Completely Subtyping Iso-recursive Types

Technical Report CSE-071012

Jeremy Blackburn Ivory Hernandez Jay Ligatti Michael Nachtigal

Department of Computer Science and Engineering
University of South Florida
{jhblackb,ihernan4,ligatti,mnachtig}@cse.usf.edu

Abstract

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 are 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 τ_1 and τ_2 such that τ_1 can safely be considered a subtype of τ_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.

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

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., τ_1 is a subtype of τ_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 τ_1 to stand in for terms of another type τ_2, when operations could be performed on τ_2-type terms that are not defined for τ_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 $L \equiv \mu t.(\texttt{unit}+(\texttt{nat} \times t))$. 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 τ for t in τ'). For example, a programmer could create a value of type L defined above by writing $\texttt{roll}(\texttt{inl}_{\texttt{unit}+(\texttt{nat}\times L)}())$; the `inl` value has type $\texttt{unit}+(\texttt{nat}\times L)$, so rolling it produces a value of type L. Languages like ML and Haskell support iso-recursive types. In contrast, 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.

1.1 Related Work

Research into subtyping completeness has focused on proving subtyping algorithms complete with respect to definitions of subtyping relations (e.g., [6, 11, 15, 23]). Sekiguchi and Yonezawa also proved a type-inference algorithm sound and complete in the presence of subtyped recursive types [19].

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 OOPLs [14, 17]. Another possibility is to consider the denotation of a type τ to be the set of terms of type τ; then a subtyping relation $\leq$ is sound when $\tau_1 \leq \tau_2 \Rightarrow [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$ and complete when $[\![\tau_1]\!] \subseteq [\![\tau_2]\!] \Rightarrow \tau_1 \leq \tau_2$ [25]. Using these definitions, Vouillon has shown that the standard subtyping variance rules for function, union, and intersection types are sound and complete (under some assumptions but overall for a broad class of languages) [25]. 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 [18]) 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 [1]. The rules and algorithm are proved sound and complete with respect to type trees that result from "in-

[Copyright notice will appear here once 'preprint' option is removed.]

finitely unrolling" equi-recursive types (i.e., the rules and algorithm determine $\tau_1 \leq \tau_2$ precisely when the type obtained by infinitely unrolling τ_1 is a subtype of the type obtained by infinitely unrolling τ_2). Other papers have since refined equi-recursive subtyping analyses and algorithms (e.g., [4, 6, 8, 9, 12, 22]).

For subtyping iso-recursive types, the most commonly used rules are the Amber rules [5]:

$$\frac{S \cup \{t \leq t'\} \vdash \tau \leq \tau'}{S \vdash \mu t.\tau \leq \mu t'.\tau'} \text{ AMBER1} \qquad \frac{}{S \cup \{t \leq t'\} \vdash t \leq t'} \text{ AMBER2}$$

These rules are elegant: "a recursive type rec$(t)T$ is included in a recursive type rec$(u)U$, if assuming t included in u implies T included in U" [5].

The Amber rules (or less-complete versions of the Amber rules tailored to specific domains, e.g., [2]) are the standard approach to defining iso-recursive subtyping (e.g., [3, 7, 10, 11, 16, 20, 21]).

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 derives that τ_1 is a subtype of τ_2 if and only if terms of type τ_1 can always stand in for terms of type τ_2 without compromising type safety. Section 2 uses evaluation contexts to formalize this intuition.

Section 3 shows that the Amber rules are incomplete for subtyping iso-recursive types. In particular, the Amber rules cannot derive that types like $\mu a.(((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a)$ are subtypes of types like $\mu c.((c + \texttt{real}) + c)$, though it's always safe for expressions of type $\mu a.(((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a)$ to stand in for expressions of type $\mu c.((c + \texttt{real}) + c)$.

Given the incompleteness of the Amber rules, Section 4 presents new subtyping rules that do not exhibit such incompleteness. The main finding here is that, for the sake of completeness, the Amber rules can be replaced by the following rules:

$$\frac{(\mu t.\tau \leq \mu t'.\tau') \notin S \qquad S \cup \{\mu t.\tau \leq \mu t'.\tau'\} \vdash [\mu t.\tau/t]\tau \leq [\mu t'.\tau'/t']\tau'}{S \vdash \mu t.\tau \leq \mu t'.\tau'} \text{ REC1}$$

$$\frac{}{S \cup \{\mu t.\tau \leq \mu t'.\tau'\} \vdash \mu t.\tau \leq \mu t'.\tau'} \text{ REC2}$$

These new rules simultaneously unroll the iso-recursive types under consideration, matching the types obtained when recursive-type values are eliminated (using unroll expressions). Moreover, these new rules imply a deterministic algorithm for deciding whether one type is a subtype of another.

Section 5 proves that the subtyping relation defined in Section 4 is 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 preciseness proof's layout, and its proof techniques, are rather general and may be helpful for proving the preciseness of other inductively defined subtyping relations.

2. Basic Definitions

This paper's analysis relies on several definitions.

2.1 Soundness, Completeness, Preciseness

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 τ_1 also has type τ_2 when $\tau_1 \leq \tau_2$, then, we wish to define $\tau_1 \leq \tau_2$ precisely when any term of type τ_2 could be replaced by any term of type τ_1 without breaking type safety.

The following definition formalizes this requirement that $\tau_1 \leq \tau_2$ if and only if τ_2-type expressions can—in any context—be replaced by τ_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 τ'-type expression produces a τ-type expression (formally, $E[\tau']{:}\tau \iff \{x{:}\tau'\} \vdash E[x]{:}\tau$, where x is not free in E).

Definition 1 (Preciseness, Soundness, and Completeness)**.** *Let metavariables E, e, and τ 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 τ_1 and τ_2:*

$$\tau_1 \leq \tau_2 \iff \left(\begin{array}{l} \neg \exists\, E, \tau, e, e' : \\ E[\tau_2]{:}\tau \wedge e{:}\tau_1 \wedge E[e] \mapsto^* e' \wedge \texttt{stuck}(e') \end{array} \right)$$

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.*

2.2 A Simple Language, $L^{\rightarrow\mu}_{+\times}$

To more concretely understand and apply these definitions, we consider a simple language $L^{\rightarrow\mu}_{+\times}$, having function types, binary (disjoint) sum and product types, and iso-recursive types. Figures 1–3 present the syntax and static and dynamic semantics. All the notation is intended to have the usual meanings, with the usual assumptions being made (e.g., variables are consistently renamed, through alpha-conversion, whenever necessary to avoid reintroducing variables into contexts).

The base types in $L^{\rightarrow\mu}_{+\times}$ are `nat` (natural numbers) and `real` (nonnegative real numbers), the idea being that $\texttt{nat} \leq \texttt{real}$. Square-root operations are defined on natural and real numbers, and a successor operation is defined on natural, but not real, numbers. Functions are named and may be recursive. The decision to allow recursive functions was made because (1) real languages sophisticated enough to have iso-recursive types and subtyping seem likely to also have recursive functions, and (2) allowing general recursion prevents our proofs from relying on normalization properties of the language being analyzed.

There are two categories of types, one for closed types τ (having no free type variables) and the other for possibly open types $\overline{\tau}$ (possibly having free type variables). This paper assumes that all closed types τ (1) never use undeclared variables and (2) have been alpha-converted to ensure the uniqueness of every declared variable. Note that unrolling a closed recursive type $\tau = \mu t.\overline{\tau}$ produces another closed type, $[\mu t.\overline{\tau}/t]\overline{\tau}$.

The typing rules for $L^{\rightarrow\mu}_{+\times}$, shown in Figure 2, are standard. Section 4 will define the subtyping judgment used by rule T-SUBSUME.

The operational rules for $L^{\rightarrow\mu}_{+\times}$, shown in Figure 3, are also standard. Figure 3 uses evaluation contexts to define the operational semantics. Evaluation contexts mark where beta-reductions may occur; contexts here specify a left-to-right, call-by-value evaluation strategy.

3. Incompleteness of the Amber Rules

Consider the recursive types τ and τ' defined as follows.

$$\tau = \mu i.\{\texttt{add}{:}i \rightarrow \texttt{unit}\}$$

$$\tau' = \mu l.\{\texttt{add}{:}(\mu i'.\{\texttt{add}{:}i' \rightarrow \texttt{unit}\}) \rightarrow \texttt{unit},\ \texttt{min}{:}\texttt{unit} \rightarrow \texttt{int}\}$$

Closed types

$\tau ::= \texttt{nat} \mid \texttt{real} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu t.\overline{\tau}$

Possibly open types

$\overline{\tau} ::= \texttt{nat} \mid \texttt{real} \mid \overline{\tau_1} \to \overline{\tau_2} \mid \overline{\tau_1} + \overline{\tau_2} \mid \overline{\tau_1} \times \overline{\tau_2} \mid \mu t.\overline{\tau} \mid t$

Expressions

$e ::= \texttt{n} \mid \texttt{r} \mid \texttt{succ}(e) \mid \texttt{sqrt}(e) \mid \texttt{fun}\ f(x{:}\tau_1){:}\tau_2\ =\ e \mid e_1(e_2) \mid x \mid \texttt{inl}_\tau(e) \mid \texttt{inr}_\tau(e) \mid$
$\texttt{case}\ e\ \texttt{of inl}x \Rightarrow e_1\ \texttt{else inr}y \Rightarrow e_2(e_1, e_2) \mid e.\texttt{fst} \mid e.\texttt{snd} \mid \texttt{roll}(e) \mid \texttt{unroll}(e)$

Figure 1. Syntax of $L_{+\times}^{\to\mu}$.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash \texttt{n} : \texttt{nat}}\ \text{T-NAT} \qquad \frac{}{\Gamma \vdash \texttt{r} : \texttt{real}}\ \text{T-REAL} \qquad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{succ}(e) : \texttt{nat}}\ \text{T-SUCC} \qquad \frac{\Gamma \vdash e : \texttt{real}}{\Gamma \vdash \texttt{sqrt}(e) : \texttt{real}}\ \text{T-SQRT}$$

$$\frac{\Gamma \cup \{x{:}\tau_1, f{:}\tau_1 \to \tau_2\} \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ f(x : \tau_1) : \tau_2 = e : \tau_1 \to \tau_2}\ \text{T-FUN} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ \text{T-APP} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inl}_{\tau_1+\tau_2} e : \tau_1 + \tau_2}\ \text{T-LEFT}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inr}_{\tau_1+\tau_2} e : \tau_1 + \tau_2}\ \text{T-RIGHT} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma \cup \{x{:}\tau_1\} \vdash e_1 : \tau \qquad \Gamma \cup \{y{:}\tau_2\} \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of inl}x \Rightarrow e_1\ \texttt{else inr}y \Rightarrow\ e_2 : \tau}\ \text{T-CASE}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \text{T-PROD} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.\texttt{fst} : \tau_1}\ \text{T-FST} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.\texttt{snd} : \tau_2}\ \text{T-SND} \qquad \frac{\Gamma \vdash e : \mu t.\overline{\tau}}{\Gamma \vdash \texttt{unroll}(e) : [\mu t.\overline{\tau}/t]\overline{\tau}}\ \text{T-UNROLL}$$

$$\frac{\Gamma \vdash e : [\mu t.\overline{\tau}/t]\overline{\tau}}{\Gamma \vdash \texttt{roll}(e) : \mu t.\overline{\tau}}\ \text{T-ROLL} \qquad \frac{}{\Gamma \cup \{x{:}\tau\} \vdash x : \tau}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e : \tau' \qquad \tau' \le \tau}{\Gamma \vdash e : \tau}\ \text{T-SUBSUME}$$

$\boxed{\Gamma \vdash E[\tau'] : \tau}$

$$\frac{\Gamma \cup \{x{:}\tau'\} \vdash E[x] : \tau \qquad (x \text{ not free in } E)}{\Gamma \vdash E[\tau'] : \tau}\ \text{T-CTXT}$$

Figure 2. Static semantics of $L_{+\times}^{\to\mu}$.

Evaluation contexts $E ::= [\,] \mid \texttt{succ}(E) \mid \texttt{sqrt}(E) \mid E\ (e) \mid v\ (E) \mid \texttt{inl}_\tau E \mid \texttt{inr}_\tau E \mid \texttt{case}\ E\ \texttt{of inl}x \Rightarrow e_1\ \texttt{else inr}y \Rightarrow e_2 \mid$
$(E, e) \mid (v, E) \mid E.\texttt{fst} \mid E.\texttt{snd} \mid \texttt{roll}(E) \mid \texttt{unroll}(E)$

Values $v ::= \texttt{n} \mid \texttt{r} \mid \texttt{fun}\ f(x : \tau_1) : \tau_2\ =\ e \mid \texttt{inl}_\tau(v) \mid \texttt{inr}_\tau(v) \mid (v_1, v_2) \mid \texttt{roll}(v)$

$\boxed{e \mapsto e'}$

$$\frac{e \mapsto_\beta e'}{E[e] \mapsto E[e']}\ \text{O-STEP}$$

$\boxed{\texttt{stuck}(e)}$

$$\frac{\neg\exists v : (e = v) \qquad \neg\exists e' : (e \mapsto e')}{\texttt{stuck}(e)}\ \text{STUCK}$$

$\boxed{e \mapsto^* e'}$

$$\frac{}{e \mapsto^* e}\ \text{MSTEP-REFL} \qquad \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}\ \text{MSTEP-TRANS}$$

$\boxed{e \mapsto_\beta e'}$

$$\frac{\texttt{n}' = \texttt{successor of n}}{\texttt{succ(n)} \mapsto_\beta \texttt{n}'}\ \beta\text{-SUCC} \qquad \frac{\texttt{r}' = \texttt{square root of r}}{\texttt{sqrt(r)} \mapsto_\beta \texttt{r}'}\ \beta\text{-SQRTR} \qquad \frac{\texttt{r} = \texttt{square root of n}}{\texttt{sqrt(n)} \mapsto_\beta \texttt{r}}\ \beta\text{-SQRTN}$$

$$\frac{}{(\texttt{fun}\ f(x : \tau_1) : \tau_2 = e)(v) \mapsto_\beta [(\texttt{fun}\ f(x : \tau_1) : \tau_2 = e)/f][v/x]e}\ \beta\text{-APP}$$

$$\frac{}{\texttt{case inl}_\tau(v)\ \texttt{of inl}x \Rightarrow e_2\ \texttt{else inr}y \Rightarrow e_3 \mapsto_\beta [v/x]e_2}\ \beta\text{-LEFT}$$

$$\frac{}{\texttt{case inr}_\tau(v)\ \texttt{of inl}x \Rightarrow e_2\ \texttt{else inr}y \Rightarrow e_3 \mapsto_\beta [v/y]e_3}\ \beta\text{-RIGHT}$$

$$\frac{}{(v_1, v_2).\texttt{fst} \mapsto_\beta v_1}\ \beta\text{-FST} \qquad \frac{}{(v_1, v_2).\texttt{snd} \mapsto_\beta v_2}\ \beta\text{-SND} \qquad \frac{}{\texttt{unroll(roll}(v)) \mapsto_\beta v}\ \beta\text{-UNROLL}$$

Figure 3. Dynamic semantics of $L_{+\times}^{\to\mu}$.

These types arise naturally when encoding the following OOPL classes into a language like $L^{\rightarrow\mu}_{+\times}$ (extended to have record, `unit`, and `int` types).

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

However, the Amber rules (in conjunction with standard subtyping rules for records and functions) can't derive $\tau'{\leq}\tau$, as Figure 4 illustrates.

For another example, let's redefine τ and τ' as follows.

$$\tau = \mu c.((c + \texttt{real}) + c)$$
$$\tau' = \mu a.(((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a)$$

This may be a more interesting example because all the declared type variables get used (unlike the type variable l in the previous example's τ'). Again, the Amber rules (in conjunction with the standard subtyping rule for binary sums) cannot be used to derive $\tau'{\leq}\tau$, as shown in Figure 5.

We prove that it's safe to consider $\tau'{\leq}\tau$ in two steps: first, Section 4 shows that $\tau'{\leq}\tau$ is derivable using new subtyping rules; then Section 5 shows that those new subtyping rules are sound with respect to type safety.

4. Defining a More Complete Subtyping Relation

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

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

$$\frac{(\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}') \notin S \qquad S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'}{S \vdash\ \mu t.\overline{\tau}{\leq}\mu t'.\overline{\tau}'}\ \textsc{Rec1}$$

$$\frac{}{S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash \mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'}\ \textsc{Rec2}$$

The context S contains subtyping assumptions of the form $\mu t.\overline{\tau}{\leq}\mu t'.\overline{\tau}'$. As with other judgment forms that use contexts (such as typing judgments), when the context is empty, we abbreviate judgments of the form $\emptyset{\vdash}\tau_1{\leq}\tau_2$ as $\tau_1{\leq}\tau_2$. Also, rules REC1 and REC2, like the Amber rules, require type-variable names to be unique; as mentioned in Section 2.2, this paper assumes that every time a type variable is introduced, it is given a distinct name. Rules REC1 and REC2 only unroll recursive types and therefore never introduce type variables. We also note that other systems have used rules similar to REC1 and REC2 to define equivalence, rather than subtyping, relations on iso-recursive types [13, 24].

With rules REC1 and REC2 we can derive $\tau'{\leq}\tau$ for both examples in Section 3, as shown in Figures 6–7. Recall that the Amber rules could not derive $\tau'{\leq}\tau$ for these examples.

It's tempting to try to define a subtyping relation for $L^{\rightarrow\mu}_{+\times}$ by including rules REC1 and REC2 verbatim; however, to make the subtyping relation complete, we must consider a technicality that will affect the REC1 rule. The technicality is that, because $L^{\rightarrow\mu}_{+\times}$ has (1) a call-by-value semantics, (2) recursive functions, and (3) iso-recursive types, we find that all types in $L^{\rightarrow\mu}_{+\times}$ are inhabited (e.g., the expression `(fun f(x:nat):`τ`=f(x))(0)` has type τ for any τ), but some types are inhabited only by nonterminating expressions. 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.

Because $L^{\rightarrow\mu}_{+\times}$ has a call-by-value semantics, we can treat any type inhabited only by diverging expressions as being equivalent to a $\bot$ type. If all expressions of a type τ diverge, then any τ-type expression can substitute for any expression of any type; such a substitution won't compromise type safety because the τ-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 begin defining the subtyping relation for $L^{\rightarrow\mu}_{+\times}$ with the following rules:

$$\frac{\mathrm{val}(\tau_1) = \emptyset}{S \vdash\ \tau_1{\leq}\tau_2}\ \text{S-}\bot \qquad \frac{\mathrm{val}(\tau_1) \neq \emptyset \qquad \mathrm{val}(\tau_2) = \emptyset}{S \vdash \tau_1 \leq \tau_2{\rightarrow}\tau_2'}\ \text{S-}\bot\textsc{Fun}$$

The judgment $\mathrm{val}(\tau) = \emptyset$ indicates that type τ is uninhabited by values (i.e., $\tau \equiv \bot$), while judgment $\mathrm{val}(\tau) \neq \emptyset$ indicates the opposite. These rules are similar to ones described by Vouillon [25].

Given rules S-$\bot$ and S-$\bot$FUN, we can continue making the subtyping rules deterministic by including a premise of the form $\mathrm{val}(\tau_1) \neq \emptyset$ in every rule for concluding $S{\vdash}\tau_1{\leq}\tau_2$, where τ_1 could otherwise be value-uninhabited. For example, we add this premise to rule REC1 to obtain the following.

$$\frac{\mathrm{val}(\mu t.\overline{\tau}) \neq \emptyset \qquad (\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}') \notin S \qquad S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'}{S \vdash\ \mu t.\overline{\tau}{\leq}\mu t'.\overline{\tau}'}\ \text{S-}\textsc{Rec1}$$

The derivations in Figures 6–7 only consider value-inhabited types, so they could use rule S-REC1 in place of REC1 without any major changes.

Continuing to define the other subtyping rules in this way, we arrive at the full definition of the subtyping relation for $L^{\rightarrow\mu}_{+\times}$, as shown in Figure 8.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Downarrow\ \texttt{Derivation fails here}\ \Downarrow}{\{l{\leq}i\} \vdash i \leq \mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}} \qquad \dfrac{}{\{l{\leq}i\} \vdash \texttt{unit}{\leq}\texttt{unit}}\ \textsc{Reflexive}}{\{l{\leq}i\} \vdash (\mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}){\to}\texttt{unit} \leq i{\to}\texttt{unit}}\ \textsc{Function-Sub}}{\{l{\leq}i\} \vdash \{\texttt{add}{:}(\mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}){\to}\texttt{unit},\ \texttt{min}{:}\texttt{unit}{\to}\texttt{int}\} \leq \{\texttt{add}{:}i{\to}\texttt{unit}\}}\ \textsc{Record-Sub}}{\mu l.\{\texttt{add}{:}(\mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}){\to}\texttt{unit},\ \texttt{min}{:}\texttt{unit}{\to}\texttt{int}\} \leq \mu i.\{\texttt{add}{:}i{\to}\texttt{unit}\}}\ \textsc{Amber1}$$

Figure 4. Failing derivation of $\mu l.\{\texttt{add}{:}(\mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}){\to}\texttt{unit},\ \texttt{min}{:}\texttt{unit}{\to}\texttt{int}\} \leq \mu i.\{\texttt{add}{:}i{\to}\texttt{unit}\}$, using the Amber rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Downarrow\ \texttt{Derivation fails here}\ \Downarrow}{\{a{\leq}c\} \vdash \mu b.((b + \texttt{nat}) + a) \leq c} \qquad \dfrac{}{\{a{\leq}c\} \vdash \texttt{nat}{\leq}\texttt{real}}\ \textsc{Base}}{\{a{\leq}c\} \vdash (\mu b.((b + \texttt{nat}) + a)) + \texttt{nat} \leq c + \texttt{real}}\ \textsc{Sum-Sub} \qquad \dfrac{}{\{a{\leq}c\} \vdash a{\leq}c}\ \textsc{Amber2}}{\{a{\leq}c\} \vdash ((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a \leq (c + \texttt{real}) + c}\ \textsc{Sum-Sub}}{\mu a.(((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a) \leq \mu c.((c + \texttt{real}) + c)}\ \textsc{Amber1}$$

Figure 5. Failing derivation of $\mu a.(((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a) \leq \mu c.((c + \texttt{real}) + c)$, using the Amber rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F \vdash I{\leq}I'}\ \textsc{Rec2} \qquad \dfrac{}{F \vdash \texttt{unit}{\leq}\texttt{unit}}\ \textsc{Reflexive}}{F \vdash I'{\to}\texttt{unit}{\leq}I{\to}\texttt{unit}}\ \textsc{Function-Sub}}{F \vdash \{\texttt{add}{:}I'{\to}\texttt{unit}\}{\leq}\{\texttt{add}{:}I{\to}\texttt{unit}\}}\ \textsc{Record-Sub}}{\{L{\leq}I, I{\leq}I'\} \vdash I'{\leq}I}\ \textsc{Rec1} \qquad \dfrac{}{\{L{\leq}I, I{\leq}I'\} \vdash \texttt{unit}{\leq}\texttt{unit}}\ \textsc{Reflexive}}{\{L{\leq}I, I{\leq}I'\} \vdash I{\to}\texttt{unit}{\leq}I'{\to}\texttt{unit}}\ \textsc{Function-Sub}}{\{L{\leq}I, I{\leq}I'\} \vdash \{\texttt{add}{:}I{\to}\texttt{unit}\}{\leq}\{\texttt{add}{:}I'{\to}\texttt{unit}\}}\ \textsc{Record-Sub}}{\{L{\leq}I\} \vdash I{\leq}I'}\ \textsc{Rec1} \qquad \dfrac{}{\{L{\leq}I\} \vdash \texttt{unit}{\leq}\texttt{unit}}\ \textsc{Reflexive}}{\{L{\leq}I\} \vdash I'{\to}\texttt{unit}{\leq}I{\to}\texttt{unit}}\ \textsc{Function-Sub}}{\{L{\leq}I\} \vdash \{\texttt{add}{:}I'{\to}\texttt{unit}, \texttt{min}{:}\texttt{unit}{\to}\texttt{int}\}{\leq}\{\texttt{add}{:}I{\to}\texttt{unit}\}}\ \textsc{Record-Sub}}{L{\leq}I}\ \textsc{Rec1}$$

Figure 6. Derivation of $L{\leq}I$ using the new subtyping rules, where $L{=}\mu l.\{\texttt{add}{:}(\mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}){\to}\texttt{unit}, \texttt{min}{:}\texttt{unit}{\to}\texttt{int}\}$, $I{=}\mu i.\{\texttt{add}{:}i{\to}\texttt{unit}\}$, $I'{=}\mu i'.\{\texttt{add}{:}i'{\to}\texttt{unit}\}$, and $F = \{L{\leq}I, I{\leq}I', I'{\leq}I\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{S{\vdash}B{\leq}C}\ \textsc{Rec2} \qquad \dfrac{}{S{\vdash}\texttt{nat}{\leq}\texttt{real}}\ \textsc{Base}}{S{\vdash}B + \texttt{nat}{\leq}C + \texttt{real}}\ \textsc{Sum-Sub} \qquad \dfrac{}{S{\vdash}A{\leq}C}\ \textsc{Rec2}}{S{\vdash}(B + \texttt{nat}) + A \leq (C + \texttt{real}) + C}\ \textsc{Sum-Sub}}{\{A{\leq}C\}{\vdash}B{\leq}C}\ \textsc{Rec1} \qquad \dfrac{}{\{A{\leq}C\}{\vdash}\texttt{nat}{\leq}\texttt{real}}\ \textsc{Base}}{\{A{\leq}C\}{\vdash}B + \texttt{nat} \leq C + \texttt{real}}\ \textsc{Sum-Sub} \qquad \dfrac{}{\{A{\leq}C\}{\vdash}A{\leq}C}\ \textsc{Rec2}}{\{A{\leq}C\}{\vdash}(B + \texttt{nat}) + A \leq (C + \texttt{real}) + C}\ \textsc{Sum-Sub}}{A{\leq}C}\ \textsc{Rec1}$$

Figure 7. Derivation of $A{\leq}C$ using the new subtyping rules, where $A{=}\mu a.(((\mu b.((b + \texttt{nat}) + a)) + \texttt{nat}) + a)$, $B{=}\mu b.((b + \texttt{nat}) + A)$, $C{=}\mu c.((c + \texttt{real}) + c)$, and $S = \{A{\leq}C, B{\leq}C\}$.

$\boxed{S \vdash\ \tau_1 \leq \tau_2}$

$$\frac{}{S \vdash\ \texttt{nat} \leq \texttt{real}}\ \text{S-BASE} \qquad \frac{}{S \vdash\ \texttt{nat} \leq \texttt{nat}}\ \text{S-NAT} \qquad \frac{}{S \vdash\ \texttt{real} \leq \texttt{real}}\ \text{S-REAL} \qquad \frac{\text{val}(\tau_1) = \emptyset}{S \vdash\ \tau_1 \leq \tau_2}\ \text{S-}\bot$$

$$\frac{\text{val}(\tau_1) \neq \emptyset \qquad \text{val}(\tau_2) = \emptyset}{S \vdash\ \tau_1 \leq \tau_2 \to \tau_2'}\ \text{S-}\bot\text{FUN} \qquad \frac{\text{val}(\tau_1') \neq \emptyset \qquad S \vdash\ \tau_1' \leq \tau_1 \qquad S \vdash\ \tau_2 \leq \tau_2'}{S \vdash\ \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}\ \text{S-FUN}$$

$$\frac{\text{val}(\tau_1 + \tau_2) \neq \emptyset \qquad S \vdash\ \tau_1 \leq \tau_1' \qquad S \vdash\ \tau_2 \leq \tau_2'}{S \vdash\ \tau_1 + \tau_2 \leq \tau_1' + \tau_2'}\ \text{S-SUM} \qquad \frac{\text{val}(\tau_1 \times \tau_2) \neq \emptyset \qquad S \vdash\ \tau_1 \leq \tau_1' \qquad S \vdash\ \tau_2 \leq \tau_2'}{S \vdash\ \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}\ \text{S-PROD}$$

$$\frac{\text{val}(\mu t.\overline{\tau}) \neq \emptyset \qquad (\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}') \notin S \qquad S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'}{S \vdash\ \mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'}\ \text{S-REC1}$$

$$\frac{}{S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash \mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'}\ \text{S-REC2}$$

$\boxed{U \vdash \text{val}(\tau) = \emptyset}$

$$\frac{U \vdash \text{val}(\tau_1) = \emptyset \qquad U \vdash \text{val}(\tau_2) = \emptyset}{U \vdash \text{val}(\tau_1 + \tau_2) = \emptyset}\ \text{U-SUM} \qquad \frac{U \vdash \text{val}(\tau_1) = \emptyset}{U \vdash \text{val}(\tau_1 \times \tau_2) = \emptyset}\ \text{U-PROD1} \qquad \frac{U \vdash \text{val}(\tau_1) \neq \emptyset \quad U \vdash \text{val}(\tau_2) = \emptyset}{U \vdash \text{val}(\tau_1 \times \tau_2) = \emptyset}\ \text{U-PROD2}$$

$$\frac{\mu t.\overline{\tau} \notin U \qquad U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) = \emptyset}{U \vdash \text{val}(\mu t.\overline{\tau}) = \emptyset}\ \text{U-REC1} \qquad \frac{}{U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}(\mu t.\overline{\tau}) = \emptyset}\ \text{U-REC2}$$

$\boxed{U \vdash \text{val}(\tau) \neq \emptyset}$

$$\frac{}{U \vdash \text{val}(\texttt{nat}) \neq \emptyset}\ \text{I-NAT} \qquad \frac{}{U \vdash \text{val}(\texttt{real}) \neq \emptyset}\ \text{I-REAL} \qquad \frac{}{U \vdash \text{val}(\tau_1 \to \tau_2) \neq \emptyset}\ \text{I-FUN} \qquad \frac{U \vdash \text{val}(\tau_1) \neq \emptyset}{U \vdash \text{val}(\tau_1 + \tau_2) \neq \emptyset}\ \text{I-SUM1}$$

$$\frac{U \vdash \text{val}(\tau_1) = \emptyset \quad U \vdash \text{val}(\tau_2) \neq \emptyset}{U \vdash \text{val}(\tau_1 + \tau_2) \neq \emptyset}\ \text{I-SUM2} \qquad \frac{U \vdash \text{val}(\tau_1) \neq \emptyset \quad U \vdash \text{val}(\tau_2) \neq \emptyset}{U \vdash \text{val}(\tau_1 \times \tau_2) \neq \emptyset}\ \text{I-PROD} \qquad \frac{\mu t.\overline{\tau} \notin U \quad U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) \neq \emptyset}{U \vdash \text{val}(\mu t.\overline{\tau}) \neq \emptyset}\ \text{I-REC}$$

Figure 8. Subtyping and value-(un)inhabitation rules for $L_{+\times}^{\to\mu}$.

4.1 A Subtyping Algorithm

The subtyping rules for $L_{+\times}^{\to\mu}$ rely on auxiliary value-inhabitation judgments. When deriving these auxiliary judgments, this paper assumes that rule I-SUM1 takes precedence over I-SUM2 (i.e., I-SUM2 gets used to try to derive $U \vdash \text{val}(\tau_1 + \tau_2) \neq \emptyset$ exactly when I-SUM1 has failed to derive $U \vdash \text{val}(\tau_1 + \tau_2) \neq \emptyset$). Similarly, U-PROD1 takes precedence over U-PROD2, U-REC2 takes precedence over U-REC1, S-$\bot$ takes precedence over all the other subtyping rules, S-$\bot$FUN takes precedence over S-FUN, and S-REC2 takes precedence over S-REC1.

With these precedence assumptions, the rules in Figure 8 are deterministic and algorithmic: at every point in attempting to derive a judgment, there's 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 τ at most once).

Hence, a simple algorithm for deciding whether $\tau_1 \leq \tau_2$ is to traverse the (possibly failing) derivation of $\tau_1 \leq \tau_2$, rejecting if and only if the derivation fails at some point (because one of the traversed judgments cannot be the conclusion of any inference rule). By the discussion above, this algorithm is deterministic and guaranteed to terminate. An implementation of this algorithm (optimized to have worst-case $O(n^2)$ running time, where n is the size of the types being considered) appears in Appendix A.

One way we have maintained the determinism of the subtyping system is by replacing explicit reflexivity and transitivity rules with rules S-NAT and S-REAL. Section 5 proves that the subtyping relation defined in Figure 8 is indeed reflexive and transitive for all types in $L_{+\times}^{\to\mu}$.

4.2 Induction on Failing Derivations

Because every subtyping/inhabitation derivation succeeds at a finite height, we can prove properties of valid judgments with the standard technique of induction on derivations. Dually, because every subtyping/inhabitation derivation fails at a finite height, we can prove properties of invalid (underivable) judgments with a technique that we call induction on failing derivations.

Every underivable judgment has a finite failing derivation, in which at least one leaf judgment in the derivation tree gets "stuck" (because that judgment cannot be the conclusion of any rule). Figures 4–5 illustrate failing derivations. Notice that all judgments between a failing leaf judgment and the root of the derivation tree must also be failing. Proofs by induction on failing derivations trace the failure from a leaf to the root of the failing derivation tree, showing that the desired property holds on every (underivable) 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.

As an example, consider proving a property P on underivable $S \vdash \tau_1 \leq \tau_2$ judgments by induction on failing derivations. The leaf nodes in a failing $S \vdash \tau_1 \leq \tau_2$ derivation can only occur when $\tau_1 = \texttt{real}$ and $\tau_2 = \texttt{nat}$, or when exactly one of τ_1 and τ_2 is a function/product/sum/recursive type; the base cases of the proof would therefore show that P holds on all such judgments. For the in-

ductive cases, first note that rules that take precedence over other rules cannot appear in failing derivations (because when such rules fail, the lower-precedence rules get used instead), so rules S-$\bot$, S-$\bot$FUN, and S-REC2 never appear in failing derivations. We're thus left with S-FUN, S-SUM, S-PROD, and S-REC1 as the only possible internal nodes in failing derivations of $S \vdash \tau_1 \leq \tau_2$, so in each of these four cases the proof would show, while inductively assuming that P holds on the rule's failing premises (of which there must be at least one), 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 τ_1 and τ_2, if there do not exist E, τ, e, and e' such that $E[\tau_2]{:}\tau$, $e{:}\tau_1$, $E[e] \mapsto^* e'$, and $\texttt{stuck}(e')$, 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 $\tau_1 \leq \tau_2$. Lemma 18's proof in Section 5 operates in this way to prove a strong version of completeness. Lemmas 1 and 12 are also proved with induction on failing derivations.

5. Proof of the Subtyping Relation's Preciseness

The following proof shows that the subtyping relation defined in Figure 8 is precise with respect to type safety. The proof provides a framework and techniques that may be useful for other preciseness proofs.

5.1 Basic Properties of the val Relations

The proof begins with many "sanity checks" on the $\leq$ and `val` relations (from Lemma 1 to Lemma 14). Lemma 1 and Corollary 2 show that, for all types τ, either $\text{val}(\tau){=}\emptyset$ is derivable or $\text{val}(\tau) \neq \emptyset$ is derivable, but not both.

Lemma 1. *Strong* val *Negation.*
$\forall U, \tau : (U \vdash \text{val}(\tau) = \emptyset$ *is not derivable* $\Leftrightarrow U \vdash \text{val}(\tau) \neq \emptyset)$

Proof. The if direction ($\Leftarrow$) is by straightforward induction on the derivation of $U \vdash \text{val}(\tau) \neq \emptyset$.

The only-if direction ($\Rightarrow$) is by induction on the failing derivation of $U \vdash \text{val}(\tau) = \emptyset$. The base cases (leaves) of a failing-derivation tree of $U \vdash \text{val}(\tau) = \emptyset$ occur when τ is `nat`, `real`, or a function type; in all these base cases $U \vdash \text{val}(\tau) = \emptyset$ is not derivable, but $U \vdash \text{val}(\tau) \neq \emptyset$ as required. The three inductive cases (i.e., inner judgments) of a failing-derivation tree of $U \vdash \text{val}(\tau) = \emptyset$ occur when (1) $\tau = \tau_1 + \tau_2$ and a premise of U-SUM isn't derivable, (2) $\tau = \tau_1 \times \tau_2$ and $U \vdash \text{val}(\tau_1) \neq \emptyset$ and the $U \vdash \text{val}(\tau_2) = \emptyset$ premise of U-PROD2 is not derivable, or (3) $\tau = \mu t.\overline{\tau}$ and $\mu t.\overline{\tau} \notin U$ and the $U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) = \emptyset$ premise of U-REC1 is not derivable. Note that a derivation of $U \vdash \text{val}(\tau) = \emptyset$ cannot fail on rule U-PROD1 because if U-PROD1's premise is not derivable then U-PROD2 will be used instead, which guarantees that a failing derivation rooted on a use of U-PROD2 has a valid premise of $U \vdash \text{val}(\tau_1) \neq \emptyset$ and a failing premise of $U \vdash \text{val}(\tau_2) = \emptyset$ (where $\tau = \tau_1 \times \tau_2$). A derivation of $U \vdash \text{val}(\tau) = \emptyset$ also cannot fail on U-REC2, because then U-REC1 would be used instead.

Now let's consider the 3 inductive cases outlined above.

(1) If a derivation of $U \vdash \text{val}(\tau_1 + \tau_2) = \emptyset$ fails because a premise of U-SUM fails, then by the inductive hypothesis we have either $U \vdash \text{val}(\tau_1) \neq \emptyset$ or $U \vdash \text{val}(\tau_2) \neq \emptyset$. In both subcases, $U \vdash \text{val}(\tau_1 + \tau_2) \neq \emptyset$ (by I-SUM1 or I-SUM2).

(2) If a derivation of $U \vdash \text{val}(\tau_1 \times \tau_2) = \emptyset$ fails because $U \vdash \text{val}(\tau_2) = \emptyset$ is not derivable but $U \vdash \text{val}(\tau_1) \neq \emptyset$, then the inductive hypothesis implies that $U \vdash \text{val}(\tau_2) \neq \emptyset$. Therefore, $U \vdash \text{val}(\tau_1 \times \tau_2) \neq \emptyset$ by I-PROD.

(3) If a derivation of $U \vdash \text{val}(\mu t.\overline{\tau}) = \emptyset$ fails because $U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) = \emptyset$ is not derivable and $\mu t.\overline{\tau} \notin U$, then the inductive hypothesis implies that $U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) \neq \emptyset$. Therefore, $U \vdash \text{val}(\mu t.\overline{\tau}) \neq \emptyset$ by I-REC.

In all cases of failing derivations of $U \vdash \text{val}(\tau) = \emptyset$, we have $U \vdash \text{val}(\tau) \neq \emptyset$. □

Corollary 2. val *Negation.*

$$\forall \tau : (\text{val}(\tau) = \emptyset \textit{ is not derivable} \Leftrightarrow \text{val}(\tau) \neq \emptyset)$$

Proof. Immediate by Lemma 1. □

5.2 Basic Properties of the Subtyping Relation

The next three lemmas establish some basic properties of the subtyping relation. First, Lemma 3 provides a standard subtyping-inversion result, though the result is complicated by the value-inhabitation premises in the subtyping rules. Lemma 4 shows that subtyping contexts can be weakened by adding assumptions, and Lemma 5 states that if a value-inhabited recursive type τ_1 is a subtype of another recursive type τ_2, then the unrolled version of τ_1 is a subtype of the unrolled version of τ_2.

Lemma 3. *Subtyping Inversion.*

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

A. $\text{val}(\tau_1) = \emptyset$*, or*
B. $\text{val}(\tau_1) \neq \emptyset$*,* $\tau_2 = \tau_2' \to \tau_2''$*, and* $\text{val}(\tau_2') = \emptyset$*, or*
C. Neither A nor B hold, and all of the following hold:
- *i.* $\tau_1 = \texttt{real} \Rightarrow \tau_2 = \texttt{real}$
- *ii.* $\tau_1 = \texttt{nat} \Rightarrow (\tau_2 = \texttt{real} \vee \tau_2 = \texttt{nat})$
- *iii.* $\tau_1 = \tau_1' \to \tau_1'' \Rightarrow (\tau_2 = \tau_2' \to \tau_2'' \wedge S \vdash \tau_2' \leq \tau_1' \wedge S \vdash \tau_1'' \leq \tau_2'')$
- *iv.* $\tau_1 = \tau_1' + \tau_1'' \Rightarrow (\tau_2 = \tau_2' + \tau_2'' \wedge S \vdash \tau_1' \leq \tau_2' \wedge S \vdash \tau_1'' \leq \tau_2'')$
- *v.* $\tau_1 = \tau_1' \times \tau_1'' \Rightarrow (\tau_2 = \tau_2' \times \tau_2'' \wedge S \vdash \tau_1' \leq \tau_2' \wedge S \vdash \tau_1'' \leq \tau_2'')$
- *vi.* $\tau_1 = \mu t.\overline{\tau} \Rightarrow \tau_2 = \mu t'.\overline{\tau}'$ *and either* $\tau_1 \leq \tau_2 \in S$ *or* $S \cup \{\tau_1 \leq \tau_2\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'$
- *vii.* $\tau_2 = \texttt{real} \Rightarrow (\tau_1 = \texttt{nat} \vee \tau_1 = \texttt{real})$
- *viii.* $\tau_2 = \texttt{nat} \Rightarrow \tau_1 = \texttt{nat}$
- *ix.* $\tau_2 = \tau_2' \to \tau_2'' \Rightarrow (\tau_1 = \tau_1' \to \tau_1'' \wedge S \vdash \tau_2' \leq \tau_1' \wedge S \vdash \tau_1'' \leq \tau_2'')$
- *x.* $\tau_2 = \tau_2' + \tau_2'' \Rightarrow (\tau_1 = \tau_1' + \tau_1'' \wedge S \vdash \tau_1' \leq \tau_2' \wedge S \vdash \tau_1'' \leq \tau_2'')$
- *xi.* $\tau_2 = \tau_2' \times \tau_2'' \Rightarrow (\tau_1 = \tau_1' \times \tau_1'' \wedge S \vdash \tau_1' \leq \tau_2' \wedge S \vdash \tau_1'' \leq \tau_2'')$
- *xii.* $\tau_2 = \mu t'.\overline{\tau}' \Rightarrow \tau_1 = \mu t.\overline{\tau}$ *and either* $\tau_1 \leq \tau_2 \in S$ *or* $S \cup \{\tau_1 \leq \tau_2\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'$

Proof. By straightforward case analysis of the rules deriving $S \vdash \tau_1 \leq \tau_2$. □

Lemma 4. *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 $S \vdash \tau_1 \leq \tau_2$. □

Lemma 5. *Empty Unrolled Subtyping.*

$$\forall t_1, t_2, \overline{\tau}_1, \overline{\tau}_2 : \begin{pmatrix} \text{val}(\mu t_1.\overline{\tau}_1) \neq \emptyset \\ \wedge\, \mu t_1.\overline{\tau}_1 \leq \mu t_2.\overline{\tau}_2 \end{pmatrix} \Rightarrow [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2$$

Proof. Let $\tau_1 = \mu t_1.\overline{\tau}_1$, $\tau_2 = \mu t_2.\overline{\tau}_2$, $\tau_{1u} = [\tau_1/t_1]\overline{\tau}_1$, and $\tau_{2u} = [\tau_2/t_2]\overline{\tau}_2$. Because $\text{val}(\tau_1) \neq \emptyset$, $\tau_1 \leq \tau_2$ is only derivable with S-REC1, so its premise must be derivable, i.e., $\{\tau_1 \leq \tau_2\} \vdash \tau_{1u} \leq \tau_{2u}$. The derivation of $\tau_1 \leq \tau_2$ therefore has the form

$$\frac{\text{val}(\tau_1) \neq \emptyset \qquad \dfrac{D}{\{\tau_1 \leq \tau_2\} \vdash \tau_{1u} \leq \tau_{2u}}}{\tau_1 \leq \tau_2}$$

for some derivation forest D. Because $\tau_1 \leq \tau_2$, there also exists a derivation of $S \vdash \tau_1 \leq \tau_2$ (for all subtyping-assumption sets S), by Lemma 4.

Now let's consider a new derivation forest D' constructed by modifying D in 2 ways: (1) whenever a judgment of the form $S \cup \{\tau_1 \leq \tau_2\} \vdash \tau_1 \leq \tau_2$ (i.e., a use of S-REC2 on τ_1 and τ_2) appears in D, D' replaces it with the derivation of $S \vdash \tau_1 \leq \tau_2$, and (2) whenever a judgment of the form $S' \vdash \tau_1' \leq \tau_2'$ appears in D (besides a use of S-REC2 on τ_1 and τ_2), D' replaces it with the judgment $S' \backslash \{\tau_1 \leq \tau_2\} \vdash \tau_1' \leq \tau_2'$. Observe that D' is the same as D, except D' doesn't require the initial $\tau_1 \leq \tau_2$ assumption; whenever D uses the $\tau_1 \leq \tau_2$ assumption (with rule S-REC2), D' just derives $\tau_1 \leq \tau_2$ directly. Hence, because $\dfrac{D}{\{\tau_1 \leq \tau_2\} \vdash \tau_{1u} \leq \tau_{2u}}$, $\dfrac{D'}{\tau_{1u} \leq \tau_{2u}}$. Thus, $\tau_{1u} \leq \tau_{2u}$ as required. □

5.3 Relationships between the val and Subtyping Relations

This subsection relates the val and subtyping relations, to establish three useful results: (1) if a type is value inhabited then its supertypes must also be value inhabited (Lemma 7), (2) a type τ is inhabited by values exactly when $\text{val}(\tau) \neq \emptyset$ (Lemma 8), and (3) if a type is value uninhabited then its subtypes must also be value uninhabited (Corollary 11).

We prove these results by first replacing rule I-REC with the following alternate rule:

$$\frac{U \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) \neq \emptyset}{U \vdash \text{val}(\mu t.\overline{\tau}) \neq \emptyset}\ \text{I-REC}_2$$

Lemma 6 shows that the deductive system for judging $U \vdash \text{val}(\tau) \neq \emptyset$, as defined in Figure 8, is equivalent to the system obtained by replacing rule I-REC with I-REC$_2$.

Lemma 6. *Equivalence of Systems with* I-REC *and* I-REC$_2$.

$\forall \tau$: ($\text{val}(\tau) \neq \emptyset$ *in the inhabitation system with rule* I-REC
$\Leftrightarrow \text{val}(\tau) \neq \emptyset$ *in the inhabitation system with rule* I-REC$_2$)

Proof. For clarity, we'll write $\text{val}(\tau) \neq_2 \emptyset$ when $\text{val}(\tau) \neq \emptyset$ is derivable in the inhabitation system with rule I-REC$_2$ replacing I-REC. Note that all the rules defining $\neq$ and $\neq_2$ are identical, except that the $\neq$ rules stop (failing) derivation trees (with rule I-REC) at any point of considering a $\tau = \mu t.\overline{\tau}$ that has been considered before (i.e., lower/closer to the root of the derivation tree). Derivations that never get stopped in this way using the $\neq$ rules will not get stopped with the $\neq_2$ rules, because the $\neq_2$ rules have no such stopping mechanism (with I-REC$_2$). Hence, $\text{val}(\tau) \neq \emptyset \Rightarrow \text{val}(\tau) \neq_2 \emptyset$.

On the other hand, a derivation of $\text{val}(\tau) \neq_2 \emptyset$ must never consider a $\tau = \mu t.\overline{\tau}$ that has been considered before (lower) in the derivation tree (if it did, the fact that the $\neq_2$ rules are deterministic implies that $\tau = \mu t.\overline{\tau}$ would have to be considered again and again, infinitely, which would prevent $\text{val}(\tau) \neq_2 \emptyset$ from being derived). Hence, $\text{val}(\tau) \neq_2 \emptyset \Rightarrow \text{val}(\tau) \neq \emptyset$. □

Given that replacing I-REC with I-REC$_2$ has no effect on which $\text{val}(\tau) \neq \emptyset$ judgments are derivable, the remainder of this subsection's proofs will assume that valid $\text{val}(\tau) \neq \emptyset$ judgments are derivable with only the I-NAT, I-REAL, I-FUN, I-SUM1, I-SUM2, I-PROD, and I-REC$_2$ rules.

The following lemma shows that value inhabitation is closed under supertyping.

Lemma 7. *Value Inhabitation is Closed under Supertyping.*

$$\forall \tau, \tau' : ((\text{val}(\tau) \neq \emptyset \wedge \tau \leq \tau') \Rightarrow \text{val}(\tau') \neq \emptyset)$$

Proof. By induction on the derivation of $\text{val}(\tau) \neq \emptyset$, where the derivation uses rule I-REC$_2$ rather than I-REC (which is valid by Lemma 6). Note that in all cases, $\text{val}(\tau) = \emptyset$ is not derivable (by Corollary 2), and if τ' is a function type then $\text{val}(\tau') \neq \emptyset$ immediately by I-FUN. Hence, in all cases of deriving $\text{val}(\tau) \neq \emptyset$ with $\tau \leq \tau'$, we have eliminated the A and B subcases of Lemma 3 and can assume that subcase C of Lemma 3 characterizes the relationship between τ and τ'.

$$\text{Case}\ \frac{}{\text{val}(\texttt{nat}) \neq \emptyset}\ \text{I-NAT}$$

By Lemma 3(C)(ii), $\tau' = \texttt{real}$ or $\tau' = \texttt{nat}$, so $\text{val}(\tau') \neq \emptyset$ by I-NAT or I-REAL.

The I-REAL and I-FUN cases are proved similarly to the I-NAT case.

$$\text{Case}\ \frac{\text{val}(\tau_1) \neq \emptyset}{\text{val}(\tau_1 + \tau_2) \neq \emptyset}\ \text{I-SUM1}$$

By Lemma 3(C)(iv), $\tau' = \tau_1' + \tau_2'$ and $\tau_1 \leq \tau_1'$. By assumption, $\text{val}(\tau_1) \neq \emptyset$, so by the inductive hypothesis, $\text{val}(\tau_1') \neq \emptyset$, which implies $\text{val}(\tau') \neq \emptyset$ by I-SUM1.

The I-SUM2 and I-PROD cases are proved similarly to the I-SUM1 case.

$$\text{Case}\ \frac{\text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) \neq \emptyset}{\text{val}(\mu t.\overline{\tau}) \neq \emptyset}\ \text{I-REC}_2$$

By Lemma 3(C)(vi), $\tau' = \mu t'.\overline{\tau}'$, and by assumption, $\text{val}(\mu t.\overline{\tau}) \neq \emptyset$ and $\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'$. Lemma 5 thus implies $[\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']/\overline{\tau}'$, which combines with the $\text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) \neq \emptyset$ assumption and the inductive hypothesis to provide that $\text{val}([\mu t'.\overline{\tau}'/t']\overline{\tau}') \neq \emptyset$. Hence, by rule I-REC$_2$, $\text{val}(\mu t'.\overline{\tau}') \neq \emptyset$, as required. □

Now we can prove that the val judgments mean what we want them to mean: $\text{val}(\tau) \neq \emptyset$ is derivable exactly when there exists a value of type τ.

Lemma 8. *Value Inhabitation.*

$$\forall \tau : (\text{val}(\tau) \neq \emptyset \Leftrightarrow \exists v : (v{:}\tau))$$

Proof. We again assume that derivations of $\text{val}(\tau) \neq \emptyset$ use rule I-REC$_2$ instead of I-REC (which is valid by Lemma 6). The if direction ($\Leftarrow$) is by induction on the derivation of $v{:}\tau$. The T-NAT, T-REAL, and T-FUN cases are immediate with rules I-NAT, I-REAL, and I-FUN. The remaining cases deriving $v{:}\tau$ are T-LEFT, T-RIGHT, T-PROD, T-ROLL, and T-SUBSUME; the first four of these are all proved similarly. For example, the T-LEFT case assumes $v = \texttt{inl}_\tau v_1$, where $\tau = \tau_1 + \tau_2$ and $v_1{:}\tau_1$. By the inductive hypothesis, we have $\text{val}(\tau_1) \neq \emptyset$, implying by rule I-SUM1 that $\text{val}(\tau) \neq \emptyset$. For the T-SUBSUME case, we assume $v{:}\tau'$ and $\tau' \leq \tau$. By the inductive hypothesis, then, $\text{val}(\tau') \neq \emptyset$, so by Lemma 7, $\text{val}(\tau) \neq \emptyset$ as required.

The only-if direction ($\Rightarrow$) is by induction on the derivation of $\text{val}(\tau) \neq \emptyset$. When $\tau = \texttt{nat}$, let $v = 0$; then $v{:}\texttt{nat}$. When $\tau = \tau_1 \to \tau_2$, let $v = (\texttt{fun} f(x{:}\tau_1){:}\tau_2 = f(x))$; then $v{:}\tau_1 \to \tau_2$. When $\tau = \tau_1 + \tau_2$ and $\text{val}(\tau) \neq \emptyset$ with rule I-SUM1, the inductive hypothesis provides that $\exists v_1{:}(v_1{:}\tau_1)$; hence we let $v = \texttt{inl}_\tau v_1$, so $v{:}\tau$ by T-LEFT. All the other cases are proved similarly to the I-NAT and I-SUM1 cases. □

The following lemma is a simple context-weakening result.

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 $U \vdash \text{val}(\tau) = \emptyset$. □

Now we're ready to prove that subtypes of value-uninhabited types are also uninhabited. Lemma 10 is a strong version of this result; the extra assumptions in Lemma 10 provide a strong enough inductive hypothesis to prove the result in all cases. Then the weaker but desired version follows as a corollary.

Lemma 10. *Strong Value Uninhabitation Closed under Subtyping.*

$$\forall S, S_1, S_2, \tau_1, \tau_2 : \begin{pmatrix} S \vdash \tau_1 \leq \tau_2 \\ \wedge\, S_1 = \{\tau_1 | \tau_1 \leq \tau_2 \in S\} \\ \wedge\, S_2 = \{\tau_2 | \tau_1 \leq \tau_2 \in S\} \\ \wedge\, S_2 \vdash \mathrm{val}(\tau_2) = \emptyset \end{pmatrix} \Rightarrow S_1 \vdash \mathrm{val}(\tau_1) = \emptyset$$

Proof. By induction on the derivation of $S \vdash \tau_1 \leq \tau_2$. Cases S-BASE, S-NAT, S-REAL, S-⊥FUN, and S-FUN hold vacuously because they don't allow $S_2 \vdash \mathrm{val}(\tau_2) = \emptyset$. Case S-⊥ holds because its premise ensures that $\mathrm{val}(\tau_1) = \emptyset$, so by Lemma 9, $S_1 \vdash \mathrm{val}(\tau_1) = \emptyset$. Case S-REC2 assumes $\tau_1 = \mu t.\overline{\tau}, \tau_2 = \mu t'.\overline{\tau}'$, and $\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}' \in S$, so $\mu t.\overline{\tau} \in S_1$ and, by rule U-REC2, $S_1 \vdash \mathrm{val}(\tau_1) = \emptyset$. The three remaining cases (S-PROD, S-SUM, and S-REC1) are all proved with straightforward inductive arguments. □

Corollary 11. *Value Uninhabitation is Closed under Subtyping.*

$$\forall \tau_1, \tau_2 : ((\tau_1 \leq \tau_2 \wedge \mathrm{val}(\tau_2) = \emptyset) \Rightarrow \mathrm{val}(\tau_1) = \emptyset)$$

Proof. Immediate from Lemma 10, where $S = S_1 = S_2 = \emptyset$. □

5.4 Subtyping Reflexivity and Transitivity

For the sake of determinism, the subtyping relation under consideration lacks explicit reflexivity and transitivity rules. This subsection shows that the subtyping relation is nonetheless reflexive and transitive.

Lemma 12. *Strong Subtyping Reflexivity.*

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

Proof. By induction on the failing derivation of $S \vdash \tau_1 \leq \tau_2$. This derivation can fail when $\tau_1 = \texttt{real}$ and $\tau_2 = \texttt{nat}$, or when exactly one of τ_1 and τ_2 is a function/product/sum/recursive type. In all these cases, $\tau_1 \neq \tau_2$, as required. Moreover, these are the only base cases of a failing derivation of $S \vdash \tau_1 \leq \tau_2$ (i.e., they're the only possible leaf nodes in a failing derivation tree rooted at $S \vdash \tau_1 \leq \tau_2$).

The inductive cases of a failing derivation of $S \vdash \tau_1 \leq \tau_2$ occur when $S \vdash \tau_1 \leq \tau_2$ fails due to a premise being underivable (i.e., the inductive cases occur at internal nodes in a failing derivation tree). The only inductive cases of a failing $S \vdash \tau_1 \leq \tau_2$ derivation are uses of S-FUN, S-SUM, S-PROD, or S-REC1 (failure cannot occur with S-REC2 because then S-REC1 would be used). In all these inductive cases, the failing premise must be of the form $S' \vdash \tau_1' \leq \tau_2'$, because premises of other forms are only used to control which rule must be used at each step of a (possibly failing) derivation of $S \vdash \tau_1 \leq \tau_2$ (in other words, premises of other forms are only used to make the derivations deterministic).

In all the four inductive cases S-FUN, S-SUM, S-PROD, and S-REC1, a failing premise of the form $S' \vdash \tau_1' \leq \tau_2'$ implies, by the inductive hypothesis, that $\tau_1' \neq \tau_2'$, which guarantees that $\tau_1 \neq \tau_2$. For example, in the S-FUN case, $\tau_1 = \tau_1' \to \tau_1''$ and $\tau_2 = \tau_2' \to \tau_2''$, and for $S \vdash \tau_1 \leq \tau_2$ to fail, $S \vdash \tau_2' \leq \tau_1'$ fails or $S \vdash \tau_1'' \leq \tau_2''$ fails. Hence, by the inductive hypothesis, $\tau_1' \neq \tau_2'$ or $\tau_1'' \neq \tau_2''$, so $\tau_1 \neq \tau_2$. The other inductive cases are proved similarly. □

Corollary 13. *Subtyping Reflexivity.*

$$\forall \tau : \tau \leq \tau$$

Proof. Immediate by Lemma 12. □

Lemma 14. *Subtyping Transitivity.*

$$\forall \tau_1, \tau_2, \tau_3 : ((\tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_3) \Rightarrow \tau_1 \leq \tau_3)$$

Proof. We consider all the possibilities obtained by applying Lemma 3 to the assumptions that (1) $\tau_1 \leq \tau_2$ and (2) $\tau_2 \leq \tau_3$. If assumption (1) satisfies (A) of Lemma 3, then $\tau_1 \leq \tau_3$ by rule S-⊥. If assumption (1) doesn't satisfy (A) of Lemma 3, then assumption (2) can't satisfy (A); otherwise we'd have $\mathrm{val}(\tau_1) \neq \emptyset$, $\tau_1 \leq \tau_2$, and $\mathrm{val}(\tau_2) = \emptyset$, which would contradict Corollary 11. If assumption (2) satisfies (B) of Lemma 3, then $\tau_1 \leq \tau_3$ by rule S-⊥FUN. If assumption (1) satisfies (B) and (2) satisfies (C), then $\tau_2 = \tau_2' \to \tau_2''$, $\mathrm{val}(\tau_2') = \emptyset$, $\tau_3 = \tau_3' \to \tau_3''$, $\mathrm{val}(\tau_3') \neq \emptyset$, and $\tau_3' \leq \tau_2'$. But $\tau_3' \leq \tau_2'$ cannot occur when $\mathrm{val}(\tau_3') \neq \emptyset$ and $\mathrm{val}(\tau_2') = \emptyset$ (by Corollary 11), so the lemma holds vacuously in this case.

The only remaining combination to consider is that both assumptions (1) and (2) satisfy (C) of Lemma 3. We proceed by induction on the derivations of $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_3$. If $\tau_1 = \texttt{real}$ then (by Lemma 3) $\tau_2 = \texttt{real}$, implying (again by Lemma 3) that $\tau_3 = \texttt{real}$, so $\tau_1 \leq \tau_3$ by S-REAL. If $\tau_1 = \texttt{nat}$ then by Lemma 3, $\tau_3 = \texttt{real}$ or $\tau_3 = \texttt{nat}$; in both subcases $\tau_1 \leq \tau_3$.

If $\tau_1 = \tau_1' \to \tau_1''$ then because assumptions (1) and (2) satisfy (C) of Lemma 3, we have $\tau_2 = \tau_2' \to \tau_2''$, $\tau_2' \leq \tau_1'$, $\tau_1'' \leq \tau_2''$, $\tau_3 = \tau_3' \to \tau_3''$, $\tau_3' \leq \tau_2'$, $\tau_2'' \leq \tau_3''$, $\mathrm{val}(\tau_2') \neq \emptyset$, and $\mathrm{val}(\tau_3') \neq \emptyset$. Then by the inductive hypothesis, applied to $\tau_3' \leq \tau_2'$ and $\tau_2' \leq \tau_1'$, and to $\tau_1'' \leq \tau_2''$ and $\tau_2'' \leq \tau_3''$, we have $\tau_3' \leq \tau_1'$ and $\tau_1'' \leq \tau_3''$. Because $\mathrm{val}(\tau_3') \neq \emptyset$, then, S-FUN implies that $\tau_1 \leq \tau_3$. The cases of $\tau_1 = \tau_1' + \tau_1''$ and $\tau_1 = \tau_1' \times \tau_1''$ (i.e., S-SUM and S-PROD) are proved similarly.

The final case to consider is that $\tau_1 = \mu t_1.\overline{\tau}_1$. In this case, because assumptions (1) and (2) satisfy (C) of Lemma 3, we have $\tau_2 = \mu t_2.\overline{\tau}_2$, $\tau_3 = \mu t_3.\overline{\tau}_3$, $\mathrm{val}(\tau_1) \neq \emptyset$, and $\mathrm{val}(\tau_2) \neq \emptyset$. Combining these results with Lemma 5, then, we obtain $[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2$ and $[\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2 \leq [\mu t_3.\overline{\tau}_3/t_3]\overline{\tau}_3$. Applying the inductive hypothesis to these two subtyping judgments yields $[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_3.\overline{\tau}_3/t_3]\overline{\tau}_3$, so by Lemma 4, $\{\tau_1 \leq \tau_3\} \vdash [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_3.\overline{\tau}_3/t_3]\overline{\tau}_3$. Finally, because $\mathrm{val}(\tau_1) \neq \emptyset$ and $\{\tau_1 \leq \tau_3\} \vdash [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_3.\overline{\tau}_3/t_3]\overline{\tau}_3$, rule S-REC1 ensures that $\tau_1 \leq \tau_3$, as required. □

5.5 Properties of the Static and Dynamic Semantics

Having completed the "sanity checks" on the $\leq$ and val relations, Lemmas 15–17 present standard inversion, weakening, and canonical-forms lemmas, which are used to prove both completeness and soundness.

Lemma 15. *Typing Inversion.*

A. $\Gamma \vdash \texttt{n} : \tau \Rightarrow \texttt{nat} \leq \tau$

B. $\Gamma \vdash \texttt{r} : \tau \Rightarrow \texttt{real} \leq \tau$

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

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. $\Gamma \vdash \texttt{sqrt}(e) : \tau \Rightarrow (\Gamma \vdash e : \texttt{real} \wedge \texttt{real} \leq \tau)$

F. $\Gamma \vdash (\texttt{fun}\ f(x{:}\tau_1){:}\tau_2 = e) : \tau \Rightarrow$
$(\Gamma \cup \{f{:}\tau_1 \to \tau_2, x{:}\tau_1\} \vdash e : \tau_2 \wedge \tau_1 \to \tau_2 \leq \tau)$

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. $\Gamma \vdash \texttt{inl}_{\tau'} e : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e : \tau_1 \wedge \tau_1 + \tau_2 \leq \tau \wedge \tau' = \tau_1 + \tau_2)$

I. $\Gamma \vdash \texttt{inr}_{\tau'} e : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e : \tau_2 \wedge \tau_1 + \tau_2 \leq \tau \wedge \tau' = \tau_1 + \tau_2)$

J. $\Gamma \vdash (\texttt{case}\ e_1\ \texttt{of inl}\, x \Rightarrow e_2\ \texttt{else inr}\, y \Rightarrow e_3) : \tau \Rightarrow$
$\exists \tau_1, \tau_2, \tau' : \quad \Gamma \vdash e_1 : \tau_1 + \tau_2 \wedge \Gamma \cup \{x{:}\tau_1\} \vdash e_2 : \tau'$
$\wedge\, \Gamma \cup \{y{:}\tau_2\} \vdash e_3 : \tau' \wedge \tau' \leq \tau$

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

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

M. $\Gamma \vdash \texttt{roll}(e) : \tau \Rightarrow \exists t, \overline{\tau} : (\Gamma \vdash e : [\mu t.\overline{\tau}/t]\overline{\tau} \wedge \mu t.\overline{\tau} \leq \tau)$

N. $\Gamma \vdash \texttt{unroll}(e) : \tau \Rightarrow \exists t, \overline{\tau} : (\Gamma \vdash e : \mu t.\overline{\tau} \wedge [\mu t.\overline{\tau}/t]\overline{\tau} \leq \tau)$

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 2 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, $\Gamma \vdash e.\texttt{fst} : \tau$ is derivable with T-SUBSUME and T-FST. With T-SUBSUME, the inductive hypothesis implies $\Gamma \vdash e : \tau_1 \times \tau_2$ and $\tau_1 \leq \tau'$, for a type τ' such that $\tau' \leq \tau$. By Lemma 14 then, $\tau_1 \leq \tau$, as required. If $\Gamma \vdash e.\texttt{fst} : \tau$ is derived with T-FST, we can assume $\Gamma \vdash e : \tau_1 \times \tau_2$ and $\tau = \tau_1$. By Lemma 13 then, $\tau_1 \leq \tau$, as required. All the other cases are proved similarly. □

Lemma 16. *Weakening.*

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

Proof. By straightforward induction on the derivation of $\Gamma \vdash e{:}\tau$. □

Lemma 17. *Canonical Forms.*

$$\forall v, \tau : \textit{If } v{:}\tau \textit{ then}$$

A. $\tau{=}\texttt{nat} \Rightarrow v{=}\texttt{n}$ *(for some* $\texttt{n}$*)*
B. $\tau{=}\texttt{real} \Rightarrow v{=}\texttt{n}$ *or* $v{=}\texttt{r}$ *(for some* $\texttt{n}$ *or* $\texttt{r}$*)*
C. $(\tau{=}\tau_1{\to}\tau_2 \wedge \text{val}(\tau_1) \neq \emptyset) \Rightarrow v = (\texttt{fun}\ f(x : \tau_3) : \tau_4 = e)$ *(for some* f*,* x*,* τ_3*,* τ_4*, and* e*)*
D. $\tau{=}\tau_1{+}\tau_2 \Rightarrow v{=}\texttt{inl}_{\tau'} v'$ *or* $v{=}\texttt{inr}_{\tau'} v'$ *(for some* τ' *and* v'*)*
E. $\tau{=}\tau_1 \times \tau_2 \Rightarrow v{=}(v_1, v_2)$ *(for some* v_1 *and* v_2*)*
F. $\tau{=}\mu t.\overline{\tau} \Rightarrow v{=}\texttt{roll}(v')$ *(for some* v'*)*

Proof. By induction on the derivation of $v{:}\tau$. The only nontrivial case is T-SUBSUME, in which $v{:}\tau'$, $v{:}\tau$, and $\tau' \leq \tau$. Because $v{:}\tau'$ and $v{:}\tau$, Lemma 8 ensures that $\text{val}(\tau') \neq \emptyset$ and $\text{val}(\tau) \neq \emptyset$. We next consider each of the six cases in the lemma statement and show that the desired result holds in every case. If $\tau = \texttt{real}$ then by Lemma 3, $\tau' = \texttt{nat}$ or $\tau' = \texttt{real}$, so by the inductive hypothesis (applied to $v{:}\tau'$), $v = \texttt{n}$ or $v = \texttt{r}$. If $\tau = \texttt{nat}$ then by Lemma 3, $\tau' = \texttt{nat}$, so by the inductive hypothesis, $v = \texttt{n}$. If $\tau = \tau_1{\to}\tau_2$, and $\text{val}(\tau_1) \neq \emptyset$, then by Lemma 3, $\tau' = \tau_1'{\to}\tau_2'$ and $\tau_1 \leq \tau_1'$. Because $\text{val}(\tau_1) \neq \emptyset$ and $\tau_1 \leq \tau_1'$, Lemma 7 ensures that $\text{val}(\tau_1') \neq \emptyset$. Then applying the inductive hypothesis to $v{:}\tau'$, where $\tau' = \tau_1'{\to}\tau_2'$ and $\text{val}(\tau_1') \neq \emptyset$, we find that $v = (\texttt{fun}\ f(x{:}\tau_3) : \tau_4 = e)$, as required. If $\tau = \tau_1{+}\tau_2$ then by Lemma 3, $\tau' = \tau_1'{+}\tau_2'$, so by the inductive hypothesis, $v = \texttt{inl}_{\tau'} v'$ or $v = \texttt{inr}_{\tau'} v'$. If $\tau = \tau_1 \times \tau_2$ then by Lemma 3, $\tau' = \tau_1' \times \tau_2'$, so by the inductive hypothesis, $v = (v_1, v_2)$. Finally, if $\tau = \mu t.\overline{\tau}$ then by Lemma 3, $\tau' = \mu t'.\overline{\tau}'$, so by the inductive hypothesis, $v = \texttt{roll}(v')$. □

5.6 Subtyping Completeness

We're now ready to state and prove the key lemma used to show completeness, Lemma 18. This lemma is a stronger version of completeness; the stronger version provides a strong enough inductive hypothesis to prove completeness in all cases.

The proof of Lemma 18 is constructive; given any S, τ_1, and τ_2 such that $S{\vdash}\tau_1{\leq}\tau_2$ is not derivable, the proof shows how to construct a well-typed program that gets stuck when its τ_2-type subexpression is replaced by a τ_1-type value.

Lemma 18. *Strong Completeness.*

$$\forall S, \tau_1, \tau_2 : \textit{If } S \vdash \tau_1{\leq}\tau_2 \textit{ is not derivable, then } \exists\, E, \tau, v, e :$$
$$(E[\tau_2] : \tau \wedge v : \tau_1 \wedge E[v] \mapsto^* e \wedge \texttt{stuck}(e))$$

Proof. The proof is by induction on the failing derivation of $S \vdash \tau_1{\leq}\tau_2$. This derivation can fail when $\tau_1 = \texttt{real}$ and $\tau_2 = \texttt{nat}$, or when exactly one of τ_1 and τ_2 is a function/product/sum/recursive type (and $\text{val}(\tau_1) \neq \emptyset$ and τ_2 is not a function type with uninhabited argument type). These are the only base cases of a failing derivation of $S \vdash \tau_1 \leq \tau_2$ (i.e., they're the only possible leaf nodes in a failing derivation tree rooted at $S \vdash \tau_1 \leq \tau_2$). We first prove the lemma for these cases.

Case $\tau_1 = \texttt{real}$ and $\tau_2 = \texttt{nat}$:
Let $E = \texttt{succ}([\,])$, $\tau = \texttt{nat}$, $v = 0.5$, and $e = \texttt{succ}(0.5)$. Then $v{:}\tau_1$ (by T-REAL), $E[\tau_2]{:}\tau$ (by T-CTXT and T-SUCC), $E[v] \mapsto^* e$ (by MSTEP-REFL), and $\texttt{stuck}(e)$.

Case $\tau_1 = \tau_1'{\to}\tau_1''$ and $\tau_2 \neq \tau_2'{\to}\tau_2''$:
Let $v = (\texttt{fun}\ f(x{:}\tau_1') : \tau_1'' = f(x))$, so $v{:}\tau_1$ by rules T-FUN and T-APP. Define E and τ as follows:

$$E{=}\begin{cases}\texttt{sqrt}([\,]) & \texttt{if } \tau_2 = \texttt{nat or } \tau_2 = \texttt{real}\\ \texttt{case}\ [\,]\ \texttt{of inl}\ x'{\Rightarrow}0.5 & \\ \quad \texttt{else inr}\ y'{\Rightarrow}0.5 & \texttt{if } \tau_2 = \tau_2'{+}\tau_2''\\ [\,].\texttt{snd} & \texttt{if } \tau_2 = \tau_2' \times \tau_2''\\ \texttt{unroll}([\,]) & \texttt{if } \tau_2 = \mu t.\overline{\tau}\end{cases}$$
$$\tau{=}\begin{cases}\texttt{real} & \texttt{if } \tau_2 = \texttt{nat or } \tau_2 = \texttt{real or } \tau_2 = \tau_2'{+}\tau_2''\\ \tau_2'' & \texttt{if } \tau_2 = \tau_2' \times \tau_2''\\ [\mu t.\overline{\tau}/t]\,\overline{\tau} & \texttt{if } \tau_2 = \mu t.\overline{\tau}\end{cases}$$

Then $E[\tau_2]{:}\tau$, by the definitions of E and τ and the typing rules. Moreover, let $e = E[v]$, so $E[v] \mapsto^* e$ and $\texttt{stuck}(e)$ (because $\texttt{stuck}(E[v])$, where $v = (\texttt{fun}\ f(x : \tau_1') : \tau_1'' = f(x))$).

Case $\tau_1{\neq}\tau_1'{\to}\tau_1''$ and $\tau_2{=}\tau_2'{\to}\tau_2''$ ($\text{val}(\tau_1){\neq}\emptyset$, $\text{val}(\tau_2'){\neq}\emptyset$):
By assumption, $\text{val}(\tau_1) \neq \emptyset$ and $\text{val}(\tau_2') \neq \emptyset$, so by Lemma 8 there exist v and v_2' such that $v : \tau_1$ and $v_2' : \tau_2'$. Because $\tau_1 \neq \tau_1'{\to}\tau_1''$ and $v : \tau_1$, Lemma 17 implies that $v \neq (\texttt{fun}\ f_3(x_3 : \tau_3) : \tau_3' = e_3)$ (for all f_3, x_3, τ_3, τ_3', and e_3). Let $E = [\,](v_2')$, $\tau = \tau_2''$, and $e = v(v_2')$. Then $E[\tau_2]{:}\tau$ (because $\tau_2 = \tau_2'{\to}\tau_2''$, $v_2'{:}\tau_2'$, and $\tau = \tau_2''$). Moreover, $E[v] = e$, so $E[v] \mapsto^* e$, and $\texttt{stuck}(e)$ (because $e = v(v_2')$, where v cannot be a function value).

Case $\tau_1 = \mu t_1.\overline{\tau}_1$ and $\tau_2 \neq \mu t_2.\overline{\tau}_2$ ($\text{val}(\tau_1) \neq \emptyset$):
By assumption, $\text{val}(\tau_1) \neq \emptyset$, so by Lemma 8 there exists a v such that $v{:}\mu t_1.\overline{\tau}_1$. Hence, by Lemma 17, $v = \texttt{roll}(v')$ for some value v'. Define E and τ as follows:

$$E{=}\begin{cases}\texttt{sqrt}([\,]) & \texttt{if } \tau_2 = \texttt{nat or } \tau_2 = \texttt{real}\\ \texttt{case}\ [\,]\ \texttt{of inl}\ x{\Rightarrow}0.5 & \\ \quad \texttt{else inr}\ y{\Rightarrow}0.5 & \texttt{if } \tau_2 = \tau_2'{+}\tau_2''\\ [\,].\texttt{snd} & \texttt{if } \tau_2 = \tau_2' \times \tau_2''\\ [\,]((\texttt{fun}\ f(x{:}\texttt{nat}){:}\tau_2'{=}f(x))(0)) & \texttt{if } \tau_2 = \tau_2'{\to}\tau_2''\end{cases}$$
$$\tau{=}\begin{cases}\texttt{real} & \texttt{if } \tau_2 = \texttt{nat or } \tau_2 = \texttt{real or } \tau_2 = \tau_2'{+}\tau_2''\\ \tau_2'' & \texttt{if } \tau_2 = \tau_2' \times \tau_2'' \texttt{ or } \tau_2 = \tau_2'{\to}\tau_2''\end{cases}$$

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

Case $\tau_1 \neq \mu t_1.\overline{\tau}_1$ and $\tau_2 = \mu t_2.\overline{\tau}_2$ ($\text{val}(\tau_1) \neq \emptyset$):
There are two subcases to consider, either (1) $\tau_1 = \tau_1'{\to}\tau_1''$ and $\text{val}(\tau_1') = \emptyset$, or (2) $(\tau_1 = \tau_1'{\to}\tau_1'') \Rightarrow (\text{val}(\tau_1') \neq \emptyset)$. In subcase (1), let $v = 0$, so $v{:}\texttt{nat}$ and $v{:}\tau_1$ by T-SUBSUME and S-⊥FUN. In subcase (2), we use the assumption that $\text{val}(\tau_1) \neq \emptyset$ and Lemma 8 to obtain a v such that $v{:}\tau_1$. Also in subcase (2), we know that $v{:}\tau_1$ but $\tau_1 \neq \mu t_1.\overline{\tau}_1$ and $(\tau_1 = \tau_1'{\to}\tau_1'') \Rightarrow (\text{val}(\tau_1') \neq \emptyset)$, so Lemma 17 implies that $v \neq \texttt{roll}(v')$ (for all v'). Hence, in all subcases, $v{:}\tau_1$ and $v \neq \texttt{roll}(v')$ (for all v'). Next, let $E = \texttt{unroll}([\,])$, $\tau = [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2$, and $e = \texttt{unroll}(v)$. Then $E[\tau_2]{:}\tau$ by T-CTXT, T-VAR, and T-UNROLL. Moreover, $E[v] = e$, so $E[v] \mapsto^* e$, and $\texttt{stuck}(e)$ (because $e = \texttt{unroll}(v)$, where v can't be a rolled value).

The remaining leaf cases (where exactly one of τ_1 and τ_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 when $S \vdash \tau_1 \leq \tau_2$ fails due to a premise being underivable (i.e., the inductive cases occur at internal nodes in a failing derivation tree). The only inductive cases of a failing $S \vdash \tau_1 \leq \tau_2$ derivation

are uses of S-FUN, S-SUM, S-PROD, or S-REC1 (failure cannot occur with S-REC2 because then S-REC1 would be used instead). In all these inductive cases, the failing premise must be of the form $S' \vdash \tau_1' \leq \tau_2'$, because premises of other forms are only used to control which rule must be used at each step of a (possibly failing) derivation of $S \vdash \tau_1 \leq \tau_2$ (in other words, premises of other forms are only used to make the derivations deterministic). Also in all these cases we have $\mathrm{val}(\tau_1) \neq \emptyset$ and $\tau_2 \neq \tau_2' \to \tau_2''$ such that $\mathrm{val}(\tau_2') = \emptyset$ (otherwise rule S-$\bot$ or S-$\bot$Fun would have been used to derive $S \vdash \tau_1 \leq \tau_2$).

$$\text{Case } \frac{\mathrm{val}(\tau_2') \neq \emptyset \qquad S \vdash \ \tau_2' \leq \tau_1' \qquad S \vdash \ \tau_1'' \leq \tau_2''}{S \vdash \ \tau_1' \to \tau_1'' \leq \tau_2' \to \tau_2''} \text{ S-FUN}$$

In this case $S \vdash \ \tau_2' \leq \tau_1'$ is not derivable or $S \vdash \ \tau_1'' \leq \tau_2''$ is not derivable. We consider each possibility in turn.

If $S \vdash \ \tau_2' \leq \tau_1'$ is not derivable, then the inductive hypothesis implies that there exist E', τ', v', and e' such that $E'[\tau_1'] : \tau'$, $v' : \tau_2'$, $E'[v'] \mapsto^* e'$, and $\mathtt{stuck}(e')$. Let $v = (\mathtt{fun}\ f(x{:}\tau_1'){:}\tau_1'' = ((\mathtt{fun}\ g(y{:}\tau'){:}\tau_1'' = g(y))(E'[x])))$ (for some f, x not free in E'), $E = [\ ]v'$, $\tau = \tau_2''$, $e = (\mathtt{fun}\ g(y{:}\tau') : \tau_1'' = g(y))(e')$, and $\Gamma = \{f{:}\tau_1' \to \tau_1'', x{:}\tau_1'\}$. Also, $E'[\tau_1'] : \tau'$ implies by inversion of T-CTXT that $\{x{:}\tau_1'\} \vdash E'[x]{:}\tau'$, which in turn implies by Lemma 16 that $\Gamma \vdash E'[x]{:}\tau'$. Hence, we can use rules T-FUN and T-APP to derive $v{:}\tau_1' \to \tau_1''$, as required. Also, because $v'{:}\tau_2'$, we have $[\tau_2' \to \tau_2'']v' : \tau_2''$ (by rules T-CTXT, T-VAR, and T-APP), i.e., $E[\tau_2] : \tau$. In addition, $E[v] \mapsto (\mathtt{fun}\ g(y{:}\tau'){:}\tau_1'' = g(y))(E'[v'])$. Given that $E'[v'] \mapsto^* e'$, we have $E[v] \mapsto^* (\mathtt{fun}\ g(y{:}\tau'){:}\tau_1'' = g(y))(e')$, i.e., $E[v] \mapsto^* e$. Because $\mathtt{stuck}(e')$, we also have $\mathtt{stuck}(e)$.

On the other hand, if $S \vdash \ \tau_1'' \leq \tau_2''$ is not derivable, the inductive hypothesis implies that there exist E', τ', v', and e' such that $E'[\tau_2'']{:}\tau'$, $v'{:}\tau_1''$, $E'[v'] \mapsto^* e'$, and $\mathtt{stuck}(e')$. Observe that, by Lemma 8 and the assumption that $\mathrm{val}(\tau_2') \neq \emptyset$, there exists a value v_2 such that $v_2{:}\tau_2'$. Let $v = (\mathtt{fun}\ f(x{:}\tau_1') : \tau_1'' = v')$, $E = E'[[\ ]v_2]$, $\tau = \tau'$, and $e = e'$. Because $v'{:}\tau_1''$, Lemma 16 and rule T-FUN imply that $v{:}\tau_1$. Also, $E'[\tau_2'']{:}\tau'$ implies by inversion of T-CTXT that $\{x'{:}\tau_2''\} \vdash E'[x']{:}\tau'$, which in turn implies that $\{x''{:}\tau_2' \to \tau_2''\} \vdash E'[x''(v_2)]{:}\tau'$ (because any use of T-VAR to type the free x' in a typing derivation of $E'[x']$ can be replaced by a use of T-APP and T-VAR, and whatever rules are used to derive $v_2{:}\tau_2'$, to produce an otherwise identical typing derivation of $E'[x''(v_2)]$). Hence, by T-CTXT, $E[\tau_2]{:}\tau$. Also, by the definitions of E and v, $E[v] = E'[v(v_2)]$, and by the operational semantics, $E'[v(v_2)] \mapsto E'[v']$. Thus, because $E'[v'] \mapsto^* e'$ and $e' = e$, we have $E[v] \mapsto^* e$, where $\mathtt{stuck}(e)$.

$$\text{Case } \frac{\mathrm{val}(\mu t_1.\overline{\tau}_1) \neq \emptyset \qquad \mu t_1.\overline{\tau}_1 \leq \mu t_2.\overline{\tau}_2 \notin S \qquad S \cup \{\mu t_1.\overline{\tau}_1 \leq \mu t_2.\overline{\tau}_2\} \vdash [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2}{S \vdash \ \mu t_1.\overline{\tau}_1 \leq \mu t_2.\overline{\tau}_2} \text{ S-REC1}$$

The failing premise in this case must be $S \cup \{\mu t_1.\overline{\tau}_1 \leq \mu t_2.\overline{\tau}_2\} \vdash [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2$, so by the inductive hypothesis, there exist E', τ', v', and e' such that $E'[[\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2]{:}\tau'$, $v'{:}[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1$, $E'[v'] \mapsto^* e'$, and $\mathtt{stuck}(e')$. Let $v = \mathtt{roll}(v')$, $E = E'[\mathtt{unroll}([\ ])]$, $\tau = \tau'$, and $e = e'$. Observe that by rule T-ROLL, $v{:}\tau_1$. Also, $E'[[\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2]{:}\tau'$ implies by inversion of T-CTXT that $\{x'{:}[\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2\} \vdash E'[x']{:}\tau'$, which in turn implies that $\{x{:}\mu t_2.\overline{\tau}_2\} \vdash E'[\mathtt{unroll}(x)]{:}\tau'$ (because any use of T-VAR to type the free x' in a typing derivation of $E'[x']$ can be replaced by a use of T-UNROLL and T-VAR to produce an otherwise identical typing derivation of $E'[\mathtt{unroll}(x)]$). Hence, by T-CTXT, $E[\tau_2]{:}\tau$. Also, by the definitions of E and v, we have $E[v] = E'[\mathtt{unroll}(\mathtt{roll}(v'))]$, so $E[v] \mapsto E'[v']$, where $E'[v'] \mapsto^* e'$ (from above). Therefore, $E[v] \mapsto^* e$, and $\mathtt{stuck}(e)$ because $e' = e$ and $\mathtt{stuck}(e')$.

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 `fst` or `snd` expression to eliminate the pair in E. The S-Sum case constructs v as an `inl` or `inr` expression and uses a `case` expression to eliminate the injection in E. □

Having proved a stronger version of completeness in Lemma 18, the weaker version follows as a corollary.

Corollary 19. *Completeness.*

$\forall \tau_1, \tau_2$: *If there do not exist* E, τ, e, *and* e' *such that* $E[\tau_2]{:}\tau$, $e{:}\tau_1$, $E[e] \mapsto^* e'$, *and* $\mathtt{stuck}(e')$, *then* $\tau_1 \leq \tau_2$.

Proof. By Lemma 18, if $\tau_1 \leq \tau_2$ is not derivable then there exist E, τ, e, and e' such that $E[\tau_2]{:}\tau$, $e{:}\tau_1$, $E[e] \mapsto^* e'$, and $\mathtt{stuck}(e')$. The corollary is the contrapositive of this result. □

5.7 Subtyping Soundness

With completeness proved, we move on to proving the soundness of the subtyping relation using type-safety lemmas. Lemmas 20–22 are used to prove Preservation (Lemma 23), while Lemma 24 is used to prove Progress (Lemma 25).

Lemma 20. *Variable Substitution.*

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

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

Lemma 21. *β-Preservation.*

$$\forall e, \tau, e' : ((e{:}\tau \wedge e \mapsto_\beta e') \Rightarrow e'{:}\tau)$$

Proof. By case analysis of $e \mapsto_\beta e'$. We show the proofs of the β-SUCC, β-APP, and β-UNROLL cases. The proofs of the β-SQRT cases are similar to that of β-SUCC; the proofs of the β-LEFT and β-RIGHT cases are similar to that of β-APP; and the proofs of the β-FST and β-SND cases are similar to that of β-UNROLL.

$$\text{Case } \frac{\mathtt{n'} = \mathtt{successor\ of\ n}}{\mathtt{succ(n)} \mapsto_\beta \mathtt{n'}} \ \beta\text{-SUCC}$$

Because $\mathtt{succ}(n){:}\tau$, Lemma 15 ensures that $\mathtt{nat} \leq \tau$, while rule T-NAT ensures that $\mathtt{n'}{:}\mathtt{nat}$. Hence, $\mathtt{n'}{:}\tau$ by rule T-SUBSUME.

$$\text{Case } \frac{}{\begin{array}{c}(\mathtt{fun}\ f(x : \tau_1) : \tau_2 = e)(v) \mapsto_\beta \\ [(\mathtt{fun}\ f(x : \tau_1) : \tau_2 = e)/f][v/x]e\end{array}} \ \beta\text{-APP}$$

Let $F = (\mathtt{fun}\ f(x : \tau_1) : \tau_2 = e)$. By Lemma 15 and the assumption that $F(v){:}\tau$, we have $F{:}\tau_1' \to \tau_2'$, $v{:}\tau_1'$, and $\tau_2' \leq \tau$. By Lemma 15 again and the assumption that $F{:}\tau_1' \to \tau_2'$, we also have $\{f{:}\tau_1 \to \tau_2, x{:}\tau_1\} \vdash e{:}\tau_2$ and $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$. Because $\{f{:}\tau_1 \to \tau_2, x{:}\tau_1\} \vdash e{:}\tau_2$, rule T-FUN implies that $F{:}\tau_1 \to \tau_2$. Given that $F{:}\tau_1 \to \tau_2$ and $v{:}\tau_1'$, Lemma 8 implies that $\mathrm{val}(\tau_1 \to \tau_2) \neq \emptyset$ and $\mathrm{val}(\tau_1') \neq \emptyset$, so we can use Lemma 3 on the fact that $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ to obtain $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$. Then, because $v{:}\tau_1'$, T-SUBSUME implies $v{:}\tau_1$, so by Lemma 16, $\{f{:}\tau_1 \to \tau_2\} \vdash v{:}\tau_1$. We now have gathered all the results needed to start applying Lemma 20 (Variable Substitution). Because $\{f{:}\tau_1 \to \tau_2, x{:}\tau_1\} \vdash e{:}\tau_2$ and $\{f{:}\tau_1 \to \tau_2\} \vdash v{:}\tau_1$, Lemma 20 ensures that $\{f{:}\tau_1 \to \tau_2\} \vdash [v/x]e{:}\tau_2$. Then because $F{:}\tau_1 \to \tau_2$, Lemma 20 ensures that$[F/f][v/x]e{:}\tau_2$. Finally, with $[F/f][v/x]e{:}\tau_2$ and $\tau_2 \leq \tau_2' \leq \tau$, we have $[F/f][v/x]e{:}\tau$ by T-SUBSUME.

$$\text{Case } \frac{}{\mathtt{unroll(roll}(v)) \mapsto_\beta v} \ \beta\text{-UNROLL}$$

By Lemma 15 and the assumption that $\mathtt{unroll}(\mathtt{roll}(v)){:}\tau$, we have $\mathtt{roll}(v){:}\mu t.\overline{\tau}$ and $[\mu t.\overline{\tau}/t]\overline{\tau} \leq \tau$. Then by Lemma 15 again and the result that $\mathtt{roll}(v){:}\mu t.\overline{\tau}$, we find $v{:}[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1$ and $\mu t_1.\overline{\tau}_1 \leq \mu t.\overline{\tau}$. Because $v{:}[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1$, we have $\mathrm{val}([\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1) \neq \emptyset$ by Lemma 8, so by rule I-REC$_2$ (and Lemma 6), $\mathrm{val}(\mu t_1.\overline{\tau}_1) \neq \emptyset$. Given that $\mathrm{val}(\mu t_1.\overline{\tau}_1) \neq \emptyset$ and $\mu t_1.\overline{\tau}_1 \leq \mu t.\overline{\tau}$,

Lemma 5 implies that $[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t.\overline{\tau}/t]\overline{\tau}$. Hence, we have $v{:}[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1$ and $[\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 \leq [\mu t.\overline{\tau}/t]\overline{\tau} \leq \tau$, so $v{:}\tau$ by rule T-SUBSUME. □

Lemma 22. *Well-typed, Filled Contexts.*

$$\forall \Gamma, E, e, \tau : (\Gamma \vdash E[e]{:}\tau \Rightarrow \exists \tau' : (\Gamma \vdash e{:}\tau' \wedge \Gamma \vdash E[\tau']{:}\tau))$$

Proof. By induction on the structure of E. If $E = [\,]$, then the result is immediate with $\tau' = \tau$, because $\Gamma \vdash e{:}\tau$ by assumption and $\Gamma \vdash [\tau]{:}\tau$ by the definition of well-typed contexts and rule T-VAR. If $E = \texttt{succ}(E')$ then we can apply Lemma 15 to the assumption that $\Gamma \vdash \texttt{succ}(E'[e]){:}\tau$ to find that $\Gamma \vdash E'[e]{:}\texttt{nat}$ and $\texttt{nat} \leq \tau$. By the inductive hypothesis then, there exists a τ' such that $\Gamma \vdash e{:}\tau'$ and $\Gamma \vdash E'[\tau']{:}\texttt{nat}$, so by the definition of well-typed contexts, $\Gamma \cup \{x{:}\tau'\} \vdash E'[x]{:}\texttt{nat}$. Then by rule T-SUCC, $\Gamma \cup \{x{:}\tau'\} \vdash \texttt{succ}(E'[x]){:}\texttt{nat}$, implying by T-SUBSUME and $\texttt{nat} \leq \tau$ that $\Gamma \cup \{x{:}\tau'\} \vdash \texttt{succ}(E'[x]){:}\tau$. Hence, by rule T-CTXT we have $\Gamma \vdash E[\tau']{:}\tau$, which completes this proof case. The proofs of the other cases are all similar to this proof of the $E = \texttt{succ}(E')$ case. □

Lemma 23. *Preservation.* $\forall e, \tau, e' : ((e{:}\tau \wedge e \mapsto e') \Rightarrow e'{:}\tau)$

Proof. The only rule deriving $e \mapsto e'$ is O-STEP, so it must be the case that $e = E[e_1]$, $e' = E[e_2]$, and $e_1 \mapsto_\beta e_2$ (for some E, e_1, and e_2). Because $e{:}\tau$, we have $E[e_1]{:}\tau$, so by Lemma 22 there exists a τ' such that $e_1{:}\tau'$ and $E[\tau']{:}\tau$. Combining $e_1{:}\tau'$ with $e_1 \mapsto_\beta e_2$, Lemma 21 ensures that $e_2{:}\tau'$. Finally, because $E[\tau']{:}\tau$, 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. □

Lemma 24. *Decomposition.*

$$\forall e, \tau : \left(e : \tau \Rightarrow \left(\begin{array}{c} \exists v : (e = v) \\ \vee\ \exists\, E, e_1, e_2 : (e = E[e_1] \wedge e_1 \mapsto_\beta e_2) \end{array} \right) \right)$$

Proof. By induction on the derivation of $e{:}\tau$. The proof is a standard progress proof for expressions in evaluation contexts. □

Lemma 25. *Progress.*

$$\forall e, \tau : (e{:}\tau \Rightarrow (\exists v : (e = v) \ \vee\ \exists e' : (e \mapsto e')))$$

Proof. By assumption, $e{:}\tau$, so Lemma 24 implies that either $e = v$ or $e = E[e_1]$ such that $e_1 \mapsto_\beta e_2$. In the case of $e = E[e_1]$ such that $e_1 \mapsto_\beta e_2$, rule O-STEP ensures that $e \mapsto E[e_2]$. □

With Preservation and Progress, we have type safety.

Lemma 26. *Type Safety.*

$$\forall e, \tau, e' : \left(\left(\ e{:}\tau \wedge e \mapsto^* e'\ \right) \Rightarrow \left(\ e'{:}\tau \wedge \neg \texttt{stuck}(e')\ \right) \right)$$

Proof. By induction on the derivation of $e \mapsto^* e'$, using Progress and Preservation (Lemmas 25 and 23) in the usual way. □

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

Lemma 27. *Soundness.*

$\forall \tau_1, \tau_2$: *If* $\tau_1 \leq \tau_2$ *then there do not exist* E, τ, e, *and* e' *such that* $E[\tau_2] : \tau$, $e{:}\tau_1$, $E[e] \mapsto^* e'$, *and* $\texttt{stuck}(e')$.

Proof. Assume for the sake of obtaining a contradiction that $\tau_1 \leq \tau_2$ and there exist E, τ, e, and e' such that $E[\tau_2]{:}\tau$, $e{:}\tau_1$, $E[e] \mapsto^* e'$, and $\texttt{stuck}(e')$. 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 $\{x{:}\tau_2\} \vdash E[x]{:}\tau$, which combines with $e{:}\tau_2$ and Lemma 20 to imply that $E[e]{:}\tau$. Given that $E[e]{:}\tau$ and $E[e] \mapsto^* e'$, Lemma 26 ensures that $\neg\texttt{stuck}(e')$, which contradicts the assumption that $\texttt{stuck}(e')$. Our original assumption was therefore false, so the lemma holds. □

5.8 Subtyping Preciseness

Finally, we combine the completeness and soundness results to find that the subtyping relation defined in Figure 8 is precise with respect to type safety.

Theorem 28. *Preciseness. The subtyping relation is precise with respect to type safety. Formally, for all* τ_1 *and* τ_2*:*

$$\tau_1 \leq \tau_2 \iff \left(\begin{array}{l} \neg \exists\, E, \tau, e, e' : \\ E[\tau_2] : \tau \wedge e{:}\tau_1 \wedge E[e] \mapsto^* e' \wedge \texttt{stuck}(e') \end{array} \right)$$

Proof. Immediate by Lemma 27 and Corollary 19. □

6. Summary

Although the Amber rules are commonly used to define iso-recursive subtyping, they are incomplete with respect to type safety. Iso-recursive types that arise naturally in object-oriented programming languages and that would be safe to consider as subtypes, cannot be derived as subtypes with the Amber rules.

By incorporating unrolling into the subtyping rules for iso-recursive types, it's possible to define a subtyping relation that's precise with respect to type safety. Proving this property requires a definition of preciseness, which can be created by considering evaluation contexts. Evaluation contexts enable capturing the intuition that $\tau_1 \leq \tau_2$ ought to be derivable exactly when any τ_2-type expression—in any context of a well-typed program—can be replaced by any τ_1-type expression without introducing dynamically stuck states. This definition of preciseness, along with the proof layout and techniques presented in Section 5, may be useful for proving that other languages' subtyping relations are precise with respect to type safety.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15 (4):575–631, 1993.

[2] M. Backes, C. Hriţcu, and M. Maffei. Union and intersection types for secure protocol implementations. In *Proceedings of Theory of Security and Applications (TOSCA)*, 2011.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):8, 2011.

[4] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 20(1):24, 1998.

[5] L. Cardelli. Amber. *Combinators and functional programming languages*, pages 21–47, 1986.

[6] D. Colazzo and G. Ghelli. Subtyping recursion and parametric polymorphism in kernel fun. *Information and Computation*, 198(2):71–147, 2005.

[7] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 125–135, 1989.

[8] V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.

[9] N. Gauthier and F. Pottier. Numbering matters: first-order canonical forms for second-order recursive types. *ACM SIGPLAN Notices*, 39 (9):150–161, 2004.

[10] R. Harper. *Practical Foundations for Programming Languages*. May 2012. URL `http://www.cs.cmu.edu/~rwh/plbook/`. Version 1.32 of 05.15.2012, Working Draft.

[11] H. Hosoya, B. C. Pierce, and D. N. Turner. Datatypes and subtyping. Manuscript, 1998.

[12] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. *Math. Structures in Comp. Sci.*, 5(01):113–125, 1995.

[13] C. League and Z. Shao. Formal semantics of the FLINT intermediate language. Technical Report Yale-CS-TR-1171, Yale University, 1998.

[14] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16: 1811–1841, 1994.

[15] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, 1991.

[16] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] C. Pierik and F. S. D. Boer. On behavioral subtyping and completeness. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs*, 2005.

[18] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60–61:17–139, 2004.

[19] T. Sekiguchi and A. Yonezawa. A complete type inference system for subtyped recursive types. In *Proceedings of Theoretical Aspects of Computer Software (TACS)*, pages 667–686, 1994.

[20] A. J. H. Simons. Adding axioms to Cardelli-Wegner subtyping. Technical Report CS-94-6, University of Sheffield, 1994.

[21] A. J. H. Simons. The theory of classification, part 4: Object types and subtyping. *Journal of Object Technology*, 1(5):27–35, 2002.

[22] C. A. Stone and A. P. Schoonmaker. Equational theories with recursive types. URL `http://www.cs.hmc.edu/~stone/papers/recfull.pdf`. Under consideration for publication.

[23] R. Tate, A. Leung, and S. Lerner. Taming wildcards in Java's type system. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[24] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. Typed compilation of recursive datatypes. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, 2003.

[25] J. Vouillon. Subtyping union types. In *Proceedings of the 18th International Workshop on Computer Science Logic*, 2004.

A. Subtyping Algorithm Implementation

```
1  (* Constructors for language types.  Type variables are represented as integers,
2    which are assumed to be named 0, 1, etc. *)
3  datatype typ = Nat | Real | Prod of typ * typ | Sum of typ * typ | Rec of int * typ
4               | Fun of typ * typ | Var of int;
5
6  (* An extended type is a type that, if needed, is paired with a bool indicating whether
7    the type is uninhabited by values *)
8  datatype etyp = ENat | EReal | EProd of etyp * etyp * bool | ESum of etyp * etyp * bool
9                | ERec of int * etyp * bool | EFun of etyp * etyp | EVar of int * bool;
10
11 (* Returns the number of variables defined in a type.
12    Type ids are unique, so this function returns the number of distinct ids. *)
13 fun numVars (Sum(t1, t2)) = numVars(t1) + numVars(t2)
14   | numVars (Prod(t1, t2)) = numVars(t1) + numVars(t2)
15   | numVars (Fun(t1, t2)) = numVars(t1) + numVars(t2)
16   | numVars (Rec(n, t1)) = 1 + numVars(t1)
17   | numVars _ = 0;
18
19 (* Given an extended type, this function returns a bool indicating whether
20    the type is uninhabited *)
21 fun uninhabited (EProd(_,_,b)) = b
22   | uninhabited (ESum(_,_,b)) = b
23   | uninhabited (ERec(_,_,b)) = b
24   | uninhabited (EVar(_,b)) = b
25   | uninhabited _ = false;
26
27 (* This function allocates an extended type and initializes any
28    uninhabitation flags to false *)
29 fun makeETyp Nat = ENat
30   | makeETyp Real = EReal
31   | makeETyp (Var(n)) = EVar(n, false)
32   | makeETyp (Fun(t1, t2)) = EFun(makeETyp t1, makeETyp t2)
33   | makeETyp (Rec(n, t1)) = ERec(n, makeETyp t1, false)
34   | makeETyp (Sum(t1, t2)) = ESum(makeETyp t1, makeETyp t2, false)
35   | makeETyp (Prod(t1, t2)) = EProd(makeETyp t1, makeETyp t2, false);
36
37 (* This function takes an extended type et, an array
38    of bools U, and an array of extended types ut.et has just been initialized,
39    so its uninhabitation flags are set to false.
40    U[i] is true iff the type variable i is already known to be uninhabited.
41    ut is an unroll table; ut[i] is the unrolled extended type to which type variable i refers.
42    This function returns extended type et, but with its uninhabitation flags set correctly.
43    Recursive types are represented as just their type variables, so et should never be an ERec. *)
44 fun setUninhabited ENat _ _ = ENat
45   | setUninhabited EReal _ _ = EReal
46   | setUninhabited (EFun(t1, t2)) U ut =
47       EFun(setUninhabited t1 U ut, setUninhabited t2 U ut)
48   | setUninhabited (ESum(t1, t2, _)) U ut =                    (* rule U-Sum *)
49       let val e1 = setUninhabited t1 U ut
50           val e2 = setUninhabited t2 U ut
51       in ESum(e1, e2, uninhabited e1 andalso uninhabited e2)
52       end
53   | setUninhabited (EProd(t1, t2, _)) U ut =                   (* rules U-Prod1 and U-Prod2 *)
54       let val e1 = setUninhabited t1 U ut
55           val e2 = setUninhabited t2 U ut
56       in EProd(e1, e2, uninhabited e1 orelse uninhabited e2)
57       end
58   | setUninhabited (EVar(n, _)) U ut =                         (* rules U-Rec1 and U-Rec2 *)
59       if U[n] then (EVar(n, true)) else
60       (* find whether the unrolled version of et is uninhabited,
61          while assuming that et itself is uninhabited *)
62       let val e1 = setUninhabited (ut[n]) (U[n]:=true; U) ut
63       in  EVar(n, uninhabited e1)
```

```
64        end
65    | setUninhabited _ _ _ = raise error;
66
67  (* This function takes an extended type et and returns the extended type obtained by replacing all
68     recursive types muX.t in et with just the type variable X *)
69  fun compress (ERec(n, _, b)) = EVar(n, b)
70    | compress (EProd(t1, t2, b)) = EProd(compress t1, compress t2, b)
71    | compress (ESum(t1, t2, b)) = ESum(compress t1, compress t2, b)
72    | compress (EFun(t1, t2)) = EFun(compress t1, compress t2)
73    | compress (EVar(_, _)) = raise freeTypeVar
74    | compress t = t;
75
76  (* This function initializes an unroll table.  It takes an extended type and an array
77     of extended types ea and updates the entries in ea such that ea[i] is the unrolled
78     extended type to which type variable i refers *)
79  fun initUnrollTable (ERec(tvar, t1, _)) a = (a[tvar]:=compress t1; initUnrollTable t1 a)
80    | initUnrollTable (EFun(t1, t2)) a = (initUnrollTable t1 a; initUnrollTable t2 a)
81    | initUnrollTable (EProd(t1, t2, _)) a = (initUnrollTable t1 a; initUnrollTable t2 a)
82    | initUnrollTable (ESum(t1, t2, _)) a = (initUnrollTable t1 a; initUnrollTable t2 a)
83    | initUnrollTable _ _ = ();
84
85  (* This is the subtyping function; it takes types t1 and t2 and returns true iff
86     t1 is a subtype of t2 *)
87  fun sub t1 t2 =
88    let
89      (* calculate numvars *)
90      val m = numVars t1;
91      val n = numVars t2;
92      (* allocate arrays for mapping recursive-type ids to uninhabitation assumptions *)
93      val ua1 = Array.array(m, false);
94      val ua2 = Array.array(n, false);
95      (* Allocate an array for storing subtyping assumptions.
96         When S[m][n] = (b1,b2), b1 indicates whether recursive type m in t1 is a subtype
97         of recursive type  n in t2, and b2 indicates whether recursive type n in t2 is a subtype
98         of recursive type m in t1. *)
99      val S = Array2.array(m, n, (false, false));
100     (* allocate an unroll table for t1 and t2 *)
101     val ut1 = Array.array(m, ENat);
102     val ut2 = Array.array(n, ENat);
103     (* convert t1 and t2 into their extended versions, with uninhabitation flags properly set *)
104     val et1 = makeETyp t1
105     val et2 = makeETyp t2
106     val _ = initUnrollTable et1 ut1;
107     val _ = initUnrollTable et2 ut2;
108     val et1 = setUninhabited (compress et1) ua1 ut1;
109     val et2 = setUninhabited (compress et2) ua2 ut2;
110
111     (* Helper subtyping function; operates on extended (rather than basic) types.
112        The swapped flag indicates whether we're subtyping in a contravariant position,
113        in which case the m in S[m][n] refers to a type in t2, and the n to a type in t1.
114        Recursive types are represented as just their type variables, so neither et1
115        nor et2 should be an ERec. *)
116     fun subh et1 et2 swapped =
117         uninhabited et1                                  (* S-Bottom *)
118       orelse
119         (case et2 of
120           EFun(et2', et2'') => uninhabited et2'          (* S-BottomFun *)
121         | _ => false)
122       orelse
123         case (et1, et2) of
124           (ENat, EReal) => true                          (* S-Base *)
125         | (ENat, ENat) => true                           (* S-Nat *)
126         | (EReal, EReal) => true                         (* S-Real *)
127         | (EFun(t1, t2), EFun(t1', t2')) =>              (* S-Fun *)
128             subh t1' t1 (not swapped) andalso subh t2 t2' swapped
```

```
129           | (ESum(t1, t2, _), ESum(t1', t2', _)) =>     (* S-Sum *)
130               subh t1 t1' swapped andalso subh t2 t2' swapped
131           | (EProd(t1, t2, _), EProd(t1', t2', _)) =>  (* S-Prod *)
132               subh t1 t1' swapped andalso subh t2 t2' swapped
133           | (EVar(m, _), EVar(n, _)) =>                (* S-Rec1 and S-Rec2 *)
134               if swapped then
135                 #2(S[n][m]) orelse subh (S[n][m]:=(#1(S[n][m]), true); ut2[m]) (ut1[n]) swapped
136               else
137                 #1(S[m][n]) orelse subh (S[m][n]:=(true, #2(S[m][n])); ut1[m]) (ut2[n]) swapped
138           | (ERec(_,_,_), _) => raise error
139           | (_, ERec(_,_,_)) => raise error
140           | _ => false
141       in
142         subh et1 et2 false
143       end;
```