{val}(\tau_1') = \emptyset$ not being derivable, and U-PROD1 has already failed due to $U \vdash \text{val}(\tau_1') = \emptyset$ not being derivable. Because $\tau_1 = \tau_1' \times \tau_1''$, $\tau_1 \leq \tau_2$ may be derived with rule S-T or S-PROD. In the S-T subcase, $\tau_2$ is a function type, so $\text{val}(\tau_2) = \emptyset$ is not derivable, as required. In the S-PROD subcase, we have $\tau_2 = \tau_2' \times \tau_2''$, $\tau_1' \leq \tau_2'$, and $\tau_1'' \leq \tau_2''$. Because $\tau_1' \leq \tau_2'$, $\tau_1'' \leq \tau_2''$, $U \vdash \text{val}(\tau_1') = \emptyset$ is not derivable, and $U \vdash \text{val}(\tau_1'') = \emptyset$ is not derivable, the inductive hypothesis implies that $\text{val}(\tau_2') = \emptyset$ is not derivable and $\text{val}(\tau_2'') = \emptyset$ is not derivable. From U-PROD1 and U-PROD2, then, $\text{val}(\tau_2) = \emptyset$ is not derivable.

\[
\begin{array}{c}
\text{Case} \quad U \cup \{\mu_{\tau_1}.\tau_1\} \vdash \text{val}(\mu_{\tau_1}.\tau_1/t_1) = \emptyset \\
U \vdash \text{val}(\mu_{\tau_1}.\tau_1) = \emptyset \\
\text{U-REC1}
\end{array}
\]

Let $\tau_{1u} = [\mu_{\tau_1}.\tau_1/t_1]\tau_1$. In this case, $\tau_1 = \mu_{\tau_1}.\tau_1$, $\tau_1 \leq \tau_2$, and U-REC1 fails due to $U \cup \{\tau_1\} \vdash \text{val}(\tau_{1u}) = \emptyset$ not being derivable. Because $\tau_1 = \mu_{\tau_1}.\tau_1$, $\tau_1 \leq \tau_2$ may be derived with rule S-T or S-REC1. In the S-T subcase, $\tau_2$ is a function type, so $\text{val}(\tau_2) = \emptyset$ is not derivable, as required. In the S-REC1 subcase, we have $\tau_2 = \mu_{\tau_2}.\tau_2$, so let $\tau_{2u} = [\mu_{\tau_2}.\tau_2/t_2]\tau_2$; then because $\tau_1 \leq \tau_2$, Lemma 12 provides that $\tau_{1u} \leq \tau_{2u}$. Given that $\tau_{1u} \leq \tau_{2u}$ and $U \cup \{\tau_1\} \vdash \text{val}(\tau_{1u}) = \emptyset$ is not derivable, the inductive hypothesis implies that $\text{val}(\tau_{2u}) = \emptyset$ is not derivable, so by the contrapositive of Lemma 11, $\text{val}(\tau_2) = \emptyset$ is not derivable. \(\square\)

Now we can prove that the val judgment means what we want it to mean: $\text{val}(\tau) = \emptyset$ exactly when there exists no value of type $\tau$.

**Lemma 14.** Value-Uninhabitation.

\[\forall \tau : (\text{val}(\tau) = \emptyset \iff \nexists v : (v:\tau))\]

**Proof.** We first prove that, for all $U$ and $\tau$, if $U \vdash \text{val}(\tau) = \emptyset$ is not derivable, then $\nexists v : (v:\tau)$. The contrapositive of the lemma's if-direction ($\Leftarrow$) follows as a result. The proof is by induction on the failing derivation of $U \vdash \text{val}(\tau) = \emptyset$, which can only have leaf nodes when $\tau$ is nat, real1, or $\tau_1 \rightarrow \tau_2$. In every one of these base cases, there exists a $v$ such that $v:\tau$ (when $\tau$ is nat or real1, let $v$ be 0, and when $\tau = \tau_1 \rightarrow \tau_2$, let $v$ be $f(x)$ such that $f(x) : \tau_1 \rightarrow \tau_2$, for $f(x) : ((\text{fun} g(y:\text{nat}) : \tau_2 = f(x)))(0)$)). The inductive cases of a failing value-uninhabitation derivation occur with rules U-SUM, U-PROD2, and U-REC1. In the U-REC1 case, $\tau = \mu\tau.\tau$ and the inductive hypothesis implies that there exists a $v'$ such that $v' : [\mu\tau].\tau/\tau$; let $v = \text{eval}(v')$ to ensure that $v:\tau$. The U-SUM and U-PROD2 cases are handled similarly (but instead of $v$ being a rolled subvalue, it’s either an injection of a subvalue or a pair of subvalues).

We next prove that, for all $v$ and $\tau$, if $v:\tau$ then $\text{val}(\tau) = \emptyset$ is not derivable. The contrapositive of the lemma’s only-if-direction ($\Rightarrow$) follows as a result. The proof is by induction on the derivation of $v:\tau$. The rules for deriving $v:\tau$ are T-NAT, T-REAL, T-FUN, T-LEFT, T-RIGHT, T-PROD, T-ROLL, and T-SUBSUME.

--- Cases T-NAT, T-REAL, T-FUN: Here $\tau$ is nat, real1, or a function type, so $\text{val}(\tau) = \emptyset$ is not derivable.
— Case T-LEFT: Here $\tau = \tau_1 + \tau_2$, $v = inl, v_1$, and $v_1: \tau_1$. By the inductive hypothesis, $\text{val}(\tau_1) = \emptyset$ is not derivable, so by rule U-SUM, $\text{val}(\tau_1 + \tau_2) = \emptyset$ is not derivable.

— Case T-RIGHT: This case is similar to the previous one.

— Case T-PROD: Here $\tau = \tau_1 \times \tau_2$, $v = (v_1, v_2)$, $v_1: \tau_1$, and $v_2: \tau_2$. By the inductive hypothesis, $\text{val}(\tau_1) = \emptyset$ is not derivable and $\text{val}(\tau_2) = \emptyset$ is not derivable, so by rules U-PROD1 and U-PROD2, $\text{val}(\tau_1 \times \tau_2) = \emptyset$ is not derivable.

— Case T-ROLL: Here $\tau = \mu t. \tau$, $v = \text{roll}(v')$, and $v': [\mu t. \tau/t] \tau$. Let $\tau_u = [\mu t. \tau/t] \tau$, so we have $v': \tau_u$. By the inductive hypothesis, $\text{val}(\tau_u) = \emptyset$ is not derivable, so by the contrapositive of Lemma 11, $\text{val}(\tau) = \emptyset$ is not derivable.

— Case T-SUBSUME: Here $\forall \tau$ and $\forall \tau' \leq \tau$. By the inductive hypothesis, $\text{val}(\tau') = \emptyset$ is not derivable, so by Lemma 13, $\text{val}(\tau) = \emptyset$ is not derivable.

In all cases, $\text{val}(\tau) = \emptyset$ is underviable, as required. □

### A.3. Subtyping Reflexivity and Transitivity

For the sake of determinism, the subtyping relation in $L_{\alpha, \chi}^{-\mu}$ lacks explicit reflexive and transitive rules. The following lemmas show that the subtyping relation is nonetheless reflexive and transitive.

**Lemma 15. Strong Subtyping Reflexivity.**

$\forall S, \tau_1, \tau_2 : (S \vdash \tau_1 \leq \tau_2 \text{ not derivable } \Rightarrow \tau_1 \neq \tau_2)$

**Proof.** By induction on the failing derivation of $S \vdash \tau_1 \leq \tau_2$. Base cases occur when $\tau_1 = \text{real}$ and $\tau_2 = \text{nat}$, or when exactly one of $\tau_1$ and $\tau_2$ is a function/product/sum/recurutive type. In all these base cases, $\tau_1 \neq \tau_2$, as required.

Inductive cases of a failing derivation of $S \vdash \tau_1 \leq \tau_2$ occur with rule S-FUN, S-SUM, S-PROD, or S-REC1. In all these inductive cases, the failing premise must be of the form $S' \vdash \tau'_1 \leq \tau'_2$, so the inductive hypothesis implies that $\tau'_1 \neq \tau'_2$, which guarantees that $\tau_1 \neq \tau_2$. For example, the inductive hypothesis in the S-REC1 case implies that unrolled types are unequal, which guarantees that the rolled types must be unequal. □

Lemma 16 provides a standard subtyping-inversion result, though the result is complicated by consideration of value-uninhabitation.

**Lemma 16. Subtyping Inversion.**

$\forall S, \tau_1, \tau_2 : \text{If } S \vdash \tau_1 \leq \tau_2, \text{ then}$

A. $\text{val}(\tau_1) = \emptyset$, or

B. $\text{val}(\tau_1) = \emptyset$ is underviable, $\tau_2 = \tau'_2 \rightarrow \tau''_2$, and $\text{val}(\tau'_2) = \emptyset$, or

C. Neither A nor B hold, and all of the following hold:

i. $\tau_1 = \text{real} \Rightarrow (\tau_2 = \text{real})$

ii. $\tau_1 = \text{nat} \Rightarrow (\tau_2 = \text{real} \lor \tau_2 = \text{nat})$

iii. $\tau_1 = \tau'_2 \rightarrow \tau''_2 \Rightarrow (\tau_2 = \tau'_2 \rightarrow \tau''_2 \land S \vdash \tau'_2 \leq \tau'_1 \land S \vdash \tau''_2 \leq \tau''_1)$

iv. $\tau_1 = \tau'_2 + \tau''_2 \Rightarrow (\tau_2 = \tau'_2 + \tau''_2 \land S \vdash \tau'_2 \leq \tau'_1 \land S \vdash \tau''_2 \leq \tau''_1)$

v. $\tau_1 = \tau'_2 \times \tau''_2 \Rightarrow (\tau_2 = \tau'_2 \times \tau''_2 \land S \vdash \tau'_2 \leq \tau'_1 \land S \vdash \tau''_2 \leq \tau''_1)$

vi. $\tau_1 = \mu t. \tau \Rightarrow (\tau_2 = \mu t'. \tau' \land \text{and either } \tau_1 \leq \tau_2 \in S \text{ or } S \cup \{\tau_1 \leq \tau_2\} \vdash [\mu t. \tau/t] \tau \leq [\mu t'. \tau'/t'] \tau')$

vii. $\tau_2 = \text{real} \Rightarrow (\tau_1 = \text{nat} \lor \tau_1 = \text{real})$

viii. $\tau_2 = \text{nat} \Rightarrow (\tau_1 = \text{nat})$

ix. $\tau_2 = \tau'_2 \rightarrow \tau''_2 \Rightarrow (\tau_1 = \tau'_2 \rightarrow \tau''_2 \land S \vdash \tau'_2 \leq \tau'_1 \land S \vdash \tau''_2 \leq \tau''_1)$

x. $\tau_2 = \tau'_2 \rightarrow \tau''_2 \Rightarrow (\tau_1 = \tau'_2 + \tau''_2 \land S \vdash \tau'_2 \leq \tau'_1 \land S \vdash \tau''_2 \leq \tau''_1)$

xi. $\tau_2 = \tau'_2 \rightarrow \tau''_2 \Rightarrow (\tau_1 = \tau'_2 \times \tau''_2 \land S \vdash \tau'_2 \leq \tau'_1 \land S \vdash \tau''_2 \leq \tau''_1)$

xii. $\tau_2 = \mu t. \tau \Rightarrow (\tau_1 = \mu t. \tau \land \text{and either } \tau_1 \leq \tau_2 \in S \text{ or } S \cup \{\tau_1 \leq \tau_2\} \vdash [\mu t. \tau/t] \tau \leq [\mu t'. \tau'/t'] \tau')$

**Proof.** By straightforward case analysis of the rules deriving $S \vdash \tau_1 \leq \tau_2$. □
LEMMA 17. Strong Subtyping Transitivity.

\[ \forall S, \tau_1, \tau_2, \tau_3 : ((S \vdash \tau_1 \leq \tau_3 \text{ not derivable} \land \tau_1 \leq \tau_2) \Rightarrow (\tau_2 \leq \tau_3 \text{ not derivable})) \]

PROOF. By induction on the failing derivation of \( S \vdash \tau_1 \leq \tau_3 \). Note that because \( S \vdash \tau_1 \leq \tau_3 \) is underviable, \( \text{val}(\tau_1) = \emptyset \) is underviable (by rule \( S \bot \)), so by Lemma 13, \( \text{val}(\tau_3) = \emptyset \) is underviable. Also because \( S \vdash \tau_1 \leq \tau_3 \) is underviable, if \( \tau_3 = \tau_3' \vdash \tau_3'' \) then \( \text{val}(\tau_3') = \emptyset \) is underviable (by rule \( S \top \)). Now suppose that \( \tau_2 = \tau_2' \vdash \tau_2'' \) and \( \text{val}(\tau_2') = \emptyset \); then the only rules for deriving \( \tau_2 \leq \tau_3 \) would be \( S \bot, S \top, \) and \( S \text{-FUN} \); however, \( S \bot \) can’t apply because \( \text{val}(\tau_2') = \emptyset \) is underviable, \( S \top \) can’t apply because if \( \tau_3 = \tau_3' \vdash \tau_3'' \) then \( \text{val}(\tau_3') = \emptyset \) is underviable, and \( S \text{-FUN} \) can’t apply because it would violate Lemma 13 to have \( \tau_3' \leq \tau_3'' \) and \( \text{val}(\tau_3') = \emptyset \) when \( \text{val}(\tau_3') = \emptyset \) is underviable. It’s therefore impossible to derive \( \tau_2 \leq \tau_3 \) when \( \tau_2 = \tau_2' \vdash \tau_2'' \) and \( \text{val}(\tau_2') = \emptyset \).

We now have that (1) \( \text{val}(\tau_1) = \emptyset \) is underviable, (2) \( \text{val}(\tau_2) = \emptyset \) is underviable, (3) if \( \tau_2 = \tau_2' \vdash \tau_2'' \) then \( \text{val}(\tau_2') = \emptyset \) is underviable, and (4) if \( \tau_3 = \tau_3' \vdash \tau_3'' \) then \( \text{val}(\tau_3') = \emptyset \) is underviable. In other words, neither \( \tau_1 \) nor \( \tau_2 \) is \( \bot \), and neither \( \tau_2 \) nor \( \tau_3 \) is \( \top \). The cases below therefore ignore these possibilities.

The base cases in a failing derivation of \( S \vdash \tau_1 \leq \tau_3 \) occur when either \( \tau_1 = \text{real} \) and \( \tau_3 = \text{nat} \) or exactly one of \( \tau_1 \) and \( \tau_3 \) is a function/product/sum/recursive type. If \( \tau_1 = \text{real} \) and \( \tau_3 = \text{nat} \), then \( \tau_2 \leq \tau_3 \) is underviable because Lemma 16 (applied to \( \tau_1 \leq \tau_2 \)) ensures that \( \tau_2 = \text{real} \). If exactly one of \( \tau_1 \) and \( \tau_3 \) is a function/product/sum/recursive type, then \( \tau_2 \leq \tau_3 \) is again underviable because otherwise, with \( \tau_1 \leq \tau_2 \) and \( \tau_2 \leq \tau_3 \), Lemma 16 would ensure that both \( \tau_1 \) and \( \tau_3 \) are the same “kind” of type (i.e., both numeric/function/product/sum/recursive types).

The inductive cases in a failing derivation of \( S \vdash \tau_1 \leq \tau_3 \) occur with rules \( S \text{-FUN}, S \text{-SUM}, S \text{-PROD}, \) and \( S \text{-REC1} \). In the \( S \text{-FUN} \) case we assume that \( \tau_1 = \tau_1' \vdash \tau_1'', \tau_3 = \tau_3' \vdash \tau_3'' \), and at least one of \( S \vdash \tau_1' \leq \tau_1'' \) and \( S \vdash \tau_3' \leq \tau_3'' \) are underviable. Because \( \tau_1 \leq \tau_2 \), Lemma 16 ensures that \( \tau_2 = \tau_2' \vdash \tau_2'' \), \( \tau_2 \leq \tau_1'' \) and \( \tau_2 \leq \tau_2'' \). The inductive hypothesis then implies that at least one of \( \tau_1' \leq \tau_1'' \) and \( \tau_2' \leq \tau_2'' \) is underviable, so \( \tau_2 \leq \tau_3 \) is underviable (by rule \( S \text{-FUN} \)). The proofs of the \( S \text{-SUM} \) and \( S \text{-PROD} \) cases are similar.

In the \( S \text{-REC1} \) case we assume that \( \tau_1 = \mu \tau_1.\tau_1, \tau_3 = \mu \tau_3.\tau_3, \) and \( \{ \tau_1 \leq \tau_3 \} \vdash \tau_1 \leq \tau_3 \) is underviable. Because \( \tau_1 \leq \tau_2 \), Lemmas 16 and 12 ensure that \( \tau_2 = \mu \tau_2.\tau_2 \) and \( \tau_1 \leq \tau_2 \) (where \( \tau_2 = \tau_2./\tau_2.\tau_2 \)). The inductive hypothesis then implies that \( \tau_2 \leq \tau_3 \) is underviable, so by the contrapositive of Lemma 12, \( \tau_2 \leq \tau_3 \) is underviable.

COROLLARY 18. \( \leq \) is a Preorder.

The subtyping relation is reflexive and transitive.

PROOF. Immediate by the contrapositives of Lemmas 15 and 17. \( \square \)

A.4 Properties of the Static and Dynamic Semantics

Having completed the “sanity checks” on the \( \leq \) and \( \text{val} \) relations, Lemmas 19–21 present standard weakening, variable-substitution, and canonical-forms lemmas, which are used to prove subtyping completeness and soundness.


\[ \forall \Gamma, e, \tau, \tau' : \Gamma \vdash e : \tau \Rightarrow \Gamma' \vdash e : \tau' \]

PROOF. By induction on the derivation of \( \Gamma \vdash e : \tau \). \( \square \)

LEMMA 20. Variable Substitution.

\[ \forall \Gamma, x, \tau', e, \tau, e' : ((\Gamma \cup \{ x : \tau' \}) \vdash e : \tau \land \Gamma \vdash e' : \tau') \Rightarrow \Gamma \vdash [e'/x]e : \tau) \]

PROOF. By induction on the derivation of \( \Gamma \cup \{ x : \tau' \} \vdash e : \tau \). \( \square \)

ACM Journal Name, Vol. V, No. N, Article A, Publication date: January YYYY.

\[ \forall v, \tau : \text{If } v: \tau \text{ then} \]

A. \( \tau = \text{nat} \Rightarrow v = n \) (for some \( n \))
B. \( \tau = \text{real} \Rightarrow v = n \text{ or } v = x \) (for some \( n \) or \( x \))
C. \( (\tau = \tau_1 \land \mathop{\text{val}}(\tau_1) = \emptyset \text{ not derivable}) \Rightarrow v = (\text{fun } f.(x:\tau_3):\tau_4 = e) \) (for some \( f, x, \tau_3, \tau_4, e \))
D. \( \tau = \tau_1 \lor \tau_2 \Rightarrow v = \text{inl}_r v' \) or \( v = \text{inr}_r v' \) (for some \( \tau' \) and \( v' \))
E. \( \tau = \tau_1 \times \tau_2 \Rightarrow v = (v_1, v_2) \) (for some \( v_1 \) and \( v_2 \))
F. \( \tau = \mu t. \tau \Rightarrow v = \text{roll}(v') \) (for some \( v' \))

PROOF. By induction on the derivation of \( v: \tau \). The only nontrivial case is \text{T-SUBSUME}, in which \( v: \tau' \), \( v: \tau \), and \( \tau \leq \tau' \). Because \( v: \tau' \), Lemma 14 ensures that \( \mathop{\text{val}}(\tau') = \emptyset \) is undervisible. We next consider each of the six cases in the lemma statement and show that the desired result holds in every case. If \( \tau = \text{nat} \) then by Lemma 16, \( \tau' = \text{nat} \), so by the inductive hypothesis (applied to \( v: \tau' \)), \( v = n \). If \( \tau = \text{real} \) then by Lemma 16, \( \tau' = \text{nat} \) or \( \tau' = \text{real} \), so by the inductive hypothesis, \( v = n \) or \( v = x \). If \( \tau = \tau_1 \lor \tau_2 \) and \( \mathop{\text{val}}(\tau_1) = \emptyset \) is not derivable, then by Lemma 16, \( \tau' = \tau_1 \lor \tau_2 \) and \( \tau \leq \tau_1' \). Because \( \mathop{\text{val}}(\tau_1) = \emptyset \) is undervisible and \( \tau_1 \leq \tau_1' \), Lemma 13 ensures that \( \mathop{\text{val}}(\tau_1') = \emptyset \) is undervisible. Then applying the inductive hypothesis to \( v: \tau' \), \( \tau' = \tau_1' \lor \tau_2' \), \( v = \text{inl}_r v' \) or \( v = \text{inr}_r v' \). The remaining cases of \( \tau = \tau_1 \times \tau_2 \) and \( \tau = \mu t. \tau \) are proved similarly. \( \square \)

A.5. Subtyping Completeness

We’re now ready to state and prove the key lemma used to show completeness, Lemma 22. As in \( \lambda \), we first prove a slightly stronger version of the desired completeness result. Also as in \( \lambda \), the proof of Lemma 22 is constructive (in part because the proof of Lemma 14 is constructive).

LEMMA 22. Strong Completeness.

\[ \forall S, \tau_1, \tau_2 : (S \vdash \tau_1 \leq \tau_2 \text{ not derivable}) \Rightarrow \exists E, \tau, v, e : (E[\tau_2] : \tau \land \mathop{\text{v}} : \tau_1 \land E[v] \rightarrow^* e \land \mathop{\text{stuck}}(e)) \]

PROOF. The proof is by induction on the failing derivation of \( S \vdash \tau_1 \leq \tau_2 \). The base cases occur when \( \tau_1 = \text{real} \) and \( \tau_2 = \text{nat} \), or when exactly one of \( \tau_1 \) and \( \tau_2 \) is a function/product/sum/recursive type and \( \mathop{\text{val}}(\tau_1) = \emptyset \) is not derivable and \( \tau_2 \) is not a function type with value-uninhabited argument type; otherwise we’d have \( S \vdash \tau_1 \leq \tau_2 \) by \( S \bot \) or \( S \bot \). We first prove the lemma for these base cases.

Case \( \tau_1 = \text{real} \) and \( \tau_2 = \text{nat} \):
This case’s proof is the same as in the proof of Lemma 7.

Case \( \tau_1 = \tau_1' \lor \tau_1'' \) and \( \tau_2 = \tau_2' \lor \tau_2'' \):
Let \( v = (\text{fun } f.(x:\tau_3') : \tau_4' = f(x)) \), so \( v: \tau_1 \). Define \( E \) and \( \tau \) as follows:

\[
E = \begin{cases}
\text{neg}([]) & \text{if } \tau_2 = \text{nat or } \tau_2 = \text{real} \\
\text{case } [x'] \Rightarrow 2.718 \text{ else } \text{inr } y \Rightarrow 2.718 & \text{if } \tau_2 = \tau_2' + \tau_2'' \\
\text{[]}.\text{snd} & \text{if } \tau_2 = \tau_2' \times \tau_2'' \\
\text{unroll}([]) & \text{if } \tau_2 = \mu t. \tau \\
\text{if } \tau_2 = \text{nat or } \tau_2 = \text{real or } \tau_2 = \tau_2' + \tau_2'' & \text{real} \\
\text{if } \tau_2 = \tau_2' \times \tau_2'' & \tau_2' \\
\text{if } \tau_2 = \mu t. \tau & \mu t. \tau
\end{cases}
\]

Then \( E[\tau_2] : \tau_2 \), by the definitions of \( E \) and \( \tau \) and the typing rules. Moreover, let \( e = E[v] \), so \( E[v] \rightarrow^* e \) and \( \text{stuck}(e) \) (because \( \text{stuck}(E[v]) \), where \( v = (\text{fun } f.(x:\tau'_1) : \tau''_1 = f(x)) \)).
Case $\tau_1 \neq \tau'_1 \rightarrow \tau''_1$, $\tau_2 = \tau'_2 \rightarrow \tau''_2$, and both $\text{val}(\tau_1) = \emptyset$ and $\text{val}(\tau_2) = \emptyset$ are underivable (otherwise the derivation would use $S \bot$ or $S \top$ and not fail):

By Lemma 14 there exist $v$ and $v'_2$ such that $v : \tau_1$ and $v'_2 : \tau'_2$. Because $\tau_1 \neq \tau'_1 \rightarrow \tau''_1$ and $v : \tau_1$, Lemma 21 implies that $v$ cannot be a function value. Let $E = [ ]_{(v'_2)} = \tau_2$, and $e = v(v'_2)$. Then $E[\tau_2] : \tau$ (because $\tau_2 = \tau'_2 \rightarrow \tau''_2$, $v'_2 : \tau'_2$, and $\tau = \tau''_2$). Moreover, $E[v] = e$, so $E[v] \vdash^* e$, and stuck(e) (because $e = v(v'_2)$, where $v$ cannot be a function value).

Case $\tau_1 = \mu t_1. \tau_1$, $\tau_2 \neq \mu t_2. \tau_2$, $\text{val}(\tau_1) = \emptyset$ is underivable (otherwise the derivation would use $S \bot$ and not fail), and if $\tau_2 = \tau'_2 \rightarrow \tau''_2$ then $\text{val}(\tau'_2) = \emptyset$ is underivable (otherwise the derivation would use $S \top$ and not fail):

By Lemma 14 there exists a $v$ such that $v : \mu t_1. \tau_1$. Hence, by Lemma 21, $v = \text{roll}(v')$ for some value $v'$. Also by Lemma 14, if $\tau_2 = \tau'_2 \rightarrow \tau''_2$ then there exists a $v'_2$ such that $v'_2 : \tau'_2$.

Now define $E$ and $\tau$ as follows:

\[
E = \begin{cases} 
\text{neg}([ ]) & \text{if } \tau_2 = \text{nat or } \tau_2 = \text{real} \\
\text{case } \text{of } \text{inl } x \mapsto 2.718 \text{ else } \text{inr } y \mapsto 2.718 & \text{if } \tau_2 = \tau'_2 + \tau''_2 \\
[ ] \text{snd } (v'_2) & \text{if } \tau_2 = \tau'_2 \times \tau''_2 \\
[ ] & \text{if } \tau_2 = \tau'_2 \rightarrow \tau''_2 
\end{cases}
\]

\[
\tau = \begin{cases} 
\text{real } & \text{if } \tau_2 = \text{nat or } \tau_2 = \text{real or } \tau_2 = \tau'_2 + \tau''_2 \\
\tau'_2 & \text{if } \tau_2 = \tau'_2 \times \tau''_2 \text{ or } \tau_2 = \tau'_2 \rightarrow \tau''_2 
\end{cases}
\]

Then $E[\tau_2] : \tau$, by the definitions of $E$ and $\tau$ and the typing rules. Moreover, let $e = E[v]$, so $E[v] \vdash^* e$ and stuck(e) (because stuck($E[v]$), where $v = \text{roll}(v')$).

Case $\tau_1 = \mu t_1. \tau_1$, $\tau_2 = \mu t_2. \tau_2$, and $\text{val}(\tau_1) = \emptyset$ is underivable (otherwise the derivation would use $S \bot$ and not fail):

There are two subcases to consider, either (1) $\tau_1 = \tau'_1 \rightarrow \tau''_1$ and $\text{val}(\tau'_1) = \emptyset$ or (2) not. In subcase (1), let $v = 0$, so $\text{v:nat}$ and $\text{v:\tau_1}$ by T-SUBSUME and S-T. In subcase (2), Lemma 14 guarantees a $v$ such that $v : \tau_1$, and Lemma 21 guarantees that $v \neq \text{roll}(v')$ (for all $v'$). Hence, in all subcases, $v : \tau_1$ and $v \neq \text{roll}(v')$ (for all $v'$). Now let $E = \text{unroll}([ ]), \tau = [ \mu t_2. \tau_2 / t_2 ] \tau_2$, and $e = \text{unroll}(v)$. Then $E[\tau_2] : \tau$ by T-CTX, T-VAR, and T-UNROLL. Moreover, $E[v] = e$, so $E[v] \vdash^* e$, and stuck(e) (because $e = \text{unroll}(v)$, where $v$ can’t be a rolled value).

The remaining base cases (where exactly one of $\tau_1$ and $\tau_2$ is a product/sum type) are proved similarly.

The inductive cases of a failing derivation of $S \vdash \tau_1 \leq \tau_2$ occur with rules S-FUN, S-SUM, S-PROD, and S-REC1. In all these cases we can again assume that $\text{val}(\tau_1) = \emptyset$ is not derivable and if $\tau_2 = \tau'_2 \rightarrow \tau''_2$ then $\text{val}(\tau'_2) = \emptyset$ is not derivable; otherwise rule $S \bot$ or $S \top$ would be used to derive $S \vdash \tau_1 \leq \tau_2$.

\[
\frac{S \vdash \tau'_2 \leq \tau'_1 \quad S \vdash \tau''_1 \leq \tau''_2} {S \vdash \tau'_1 \rightarrow \tau''_1 \leq \tau'_2 \rightarrow \tau''_2} \text{ S-FUN}
\]

This case’s proof almost matches that given for Lemma 7. All the logic remains the same, with only two nontrivial differences: (1) in the first subcase in Lemma 7, we set $v = \lambda x : \tau'_1. ((\lambda y : \tau' : v'_1)(E'[x]))$, but here we set $v = (\text{fun } f(x : \tau'_1) : \tau''_1 = ((\text{fun } g(y : \tau' : v'_2)(E'[x]))$, and (2) in the second subcase in Lemma 7, we obtained a $v'_2 : \tau'_2$ by Lemma 3, but here we obtain the same by Lemma 14 and the assumption that $\text{val}(\tau'_2) = \emptyset$ is not derivable.
Let \( \tau_{1u} = [\mu t_1, \tau_1 / t_1] \tau_1 \) and \( \tau_{2u} = [\mu t_2, \tau_2 / t_2] \tau_2 \). By the inductive hypothesis, there exist \( E', \tau', v', \) and \( e' \) such that \( E'[\tau_{2u}]: \tau', v' : \tau_{1u}, E'[v'] \vdash e' \), and stuck\( (e') \). Let \( v = \text{roll}(v') \), \( E = E'[\text{unroll}([], [])] \), \( \tau = \tau' \), and \( e = e' \). Observe that by rule T-ROLL, \( v : \tau_1 \). Also, \( E'[\tau_{2u}]: \tau' \) means that \( \{ x' : \tau_{2u} \} \vdash E'[x'] : \tau' \), which implies by Lemma 19 that \( \{ x : \tau_2, x' : \tau_{2u} \} \vdash E'[x'] : \tau' \); then because \( \{ x : \tau_2 \} \vdash \text{unroll}(x) : \tau_{2u} \), Lemma 20 ensures that \( \{ x : \tau_2 \} \vdash E'[\text{unroll}(x)] : \tau' \), which means that \( E'[\text{unroll}(\tau_{2u})] : \tau' \). Hence, \( E[\tau_2] : \tau \). Also, by the definitions of \( E \) and \( v \), we have \( E[v] = E'[\text{unroll}((\text{roll}(v')))] \), so \( E[v] \vdash E'[v'] \), where \( E'[v'] \vdash e' \). Thus, because \( e' = e \) and stuck\( (e') , E[v] \vdash e \), and stuck\( (e) \), as required.

The remaining inductive cases (S-PROD and S-SUM) are proved similarly. The S-PROD case constructs \( v \) as a pair expression and uses a \( \text{fst} \) or \( \text{snd} \) expression to eliminate the pair in \( E \). The S-SUM case constructs \( v \) as an \( \text{inl} \) or \( \text{inr} \) expression and uses a \text{case} expression to eliminate the injection in \( E \). □

Having proved a stronger version of completeness in Lemma 22, the weaker version follows as a corollary.

**COROLLARY 23. Completeness.**

\( \forall \tau_1, \tau_2 : (\tau_1 \leq \tau_2) \Rightarrow \exists E, \tau, e, e' : (E[\tau_2] : \tau \wedge e : \tau_1 \wedge E[e] \vdash e' \wedge \text{stuck}(e')) \)

**PROOF.** By Lemma 22, if \( \tau_1 \leq \tau_2 \) is not derivable then there exist \( E, \tau, e, \) and \( e' \) such that \( E[\tau_2] : \tau, e : \tau_1, E[e] \vdash e' \), and stuck\( (e') \). The corollary is the contrapositive of this result. □

### A.6. Subtyping Soundness

With completeness proved, we move on to proving the soundness of the subtyping relation using type-safety lemmas. Lemmas 24–26 are used to prove Preservation (Lemma 27), while Lemma 28 is used to prove Progress (Lemma 29).

**LEMMA 24. Typing Inversion.**

A. \( \Gamma \vdash \text{nat} : \tau \Rightarrow (\text{nat} \leq \tau) \)

B. \( \Gamma \vdash \text{real} : \tau \Rightarrow (\text{real} \leq \tau) \)

C. \( \Gamma \vdash \text{succ} : \tau \Rightarrow (\Gamma \vdash \text{nat} \wedge \text{nat} \leq \tau) \)

D. \( \Gamma \vdash e_1, e_2 : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e_1 : \tau_1 \wedge \Gamma \vdash e_2 : \tau_2 \wedge \tau_1 \times \tau_2 \leq \tau) \)

E. \( \Gamma \vdash \text{neg} : \tau \Rightarrow (\Gamma \vdash \text{real} \wedge \text{real} \leq \tau) \)

F. \( \Gamma \vdash \text{fun} (f; x : \tau_1 : \tau_2 : \tau : e) : \tau \Rightarrow (\Gamma \{ f : \tau_1 \to \tau_2, x : \tau_1 \} \vdash e : \tau_2 \wedge \tau_1 \to \tau_2 \leq \tau) \)

G. \( \Gamma \vdash e_1, e_2 : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e_1 : \tau_1 \to \tau_2 \wedge \Gamma \vdash e_2 : \tau_1 \wedge \tau_2 \leq \tau) \)

H. \( \Gamma \vdash \text{inl} : e : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e : \tau_1 \wedge \tau' \to \tau_2 \to \tau_1 + \tau_2 \leq \tau) \)

I. \( \Gamma \vdash \text{inr} : e : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e : \tau_2 \to \tau_1 + \tau_2 \wedge \tau_1 + \tau_2 \leq \tau) \)

J. \( \Gamma \vdash \text{case} e_1 \text{ of } \text{inl} x \Rightarrow e_2 \text{ else } 0 \Rightarrow e_3 : \tau \Rightarrow \)

\( \exists \tau_1, \tau_2, \tau' : (\Gamma \vdash e_1 : \tau_1 + \tau_2 \wedge \Gamma \{ x : \tau_1 \} \vdash e_2 : \tau' \wedge \Gamma \{ y : \tau_2 \} \vdash e_3 : \tau' \wedge \tau' \leq \tau) \)

K. \( \Gamma \vdash e \text{. } \text{fast} : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e : \tau \times \tau_2 \wedge \tau_1 \leq \tau) \)

L. \( \Gamma \vdash e \text{. } \text{snd} : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e : \tau \times \tau_2 \wedge \tau_2 \leq \tau) \)

M. \( \Gamma \vdash \text{roll} : e : \tau \Rightarrow \exists \tau, \tau' : (\Gamma \vdash e : \mu t. \tau / l[\tau] \wedge \mu t. \tau \leq \tau) \)

N. \( \Gamma \vdash \text{unroll} : e : \tau \Rightarrow \exists \tau, \tau' : (\Gamma \vdash e : \mu t. \tau \wedge \mu t. \tau / l[\tau] \leq \tau) \)

O. \( \Gamma \vdash x : \tau \Rightarrow (\Gamma (x) \leq \tau) \)

**PROOF.** By induction on the derivation of \( \Gamma \vdash e : \tau \). In all the lemma’s cases, exactly two rules could apply: T-SUBSUME (in which case the result follows from an inductive
argument) and another rule (in which case the result is immediate). For example,  
\( \Gamma \vdash e : T \) is derivable with T-SUBSUME and T-FST. With T-SUBSUME, the inductive hypothesis implies \( \Gamma \vdash e : \tau \times \tau_2 \) and \( \tau \leq \tau' \), for a type \( \tau' \) such that \( \tau' \leq \tau \). By Corollary 18 then, \( \tau \leq \tau' \), as required. If \( \Gamma \vdash e : \tau \times \tau_2 \) and \( \tau = \tau_1 \). By Corollary 18 then, \( \tau \leq \tau' \), as required. All the other cases are proved similarly.

**Lemma 25.** \( \beta \)-Preservation.

\[ \forall \tau, \tau' : ((e : \tau \land e \rightarrow_{\beta} e') \Rightarrow e' : \tau) \]

**Proof.** By case analysis of \( e \rightarrow_{\beta} e' \). We show the proofs of the \( \beta \)-SUC, \( \beta \)-APP, and \( \beta \)-UNROLL cases. The proofs of the \( \beta \)-NEG cases are similar to that of \( \beta \)-SUC; the proofs of the \( \beta \)-LEFT and \( \beta \)-RIGHT cases are similar to that of \( \beta \)-APP; and the proofs of the \( \beta \)-FST and \( \beta \)-SND cases are similar to that of \( \beta \)-UNROLL.

Case \[ \text{succ} : \beta \text{-SUC} \]

Because \( \text{succ}(n) : \tau \), Lemma 24 ensures that \( \text{nat} \leq \tau \), while rule T-NAT ensures that \( n' : \text{nat} \). Hence, \( n' : \tau \) by rule T-SUBSUME.

Case \[ \text{unroll} : \beta \text{-UNROLL} \]

By Lemma 24 and the assumption that \( \text{unroll}(v) : \tau \), we have \( \text{roll}(v) : \mu \text{t} . \tau \) and \( \mu \text{t} . \tau / \tau \leq \tau \). Then by Lemma 24 again and the result that \( \text{roll}(v) : \mu \text{t} . \tau \), we find \( v : \mu \text{t} . \tau / \tau \). Because \( \mu \text{t} . \tau / \tau \leq \mu \text{t} . \tau \), Lemma 12 implies that \( \mu \text{t} . \tau / \tau \leq \mu \text{t} . \tau / \tau \). Hence, we have \( v : \mu \text{t} . \tau / \tau \). By rule T-SUBSUME.

**Lemma 26.** Well-Typed, Filled Contexts.

\[ \forall \Gamma, E, e, \tau : (\Gamma \vdash E[e] : \tau) \Rightarrow \exists \rho : \Gamma' : (\Gamma \vdash e : \rho \land \Gamma \vdash E[\rho] : \rho) \]

**Proof.** By induction on the structure of \( E \). If \( E = [ \] \), then the result is immediate with \( \rho = \tau \), because \( \Gamma \vdash e : \tau \) by assumption and \( \Gamma \vdash [ \tau : \tau \). By the definition of well-typed contexts and rule T-VAR. If \( E = \text{succ}(E') \) then we can apply Lemma 24 to the assumption that \( \Gamma \vdash \text{succ}(E'[\rho]) : \tau \) to find that \( \Gamma \vdash E'[\rho] : \text{nat} \) and \( \text{nat} \leq \tau \). By the inductive hypothesis then, there exists a \( \rho' \) such that \( \Gamma \vdash e : \rho' \) and \( \Gamma \vdash E'[\rho'] : \text{nat} \), so by the definition of well-typed contexts, \( \Gamma \vdash \{ x : \tau' \} \vdash E'[\rho'] : \text{nat} \). Then by rule T-SUC, \( \Gamma \vdash \{ x : \tau' \} \vdash \text{succ}(E'[\rho']) : \text{nat} \), implying by T-SUBSUME and \( \text{nat} \leq \tau \) that \( \Gamma \vdash \{ x : \tau' \} \vdash \text{succ}(E'[\rho']) : \tau \). Hence, by rule T-CUT we have \( \Gamma \vdash E[\rho'] : \tau \), which completes this proof case. The proofs of the other cases are all similar.
LEMMA 27. Preservation.

\[ \forall e, \tau, e' : ((e : \tau \wedge e \mapsto e') \Rightarrow e' : \tau) \]

PROOF. Only one rule derived \( e \mapsto e' \), so it must be the case that \( e = E[e_1], e' = E[e_2] \), and \( e_1 \mapsto e_2 \) (for some \( E, e_1 \), and \( e_2 \)). Because \( e : \tau \), we have \( E[e_1] : \tau \), so by Lemma 26 there exists a \( \tau' \) such that \( e_1 : \tau' \) and \( E[\tau'] \). Combining \( e_1 : \tau' \) with \( e_1 \mapsto e_2 \), Lemma 25 ensures that \( e_2 : \tau' \). Finally, because \( E[\tau'] \), we have \( \{x : \tau'\} \vdash E[x] : \tau \), which combines with \( e_2 : \tau' \) and Lemma 20 to imply that \( E[e_2] : \tau \). Hence, \( e' : \tau \) as required. □


\[ \forall e, \tau : (e : \tau \Rightarrow (\exists v : (e = v) \lor \exists e' : (e \mapsto e'))) \]

PROOF. By induction on the derivation of \( e : \tau \). The proof is a standard progress proof using the canonical-forms Lemma 21 (and Lemma 14 in the T-APP case, to ensure that Case C of Lemma 21 applies). □

LEMMA 29. Progress.

\[ \forall e, \tau : (e : \tau \Rightarrow (\exists v : (e = v) \lor \exists e' : (e \mapsto e'))) \]

PROOF. By assumption, \( e : \tau \), soLemma 28 implies that either \( e = v \) or \( e = E[e_1] \) such that \( e_1 \mapsto e_2 \). In the case of \( e = E[e_1] \) such that \( e_1 \mapsto e_2 \), the dynamic semantics ensures that \( e \mapsto E[e_2] \). □

With Preservation and Progress, we have type safety.

LEMMA 30. Type Safety.

\[ \forall e, \tau, e' : ((e : \tau \wedge e \mapsto^* e') \Rightarrow \neg \text{stuck}(e')) \]

PROOF. By induction on the derivation of \( e \mapsto^* e' \), using Progress and Preservation (Lemmas 29 and 27) in the usual way. □

As with \( \lambda \), the soundness of the subtyping relation follows from the variable-substitution and type-safety results (Lemmas 20 and 30).

LEMMA 31. Soundness.

\[ \forall \tau_1, \tau_2 : (\tau_1 \leq \tau_2 \Rightarrow \neg \exists E, \tau, e, e' : (E[\tau_2] : \tau \wedge e : \tau_1 \wedge E[e] \mapsto^* e' \wedge \text{stuck}(e'))) \]

PROOF. The proof is the same as for soundness in \( \lambda \) (Lemma 6). □

A.7. Subtyping Preciseness

Finally, the completeness and soundness results combine to ensure that the subtyping relation defined in Figure 9 is precise with respect to type safety.

THEOREM 32. Preciseness.
The \( \leq \) relation is precise with respect to type safety. Formally, for all types \( \tau_1 \) and \( \tau_2 \):

\[ \tau_1 \leq \tau_2 \iff \left( \neg \exists E, \tau, e, e' : E[\tau_2] : \tau \wedge e : \tau_1 \wedge E[e] \mapsto^* e' \wedge \text{stuck}(e') \right) \]

PROOF. Immediate by Corollary 23 and Lemma 31. □