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

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

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

This paper develops 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. A new proof technique, induction on failing derivations, is provided that may be useful in other domains as well.

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.

New, algorithmic rules are defined for subtyping iso-recursive types, and the rules are proved sound and complete with respect to type safety. The fully implemented subtyping algorithm is optimized to run in $O(mn)$ time, where m is the number of μ-terms in the types being considered and n is the size of the types being considered.

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

General Terms: Languages, Algorithms

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

1. INTRODUCTION

When defining a subtyping relation for a type-safe language, one takes into account both the soundness and the completeness of the subtyping relation with respect to type safety. Soundness alone can be satisfied by making the subtyping relation the least reflexive and transitive relation over types (i.e., 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

This work was supported by the National Science Foundation, under grant CNS-0742736.
Corresponding author's email address: ligatti@cse.usf.edu
Jeremy Blackburn's email address: jeremyb@tid.es
Michael Nachtigal's email address is unknown.

DOI: http://dx.doi.org/10.1145/0000000.0000000

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 aren't defined for τ_1-type terms, potentially leading to dynamically "stuck" states.

This paper develops 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 combine with product and sum types to form algebraic data types, which are fundamental for typing aggregate data structures. A standard example of a recursive type would be a natural-number-list type $L \equiv \mu t.(\mathtt{unit}+(\mathtt{nat} \times t))$. 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 $\mathtt{roll}_L(\mathtt{inl}_{\mathtt{unit}+(\mathtt{nat}\times L)}())$; the inl value has type $\mathtt{unit}+(\mathtt{nat}\times L)$, so rolling it produces a value of type L. Although type checkers in languages with equi-recursive (also called strongly recursive) types automatically roll and unroll terms as needed, so programmers don't have to, practical programming languages that support iso-recursive types, such as ML and Haskell, ease the burden of rolling and unrolling by merging this syntax with other syntax. For example, a programmer might define an iso-recursive type for natural-number lists with

```
rectype t = nil of unit + cons of nat * t
```

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

1.1. Related Work

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

This paper approaches subtyping from a type-safety perspective, investigating the greatest subtyping relation possible without violating type safety; however, other notions of when one type can or should be a subtype of another may be preferred in other contexts. For example, subtyping may be based on particular behaviors of objects in object-oriented programming languages (OOPLs) [Liskov and Wing 1994; Pierik and Boer 2005]. Another common approach considers 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 $[\![\tau_1]\!] \subseteq [\![\tau_2]\!] \Rightarrow \tau_1 \leq \tau_2$ [Barendregt et al. 1983; van Bakel et al. 2000; Vouillon 2004; Vouillon 2006; Hosoya et al. 2005; Frisch et al. 2008; Dezani-Ciancaglini and Ghilezan 2014]. Using these definitions, it has been shown that the standard subtyping rules for function, union, and intersection types are sound and complete (under some assumptions but overall for a broad class of languages) [Barendregt et al. 1983;

van Bakel et al. 2000; Vouillon 2004]. In contrast with these other approaches to subtyping, soundness and completeness in this paper are structural properties that, like normal type safety, specify relationships between languages' static and (here, SOS-style [Plotkin 2004]) dynamic semantics.

The research on subtyping recursive types seems to have focused more on equi-recursive than iso-recursive systems. For example, Amadio and Cardelli presented rules and an algorithm for subtyping equi-recursive types [Amadio and Cardelli 1993]. The rules and algorithm are proved sound and complete with respect to type trees that result from "infinitely unrolling" equi-recursive types (i.e., the rules and algorithm determine $\tau_1 \leq \tau_2$ precisely when the type obtained by infinitely unrolling 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., [Kozen et al. 1995; Brandt and Henglein 1998; Gapeyev et al. 2002; Gauthier and Pottier 2004; Stone and Schoonmaker 2005; Colazzo and Ghelli 2005]).

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

$$\frac{S \cup \{t \leq t'\} \vdash \tau \leq \tau'}{S \vdash \mu t.\tau \leq \mu t'.\tau'}\ \textsc{Amber1} \qquad \frac{}{S \cup \{t \leq t'\} \vdash t \leq t'}\ \textsc{Amber2}$$

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" [Cardelli 1986]. Note that the Amber rules assume type variables are uniquely named, through alpha-conversion if necessary.

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

1.2. Overview and List of Contributions

Interpreting types as sets of terms, subtyping could be considered as natural in type theory as subsetting is in set theory. Besides this theoretical interest in subtyping, many practical programming languages allow terms of one type to stand in for terms of another type; this paper provides a basic framework for deciding when to make such allowances.

Section 2 formalizes what it means for a subtyping relation to be sound, complete, and precise with respect to type safety. Intuitively, a precise (i.e., sound and complete) subtyping relation specifies that 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 proves that the standard subtyping system for a simply typed lambda calculus is precise with respect to type safety. The proof's layout and techniques are rather general, so we expect them to be helpful for proving the preciseness of other inductively defined subtyping relations. Moreover, the proof of completeness uses a new (as far as we're aware) technique, induction on *failing* derivations [Ligatti 2016a], which may be of independent interest and useful in other domains.

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

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

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

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

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

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

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

Section 6 contains further discussion of the paper's definitions and results.

2. BASIC DEFINITIONS

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

Intuitively, we wish for a language's subtyping relation to define $\tau_1{\leq}\tau_2$ precisely when such a definition could not compromise type safety. By the principle of subsumption, which states that a term of type 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 a "fresh" variable, not appearing 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.

Definition 1 stipulates that precise subtyping relations allow $\tau_1 \leq \tau_2$ exactly when it's impossible to reach a "stuck" state by replacing an evaluable τ_2-type expression in a well-typed program with a τ_1-type expression and evaluating the result. That is, $\tau_1 \leq \tau_2$ means that replacing τ_2-type expressions with τ_1-type expressions can't break type safety.

3. AN INTRODUCTORY PROOF OF PRECISENESS

To more concretely understand and apply these definitions, let's consider a simple language λ, a simply typed lambda calculus with base types `nat` (natural numbers, e.g. 3) and `real` (real numbers, e.g. 3.0), the idea being that $\texttt{nat} \leq \texttt{real}$. Figure 1 presents the syntax and static and dynamic semantics. All the notation in Figure 1 is intended to have the usual meanings, with the usual assumptions being made. For example, variables are consistently renamed, through alpha-conversion, whenever necessary to avoid reintroducing variables into contexts, and empty contexts are normally omitted from judgment forms (e.g., $e{:}\tau$ means $\emptyset \vdash e : \tau$).

The expressions in λ are natural and real numbers, successor and negation operations, anonymous functions, applications, and variables. The negation operation is defined on natural and real numbers, while the successor operation is defined on natural, but not real, numbers.

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

3.1. Proof that λ's Subtyping Relation is Sound

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

LEMMA 2. *Weakening.*

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

PROOF. By induction on the derivation of $\Gamma \vdash e : \tau$. □

LEMMA 3. *Universal Value-Inhabitation.*

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

PROOF. By induction on the structure of τ. □

LEMMA 4. *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 5. *Type Safety.*

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

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

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

$$
\begin{array}{rrcl}
\text{Types} & \tau & ::= & \texttt{nat} \mid \texttt{real} \mid \tau_1{\rightarrow}\tau_2 \\
\text{Expressions} & e & ::= & \texttt{n} \mid \texttt{r} \mid \texttt{succ}(e) \mid \texttt{neg}(e) \mid \lambda x{:}\tau.e \mid e_1(e_2) \mid x \\
\text{Evaluation contexts} & E & ::= & [\,] \mid \texttt{succ}(E) \mid \texttt{neg}(E) \mid E\ (e) \mid v\ (E) \\
\text{Values} & v & ::= & \texttt{n} \mid \texttt{r} \mid \lambda x{:}\tau.e
\end{array}
$$

$\boxed{e \mapsto e'}$

$$\frac{e \mapsto_\beta e'}{E[e] \mapsto E[e']}$$

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

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

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

$$\frac{\texttt{n}' = \texttt{n} + 1}{\texttt{succ}(\texttt{n}) \mapsto_\beta \texttt{n}'} \qquad \frac{\texttt{r}' = -\texttt{r}}{\texttt{neg}(\texttt{r}) \mapsto_\beta \texttt{r}'} \qquad \frac{\texttt{r} = -\texttt{n}}{\texttt{neg}(\texttt{n}) \mapsto_\beta \texttt{r}} \qquad \frac{}{(\lambda x{:}\tau.e)(v) \mapsto_\beta [v/x]e}$$

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

$$\frac{}{e \mapsto^* e} \qquad \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}$$

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

$$\frac{}{\Gamma \vdash \texttt{n} : \texttt{nat}}\ \textsc{T-Nat} \qquad \frac{}{\Gamma \vdash \texttt{r} : \texttt{real}}\ \textsc{T-Real} \qquad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{succ}(e) : \texttt{nat}}\ \textsc{T-Succ}$$

$$\frac{\Gamma \vdash e : \texttt{real}}{\Gamma \vdash \texttt{neg}(e) : \texttt{real}}\ \textsc{T-Neg} \qquad \frac{\Gamma \cup \{x{:}\tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : \tau{\rightarrow}\tau'}\ \textsc{T-Lam}$$

$$\frac{\Gamma \vdash e_1 : \tau{\rightarrow}\tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1(e_2) : \tau'}\ \textsc{T-App} \qquad \frac{}{\Gamma \cup \{x{:}\tau\} \vdash x : \tau}\ \textsc{T-Var}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau}\ \textsc{T-Subsume}$$

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

$$\frac{\Gamma \cup \{x{:}\tau'\} \vdash E[x] : \tau \qquad (x \notin E)}{\Gamma \vdash E[\tau'] : \tau}\ \textsc{T-Ctxt}$$

$\boxed{\tau{\leq}\tau'}$

$$\frac{}{\texttt{nat}{\leq}\texttt{real}} \qquad \frac{}{\texttt{nat}{\leq}\texttt{nat}} \qquad \frac{}{\texttt{real}{\leq}\texttt{real}} \qquad \frac{\tau_1'{\leq}\tau_1 \qquad \tau_2{\leq}\tau_2'}{\tau_1{\rightarrow}\tau_2{\leq}\tau_1'{\rightarrow}\tau_2'}$$

Fig. 1. Definition of λ.

LEMMA 6. *Soundness.*

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

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 4 to imply that $E[e]{:}\tau$. Given that $E[e]{:}\tau$ and $E[e] \mapsto^* e'$, Lemma 5 ensures that $\neg\texttt{stuck}(e')$, which contradicts the assumption that $\texttt{stuck}(e')$. Our original assumption was therefore false, so the lemma holds. □

3.2. Induction on Failing Derivations

This paper's completeness proofs use a technique that we call induction on failing derivations [Ligatti 2016a].

Standard induction on derivations (as was used in the proofs of Lemmas 2, 4, and 5) is a form of tree induction, where the trees are proofs (derivations). Induction on failing derivations is also a form of tree induction, but where the trees are refutations (failing derivations). Whereas standard induction on derivations establishes that some property holds on all derivable (provable) judgments, induction on failing derivations establishes that some property holds on all underivable (refutable) judgments.

3.2.1. Navigable Systems. Let's call a deductive system *navigable* when its inference rules satisfy two properties:

(1) Every rule's conclusion completely determines its premises. Formally, there exists a total, computable function f—which we call a *rule function*—from any judgment J to a disjunctive normal form (DNF) of judgments, such that

$$f(J) = (J_1^1 \wedge .. \wedge J_{n_1}^1) \vee .. \vee (J_1^m \wedge .. \wedge J_{n_m}^m) \qquad (m, n_1, .., n_m \in \mathbb{N})$$

means J is derivable iff

— $J_1^1..J_{n_1}^1$ are all derivable (using a rule $\dfrac{J_1^1 \ .. \ J_{n_1}^1}{J}$), or

— $J_1^2..J_{n_2}^2$ are all derivable (using a different rule $\dfrac{J_1^2 \ .. \ J_{n_2}^2}{J}$), or

— etc.

Empty conjunctive clauses in $f(J)$ (i.e., when $n_i{=}0$) are written as $()$ and specify that J may be concluded using a premiseless rule. An empty disjunctive clause for $f(J)$ (i.e., when $m{=}0$) is written as ε and specifies that no rule concludes J.

(2) Infinite descent into premises is impossible; attempts to derive judgments always terminate. Formally, the relation $\{(J_p, J_c) \mid J_p \in f(J_c)\}$, which relates premise judgments to conclusion judgments, is well-founded.

The set of derivable judgments in a navigable system is plainly decidable.

For example, the subtyping system for λ is navigable:

(1) On inputs $\texttt{nat}{\leq}\texttt{real}$, $\texttt{nat}{\leq}\texttt{nat}$, and $\texttt{real}{\leq}\texttt{real}$, rule function f returns $()$, a trivially satisfiable DNF; on inputs of the form $\tau_1{\to}\tau_2{\leq}\tau_1'{\to}\tau_2'$, f returns $(\tau_1'{\leq}\tau_1 \wedge \tau_2{\leq}\tau_2')$; on all other inputs (i.e., on $\texttt{real}{\leq}\texttt{nat}$ and all inputs $\tau{\leq}\tau'$ such that exactly one of τ and τ' is a function type), f returns ε, the trivially unsatisfiable DNF.

(2) The relation of premise judgments to conclusion judgments, that is,

$$\bigcup_{\tau_1,\tau_2,\tau_1',\tau_2'} \{(\tau_1'{\leq}\tau_1, \tau_1{\to}\tau_2{\leq}\tau_1'{\to}\tau_2'), (\tau_2{\leq}\tau_2', \tau_1{\to}\tau_2{\leq}\tau_1'{\to}\tau_2')\},$$

is well-founded. Attempts to derive $\tau{\leq}\tau'$ always terminate because premises decrease the sizes of types being considered.

However, the typing system for λ is nonnavigable:

(1) No rule function f exists. Even ignoring T-SUBSUME, the unbounded nondeterminism in T-APP's premises would require f, on any input of the form $\Gamma{\vdash}e_1(e_2){:}\tau'$, to return the infinite DNF

$$\bigvee_{\tau} (\Gamma{\vdash}e_1{:}\tau{\to}\tau' \ \wedge \ \Gamma{\vdash}e_2{:}\tau).$$

(2) Due to rule T-SUBSUME, the relation of premises to conclusions would contain elements of the form $(\Gamma{\vdash}e{:}\tau', \Gamma{\vdash}e{:}\tau)$ and therefore not be well-founded.

3.2.2. Failing Derivations. Given a navigable system with rule function f, a *derivation* of J is a tree of judgments having root J, internal judgments J_i such that J_i's children are all the members of a conjunctive clause in $f(J_i)$, and leaves J_l such that $f(J_l)$ contains the empty conjunctive clause () (i.e., $f(J_l)$ is trivially satisfiable).

Complementarily, a *failing derivation* of J is a tree of judgments having root J, internal judgments J_i such that J_i's children contain exactly one member of each conjunctive clause in $f(J_i)$, and leaves J_l such that $f(J_l)=\varepsilon$ (i.e., $f(J_l)$ is trivially unsatisfiable).

For navigable systems, J is underivable iff there exists a failing derivation of J. All the leaves J_l of failing derivations are underivable (because $f(J_l)=\varepsilon$), so inductively all the internal J_i must be underivable as well (because every conjunctive clause in $f(J_i)$ contains an underivable judgment). Hence, the existence of a failing derivation of J implies that J is underivable. Conversely, if J is underivable then by the definition of f, $f(J)$ must either be ε (in which case the failing derivation of J is just the leaf J) or have an underivable judgment J_u in each conjunctive clause (in which case the failing derivation of J gets built inductively by making J an internal judgment having those J_u as children).

For example, the judgment

$$J_0 = (\texttt{real}{\rightarrow}\texttt{real}){\rightarrow}(\texttt{nat}{\rightarrow}\texttt{real}) \leq (\texttt{real}{\rightarrow}\texttt{nat}){\rightarrow}(\texttt{real}{\rightarrow}\texttt{real})$$

is underivable using λ's subtyping system, so there exists a failing derivation rooted at J_0. This failing derivation is a linear tree, with J_0's only child being

$$J_1 = \texttt{nat}{\rightarrow}\texttt{real}{\leq}\texttt{real}{\rightarrow}\texttt{real},$$

and J_1's only child being the leaf

$$J_2 = \texttt{real}{\leq}\texttt{nat}.$$

As required:

- Every internal judgment J_i has one child from each conjunctive clause in $f(J_i)$.
  - $f(J_0) = (\texttt{real}{\rightarrow}\texttt{nat}{\leq}\texttt{real}{\rightarrow}\texttt{real} \ \wedge\ J_1)$
  - $f(J_1) = (J_2 \ \wedge\ \texttt{real}{\leq}\texttt{real})$
- f returns ε on the leaf judgment $J_2 = \texttt{real}{\leq}\texttt{nat}$.

The underivability of J_0 can thus be traced from underivable J_2 to underivable J_1 to underivable J_0.

Although derivations and failing derivations exist in some nonnavigable systems (derivations exist when the set of derivable judgments is recursively enumerable, and failing derivations exist when the set of derivable judgments is co-recursively enumerable), this paper limits consideration of failing derivations to navigable systems.

3.2.3. Induction on Failing Derivations. Because judgments in navigable systems are underivable iff they root failing derivation trees, one may establish that some property P holds on all underivable J by induction on the failing derivation of J.

As an example, let's consider proving a property P on underivable $\tau{\leq}\tau'$ judgments in λ, by induction on the failing derivation of $\tau{\leq}\tau'$. Leaf judgments in such trees can only be of the form $\texttt{real}{\leq}\texttt{nat}$ or $\tau{\leq}\tau'$ such that exactly one of τ and τ' is a function type; the base cases of the proof must show that P holds on all such judgments. The inductive case occurs when subtyping function types—all internal judgments in failing derivations must be of the form $J_i = \tau_1{\rightarrow}\tau_2{\leq}\tau_1'{\rightarrow}\tau_2'$, such that J_i has one child, which can be either an underivable $\tau_1'{\leq}\tau_1$ or an underivable $\tau_2{\leq}\tau_2'$. Hence, the proof must show both cases, i.e., when inductively assuming that P holds on $\tau_1'{\leq}\tau_1$, that P holds on J_i, *and* when inductively assuming that P holds on $\tau_2{\leq}\tau_2'$, that P holds on J_i.

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 don't exist E, τ, and e such that $E[\tau_2]{:}\tau$, $e{:}\tau_1$, and $E[e]$ gets stuck, 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 $\tau_1{\leq}\tau_2$.

3.3. Proof that λ's Subtyping Relation is Complete

Lemma 7 uses induction on failing derivations to prove a slightly stronger version of completeness. The proof is constructive; given any τ_1 and τ_2 such that $\tau_1{\leq}\tau_2$ is not derivable, the proof shows how to (inductively) construct a well-typed program that gets stuck when its τ_2-type subexpression is replaced by a τ_1-type value.

LEMMA 7. *Strong Completeness.*

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

PROOF. By induction on the failing derivation of $\tau_1{\leq}\tau_2$. Leaf judgments occur when τ_1 is `real` and τ_2 is `nat`, or when exactly one of τ_1 and τ_2 is a function type. The lemma is first proved for these base cases.

— Case $\tau_1 = \texttt{real}$ and $\tau_2 = \texttt{nat}$:
  Define:
  — $E = \texttt{succ}([\,])$
  — $\tau = \texttt{nat}$
  — $v = 2.718$
  — $e = \texttt{succ}(2.718)$
  Then:
  — $E[\tau_2] : \tau$ (by rules T-CTXT, T-SUCC, and T-VAR)
  — $v : \tau_1$ (by T-REAL)
  — $E[v] \mapsto^* e$ (by the reflexive multistep rule)
  — $\texttt{stuck}(e)$ (by the definitions of `stuck` and e)
— Case $\tau_1 = \tau_1'{\rightarrow}\tau_1''$ and $\tau_2 \neq \tau_2'{\rightarrow}\tau_2''$:
  Because τ_2 isn't a function type, it must be `nat` or `real`. Also, by Lemma 3 there exists a v_1'' such that $v_1'' : \tau_1''$. Note that it is straightforward to prove Lemma 3 constructively, so we can construct this v_1''. Now define:
  — $E = \texttt{neg}([\,])$
  — $\tau = \texttt{real}$
  — $v = \lambda x : \tau_1'.v_1''$
  — $e = \texttt{neg}(\lambda x : \tau_1'.v_1'')$
  Then:
  — $E[\tau_2] : \tau$ (by T-CTXT, T-NEG, T-VAR, and T-SUBSUME when $\tau_2{=}\texttt{nat}$)
  — $v : \tau_1$ (by T-LAM)
  — $E[v] \mapsto^* e$ (by the reflexive multistep rule)
  — $\texttt{stuck}(e)$ (by the definitions of `stuck` and e)
— Case $\tau_1 \neq \tau_1'{\rightarrow}\tau_1''$ and $\tau_2 = \tau_2'{\rightarrow}\tau_2''$:
  Because τ_1 isn't a function type, it must be `nat` or `real`. Also, by Lemma 3 there exists a v_2' such that $v_2' : \tau_2'$. Now define:
  — $E = [\,](v_2')$
  — $\tau = \tau_2''$
  — $v = 0$
  — $e = 0(v_2')$
  Then:
  — $E[\tau_2] : \tau$ (by T-CTXT, T-APP, and T-VAR)
  — $v : \tau_1$ (by the fact that τ_1 is `nat` or `real`)
  — $E[v] \mapsto^* e$ (by the reflexive multistep rule)
  — $\texttt{stuck}(e)$ (by the definitions of `stuck` and e)

Internal judgments in a failing derivation of $\tau_1{\leq}\tau_2$ have the form $\tau_1'{\rightarrow}\tau_1''{\leq}\tau_2'{\rightarrow}\tau_2''$. There are two possible children of such internal judgments, either $\tau_2'{\leq}\tau_1'$ or $\tau_1''{\leq}\tau_2''$ (underivability of $\tau_1'{\rightarrow}\tau_1''{\leq}\tau_2'{\rightarrow}\tau_2''$ must be due to $\tau_2'{\leq}\tau_1'$ or $\tau_1''{\leq}\tau_2''$ being underivable). Let's consider each of these two subcases in turn.

— Case $\tau_1{=}\tau_1'{\rightarrow}\tau_1''$, $\tau_2{=}\tau_2'{\rightarrow}\tau_2''$, and $\tau_2'{\leq}\tau_1'$ is underivable:
By Lemma 3 there exists a v_1'' such that $v_1''{:}\tau_1''$. Also, by the inductive hypothesis (applied to $\tau_2'{\leq}\tau_1'$), there exist E', τ', v', and e' such that:
— $E'[\tau_1'] : \tau'$
— $v' : \tau_2'$
— $E'[v'] \mapsto^* e'$
— $\texttt{stuck}(e')$
Now define:
— $E = [\,](v')$
— $\tau = \tau_2''$
— $v = \lambda x{:}\tau_1'.((\lambda y{:}\tau'.v_1'')(E'[x]))$
— $e = (\lambda y{:}\tau'.v_1'')(e')$
Then:
— $E[\tau_2] : \tau$ (by T-CTXT, T-APP, T-VAR, and Lemma 2, where $\tau_2{=}\tau_2'{\rightarrow}\tau_2''$ and $v'{:}\tau_2'$)
— $v : \tau_1$ (by T-LAM, T-APP, and Lemma 2, where $\tau_1{=}\tau_1'{\rightarrow}\tau_1''$, $v_1''{:}\tau_1''$, and $E'[\tau_1'] : \tau'$)
— $E[v] \mapsto^* e$ (because $E[v] \mapsto (\lambda y{:}\tau'.v_1'')(E'[v'])$ and $E'[v'] \mapsto^* e'$)
— $\texttt{stuck}(e)$ (because $\texttt{stuck}(e')$)

— Case $\tau_1{=}\tau_1'{\rightarrow}\tau_1''$, $\tau_2{=}\tau_2'{\rightarrow}\tau_2''$, and $\tau_1''{\leq}\tau_2''$ is underivable:
By Lemma 3 there exists a v_2' such that $v_2'{:}\tau_2'$. Also, by the inductive hypothesis (applied to $\tau_1''{\leq}\tau_2''$), there exist E', τ', v', and e' such that:
— $E'[\tau_2''] : \tau'$
— $v' : \tau_1''$
— $E'[v'] \mapsto^* e'$
— $\texttt{stuck}(e')$
Now define:
— $E = E'[\,[\,](v_2')\,]$ (i.e., build E by filling the hole of E' with $[\,](v_2')$)
— $\tau = \tau'$
— $v = \lambda x{:}\tau_1'.v'$
— $e = e'$
Then:
— $E[\tau_2]{:}\tau$ (because $E'[\tau_2'']{:}\tau'$ means that $\{y{:}\tau_2''\}{\vdash}E'[y]{:}\tau'$, which implies by Lemma 2 that $\{z{:}\tau_2, y{:}\tau_2''\}{\vdash}E'[y]{:}\tau'$; then because $\{z{:}\tau_2\}{\vdash}z(v_2'){:}\tau_2''$, Lemma 4 ensures that $\{z{:}\tau_2\}{\vdash}E'[z(v_2')]{:}\tau'$, which means that $E'[[\tau_2](v_2')]{:}\tau'$)
— $v : \tau_1$ (by T-LAM and Lemma 2, where $\tau_1{=}\tau_1'{\rightarrow}\tau_1''$ and $v'{:}\tau_1''$)
— $E[v] \mapsto^* e$ (because $E[v] \mapsto E'[v']$ and $E'[v'] \mapsto^* e'$)
— $\texttt{stuck}(e)$ (because $\texttt{stuck}(e')$)

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

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

4. INCOMPLETENESS WITH THE AMBER RULES, FOR SUBTYPING ISO-RECURSIVE TYPES

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

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

4.1. A First Source of Incompleteness: Complications with Contravariance

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

$$\dfrac{\dfrac{\overline{\{t{\leq}t'\} \vdash \mathtt{nat}{\leq}\mathtt{nat}} \qquad \overline{\{t{\leq}t'\} \vdash t{\leq}t'}}{\{t{\leq}t'\} \vdash \mathtt{nat}{\to}t \leq \mathtt{nat}{\to}t'}}{\mu t.(\mathtt{nat}{\to}t) \leq \mu t'.(\mathtt{nat}{\to}t')}$$

But we can't derive other reflexive relationships, like $\mu t.(t{\to}\mathtt{nat}) \leq \mu t'.(t'{\to}\mathtt{nat})$:

$$\dfrac{\dfrac{\dfrac{\Downarrow\ \texttt{Derivation fails here}\ \Downarrow}{\{t{\leq}t'\} \vdash t'{\leq}t} \qquad \overline{\{t{\leq}t'\} \vdash \mathtt{nat}{\leq}\mathtt{nat}}}{\{t{\leq}t'\} \vdash t{\to}\mathtt{nat} \leq t'{\to}\mathtt{nat}}}{\mu t.(t{\to}\mathtt{nat}) \leq \mu t'.(t'{\to}\mathtt{nat})}$$

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

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

$$\dfrac{\{t'{\leq}t \mid t{\leq}t' \in S\} \vdash \tau_1'{\leq}\tau_1 \qquad S \vdash \tau_2{\leq}\tau_2'}{S \vdash \tau_1{\to}\tau_2 \leq \tau_1'{\to}\tau_2'}$$

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

— $f = \lambda x{:}\tau.\mathtt{succ}(\mathtt{unroll}(x)\ \ x)$
— $g = \lambda y{:}\tau'.(2.718)$

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

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

The only other approach we can think of to make the Amber rules reflexive, so we can derive that types like $\mu t.(t{\to}\mathtt{nat})$ are subtypes of themselves, is to add a rule explicitly saying so. To create such a rule, let's focus on the core problem here: when the Amber rules reach a judgment $\mu t.\tau{\leq}\mu t'.\tau'$, they attempt to derive $\tau{\leq}\tau'$ while assuming $t{\leq}t'$ but are unable to derive that $t'{\leq}t$. Hence, the problem arises with non-antisymmetric recursive types, where we have $\mu t.\tau{\leq}\mu t'.\tau'$ and $\mu t'.\tau'{\leq}\mu t.\tau$ (in which case we're also guaranteed that $\mu t.\tau{\neq}\mu t'.\tau'$ because $t{\neq}t'$ is required, as the previous paragraph showed).

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

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

4.2. A Second Source of Incompleteness: Complications with Unrolling

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

— $\tau' = \mu i.\{\texttt{sub}{:}i \rightarrow \texttt{unit}\}$
— $\tau = \mu n.\{\texttt{sub}{:}(\mu i'.\{\texttt{sub}{:}i' \rightarrow \texttt{unit}\}) \rightarrow \texttt{unit},\ \texttt{min}{:}\texttt{unit} \rightarrow \texttt{int}\}$

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

```
class Int {
  //subtract an Int from this Int
  public void sub(Int i) {...}
  ...
}
class Nat extends Int {
  //override Int.sub to avoid negatives
  public void sub(Int i) {...}
  public int min() {0}
  ...
}
```

The `Int` type may be encoded as τ' (with additional fields for members not shown above), and the `Nat` type as τ (also with additional fields). 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 $\{\texttt{sub}{:}\tau' \rightarrow \texttt{unit}\}$, while unrolling a τ-type expression produces an expression of type $\{\texttt{sub}{:}(\mu i'.\{\texttt{sub}{:}i' \rightarrow \texttt{unit}\}) \rightarrow \texttt{unit}, \texttt{min}{:}\texttt{unit} \rightarrow \texttt{int}\}$, which is a subtype of $\{\texttt{sub}{:}\tau' \rightarrow \texttt{unit}\}$. 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 2 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 n in the previous example's τ). Again, the Amber rules (in conjunction with the standard subtyping rule for binary sums) can't be used

$$\dfrac{\dfrac{\dfrac{\dfrac{\Downarrow\ \texttt{Derivation fails here}\ \Downarrow}{\{n\le i\}\vdash i\le \mu i'.\{\texttt{sub}{:}i'\to\texttt{unit}\}}\qquad \dfrac{}{\{n\le i\}\vdash \texttt{unit}\le\texttt{unit}}}{\{n\le i\}\vdash (\mu i'.\{\texttt{sub}{:}i'\to\texttt{unit}\})\to\texttt{unit}\le i\to\texttt{unit}}}{\{n\le i\}\vdash \{\texttt{sub}{:}(\mu i'.\{\texttt{sub}{:}i'\to\texttt{unit}\})\to\texttt{unit},\ \texttt{min}{:}\texttt{unit}\to\texttt{int}\}\le\{\texttt{sub}{:}i\to\texttt{unit}\}}}{\mu n.\{\texttt{sub}{:}(\mu i'.\{\texttt{sub}{:}i'\to\texttt{unit}\})\to\texttt{unit},\ \texttt{min}{:}\texttt{unit}\to\texttt{int}\}\le\mu i.\{\texttt{sub}{:}i\to\texttt{unit}\}}$$

Fig. 2. Attempted derivation of $\mu n.\{\texttt{sub}{:}(\mu i'.\{\texttt{sub}{:}i'\to\texttt{unit}\})\to\texttt{unit},\ \texttt{min}{:}\texttt{unit}\to\texttt{int}\}\le\mu i.\{\texttt{sub}{:}i\to\texttt{unit}\}$, using the Amber rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Downarrow\ \texttt{Derivation fails here}\ \Downarrow}{\{a\le c\}\vdash \mu b.((b+\texttt{nat})+a)\le c}\qquad\dfrac{}{\{a\le c\}\vdash\texttt{nat}\le\texttt{real}}}{\{a\le c\}\vdash(\mu b.((b+\texttt{nat})+a))+\texttt{nat}\le c+\texttt{real}}\qquad\dfrac{}{\{a\le c\}\vdash a\le c}}{\{a\le c\}\vdash((\mu b.((b+\texttt{nat})+a))+\texttt{nat})+a\le(c+\texttt{real})+c}}{\mu a.(((\mu b.((b+\texttt{nat})+a))+\texttt{nat})+a)\le\mu c.((c+\texttt{real})+c)}$$

Fig. 3. Attempted derivation of $\mu a.(((\mu b.((b+\texttt{nat})+a))+\texttt{nat})+a)\le\mu c.((c+\texttt{real})+c)$, using the Amber rules.

$$\begin{array}{rrcl}
\text{Types} & \overline{\tau} & ::= & \texttt{nat}\mid\texttt{real}\mid\overline{\tau}_1\to\overline{\tau}_2\mid\overline{\tau}_1+\overline{\tau}_2\mid\overline{\tau}_1\times\overline{\tau}_2\mid\mu t.\overline{\tau}\mid t\\
\text{Expressions} & e & ::= & \texttt{n}\mid\texttt{r}\mid\texttt{succ}(e)\mid\texttt{neg}(e)\mid\lambda x{:}\tau.e\mid e_1(e_2)\mid x\mid\\
 & & & \texttt{inl}_{\tau_1+\tau_2}(e)\mid\texttt{inr}_{\tau_1+\tau_2}(e)\mid\texttt{case}_\tau\ e\ \texttt{ of inl}\,x\Rightarrow e_1\ \texttt{else inr}\,y\Rightarrow e_2\mid\\
 & & & (e_1,e_2)\mid e.\texttt{fst}\mid e.\texttt{snd}\mid\texttt{unroll}(e)\mid\texttt{roll}_{\mu t.\overline{\tau}}(e)
\end{array}$$

Fig. 4. Syntax of λ_{ADT} .

to derive $\tau\le\tau'$, as shown in Figure 3. We prove that it's safe to consider $\tau\le\tau'$ in two steps: first, Section 5 shows that $\tau\le\tau'$ is derivable using new subtyping rules; second, Appendix A shows that the new subtyping rules are indeed sound with respect to type safety.

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

5. A PRECISE SYSTEM FOR SUBTYPING ISO-RECURSIVE TYPES

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

5.1. A Language with Algebraic Data Types, λ_{ADT}

Let's define a new language, λ_{ADT} , by adding binary (disjoint) sum, binary product, and iso-recursive types to λ. Figures 4–6 present the syntax and static and dynamic semantics. Again, all the notation is intended to have the usual meanings, with the usual assumptions being made.

Types $\overline{\tau}$ in λ_{ADT} may be open (i.e., have free type variables), but it'll often be useful to refer specifically to closed types. Let metavariable τ range over closed types (i.e., the subset of $\overline{\tau}$ that have no free variables). Note that unrolling a closed recursive type $\tau=\mu t.\overline{\tau}$ produces another closed type, $\tau_u=[\mu t.\overline{\tau}/t]\overline{\tau}$.

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

$$\frac{}{\Gamma \vdash \mathtt{n} : \mathtt{nat}}\ \text{T-Nat} \qquad \frac{}{\Gamma \vdash \mathtt{r} : \mathtt{real}}\ \text{T-Real} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(e) : \mathtt{nat}}\ \text{T-Succ}$$

$$\frac{\Gamma \vdash e : \mathtt{real}}{\Gamma \vdash \mathtt{neg}(e) : \mathtt{real}}\ \text{T-Neg} \qquad \frac{\Gamma \cup \{x{:}\tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \rightarrow \tau'}\ \text{T-Lam}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ \text{T-App} \qquad \frac{}{\Gamma \cup \{x{:}\tau\} \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{inl}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ \text{T-Left} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{inr}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ \text{T-Right}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \text{T-Prod} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.\mathtt{fst}\ : \tau_1}\ \text{T-Fst} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.\mathtt{snd}\ : \tau_2}\ \text{T-Snd}$$

$$\frac{\Gamma \vdash e : \mu t.\overline{\tau}}{\Gamma \vdash \mathtt{unroll}(e) : [\mu t.\overline{\tau}/t]\overline{\tau}}\ \text{T-Unroll} \qquad \frac{\Gamma \vdash e : [\mu t.\overline{\tau}/t]\overline{\tau}}{\Gamma \vdash \mathtt{roll}_{\mu t.\overline{\tau}}(e) : \mu t.\overline{\tau}}\ \text{T-Roll}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau}\ \text{T-Subsume}$$

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

$$\frac{\Gamma \cup \{x{:}\tau'\} \vdash E[x] : \tau \qquad (x \notin E)}{\Gamma \vdash E[\tau'] : \tau}\ \text{T-Ctxt}$$

Fig. 5. Static semantics of λ_{ADT} .

5.2. The Subtyping Rules for λ_{ADT}

Incompleteness in the Amber rules (for subtyping iso-recursive types) ultimately stems from their lack of considering unrolled types. Iso-recursive types get eliminated by unrolling, so type $\mu t.\overline{\tau}$ should 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 rule for iso-recursive types:

$$\frac{(\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}') \in S \quad or \quad S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'}{S \vdash \mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'}\ \text{S-Rec}$$

A few notes:

$$\text{Evaluation contexts } E ::= [\,] \mid \texttt{succ}(E) \mid \texttt{neg}(E) \mid E\ (e) \mid v\ (E) \mid (E,e) \mid (v,E) \mid E.\texttt{fst} \mid E.\texttt{snd} \mid \texttt{inl}_{\tau_1+\tau_2}(E) \mid \texttt{inr}_{\tau_1+\tau_2}(E) \mid \texttt{unroll}(E) \mid \texttt{roll}_{\mu t.\overline{\tau}}(E) \mid \texttt{case}_\tau\ E\ \texttt{ of inl}\,x \Rightarrow e_1 \texttt{ else inr}\,y \Rightarrow e_2$$

$$\text{Values } v ::= \texttt{n} \mid \texttt{r} \mid \lambda x:\tau.e \mid (v_1,v_2) \mid \texttt{inl}_{\tau_1+\tau_2}(v) \mid \texttt{inr}_{\tau_1+\tau_2}(v) \mid \texttt{roll}_{\mu t.\overline{\tau}}(v)$$

$\boxed{e \mapsto e'}$

$$\frac{e \mapsto_\beta e'}{E[e] \mapsto E[e']}$$

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

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

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

$$\frac{\texttt{n}' = \texttt{n}+1}{\texttt{succ}(\texttt{n}) \mapsto_\beta \texttt{n}'}\ \beta\text{-S}\small{\text{UCC}} \qquad \frac{\texttt{r}' = -\texttt{r}}{\texttt{neg}(\texttt{r}) \mapsto_\beta \texttt{r}'}\ \beta\text{-RN}\small{\text{EG}} \qquad \frac{\texttt{r} = -\texttt{n}}{\texttt{neg}(\texttt{n}) \mapsto_\beta \texttt{r}}\ \beta\text{-NN}\small{\text{EG}}$$

$$\frac{}{(\lambda x:\tau.e)(v) \mapsto_\beta [v/x]e}\ \beta\text{-A}\small{\text{PP}} \qquad \frac{}{(v_1,v_2).\texttt{fst} \mapsto_\beta v_1}\ \beta\text{-F}\small{\text{ST}} \qquad \frac{}{(v_1,v_2).\texttt{snd} \mapsto_\beta v_2}\ \beta\text{-S}\small{\text{ND}}$$

$$\frac{}{\texttt{case}_\tau\ \texttt{inl}_{\tau_1+\tau_2}(v) \texttt{ of inl}\,x \Rightarrow e_2 \texttt{ else inr}\,y \Rightarrow e_3 \mapsto_\beta [v/x]e_2}\ \beta\text{-L}\small{\text{EFT}}$$

$$\frac{}{\texttt{case}_\tau\ \texttt{inr}_{\tau_1+\tau_2}(v) \texttt{ of inl}\,x \Rightarrow e_2 \texttt{ else inr}\,y \Rightarrow e_3 \mapsto_\beta [v/y]e_3}\ \beta\text{-R}\small{\text{IGHT}}$$

$$\frac{}{\texttt{unroll}(\texttt{roll}_{\mu t.\overline{\tau}}(v)) \mapsto_\beta v}\ \beta\text{-U}\small{\text{NROLL}}$$

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

$$\frac{}{e \mapsto^* e} \qquad \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}$$

Fig. 6. Dynamic semantics of λ_{ADT} .

— As with other judgment forms that use contexts, this paper abbreviates judgments of the form $\emptyset \vdash \tau_1 {\leq} \tau_2$ as $\tau_1 {\leq} \tau_2$.
— S-REC maintains the invariant that only closed types are being considered; unrolling a closed type produces another closed type.
— Other systems have used rules similar to S-REC to define equivalence, rather than subtyping, relations on iso-recursive types [League and Shao 1998; Vanderwaart et al. 2003].

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

Interestingly, S-REC is insufficient for making the subtyping relation complete (as we learned by attempting an early proof of completeness). Because λ_{ADT} has recursive types, every type is inhabited—for all τ let d be $\lambda x{:}\mu t.(t{\to}\tau).(\texttt{unroll}(x)\ x)$; then the nonterminating expression $d(\texttt{roll}_{\mu t.(t\to\tau)}(d))$ has type τ. However, some types are value-uninhabited (i.e., inhabited only by nonterminating expressions). For example,

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F\vdash I\leq I'}\qquad\dfrac{}{F\vdash \mathtt{unit}\leq\mathtt{unit}}}{F\vdash I'\to\mathtt{unit}\leq I\to\mathtt{unit}}}{F\vdash\{\mathtt{sub}{:}I'\to\mathtt{unit}\}\leq\{\mathtt{sub}{:}I\to\mathtt{unit}\}}}{\{N\leq I, I\leq I'\}\vdash I'\leq I}\qquad\dfrac{}{\{N\leq I,I\leq I'\}\vdash\mathtt{unit}\leq\mathtt{unit}}}{\{N\leq I,I\leq I'\}\vdash I\to\mathtt{unit}\leq I'\to\mathtt{unit}}}{\{N\leq I,I\leq I'\}\vdash\{\mathtt{sub}{:}I\to\mathtt{unit}\}\leq\{\mathtt{sub}{:}I'\to\mathtt{unit}\}}}{\{N\leq I\}\vdash I\leq I'}\qquad\dfrac{}{\{N\leq I\}\vdash\mathtt{unit}\leq\mathtt{unit}}}{\{N\leq I\}\vdash I'\to\mathtt{unit}\leq I\to\mathtt{unit}}}{\{N\leq I\}\vdash\{\mathtt{sub}{:}I'\to\mathtt{unit},\mathtt{min}{:}\mathtt{unit}\to\mathtt{int}\}\leq\{\mathtt{sub}{:}I\to\mathtt{unit}\}}}{N\leq I}
$$

Fig. 7. Derivation of $N{\leq}I$ using the new subtyping rule, where $I{=}\mu i.\{\mathtt{sub}{:}i{\to}\mathtt{unit}\}$, $I'{=}\mu i'.\{\mathtt{sub}{:}i'{\to}\mathtt{unit}\}$, $N{=}\mu n.\{\mathtt{sub}{:}I'{\to}\mathtt{unit},\mathtt{min}{:}\mathtt{unit}{\to}\mathtt{int}\}$, and $F = \{N{\leq}I, I{\leq}I', I'{\leq}I\}$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{S{\vdash}B\leq C}\qquad\dfrac{}{S{\vdash}\mathtt{nat}\leq\mathtt{real}}}{S{\vdash}B+\mathtt{nat}\leq C+\mathtt{real}}\qquad\dfrac{}{S{\vdash}A\leq C}}{S{\vdash}(B+\mathtt{nat})+A\leq(C+\mathtt{real})+C}}{\{A\leq C\}{\vdash}B\leq C}\qquad\dfrac{}{\{A\leq C\}{\vdash}\mathtt{nat}\leq\mathtt{real}}}{\{A\leq C\}{\vdash}B+\mathtt{nat}\leq C+\mathtt{real}}\qquad\dfrac{}{\{A\leq C\}{\vdash}A\leq C}}{\{A\leq C\}{\vdash}(B+\mathtt{nat})+A\leq(C+\mathtt{real})+C}}{A\leq C}
$$

Fig. 8. Derivation of $A{\leq}C$ using the new subtyping rule, where $A{=}\mu a.(((\mu b.((b+\mathtt{nat})+a))+\mathtt{nat})+a)$, $B{=}\mu b.((b+\mathtt{nat})+A)$, $C{=}\mu c.((c+\mathtt{real})+c)$, and $S = \{A{\leq}C, B{\leq}C\}$.

the type $\mu t.t$ is uninhabited by (normal-form) values; writing a value of type $\mu t.t$ would require already having a value of type $\mu t.t$ to roll. Hence, every expression of type $\mu t.t$ must diverge.

We can treat any type inhabited only by diverging expressions, such as $\mu t.t$, as being equivalent to a $\perp$ type. If all expressions of a type 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 safely substitute any expression—of any type—for the function.

Based on the preceding discussion, we add the following rules to the definition of subtyping in λ_{ADT} .

$$
\frac{\mathrm{val}(\tau)=\emptyset}{S\vdash\tau{\leq}\tau'}\ \text{S-}\perp \qquad\qquad \frac{\mathrm{val}(\tau_1')=\emptyset}{S\vdash\tau\leq\tau_1'{\to}\tau_2'}\ \text{S-}\top
$$

These rules use an auxiliary judgment of the form $\mathrm{val}(\tau)=\emptyset$ to indicate that τ is value-uninhabited. Vouillon describes rules similar to S-$\perp$ and S-$\top$ [Vouillon 2004]. As part of an algorithm to decide subtyping using the denotational approach (where

$\boxed{S \vdash \tau \leq \tau'}$

$$\frac{}{S \vdash \texttt{nat} \leq \texttt{real}}\ \text{S-BASE} \qquad \frac{}{S \vdash \texttt{nat} \leq \texttt{nat}}\ \text{S-NAT} \qquad \frac{}{S \vdash \texttt{real} \leq \texttt{real}}\ \text{S-REAL}$$

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

$$\frac{S \vdash \tau_1 \leq \tau_1' \qquad S \vdash \tau_2 \leq \tau_2'}{S \vdash \tau_1 + \tau_2 \leq \tau_1' + \tau_2'}\ \text{S-SUM} \qquad \frac{S \vdash \tau_1 \leq \tau_1' \qquad S \vdash \tau_2 \leq \tau_2'}{S \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}\ \text{S-PROD}$$

$$\frac{(\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}') \in S \quad or \quad S \cup \{\mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'\} \vdash [\mu t.\overline{\tau}/t]\overline{\tau} \leq [\mu t'.\overline{\tau}'/t']\overline{\tau}'}{S \vdash \mu t.\overline{\tau} \leq \mu t'.\overline{\tau}'}\ \text{S-REC}$$

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

$$\frac{U \vdash \text{val}(\tau_1) = \emptyset \qquad U \vdash \text{val}(\tau_2) = \emptyset}{U \vdash \text{val}(\tau_1 + \tau_2) = \emptyset}\ \text{U-SUM}$$

$$\frac{U \vdash \text{val}(\tau_1) = \emptyset \quad or \quad U \vdash \text{val}(\tau_2) = \emptyset}{U \vdash \text{val}(\tau_1 \times \tau_2) = \emptyset}\ \text{U-PROD}$$

$$\frac{(\mu t.\overline{\tau}) \in U \quad or \quad U \cup \{\mu t.\overline{\tau}\} \vdash \text{val}([\mu t.\overline{\tau}/t]\overline{\tau}) = \emptyset}{U \vdash \text{val}(\mu t.\overline{\tau}) = \emptyset}\ \text{U-REC}$$

Fig. 9. Subtyping and value-uninhabitation rules for λ_{ADT} .

$\tau \leq \tau'$ iff $[\![\tau]\!] \subseteq [\![\tau']\!]$, and $[\![\tau]\!]$ is the set of values of type τ), Frisch, Castagna, and Benzaken provide an algorithm for deciding value-uninhabitation in an equi-recursive system [Frisch 2004; Frisch et al. 2008].

Combining rules S-REC, S-$\bot$, and S-$\top$ with the standard rules for subtyping `nat`, `real`, function, sum, and product types produces the subtyping system shown in Figure 9. This is the full definition of the subtyping relation for λ_{ADT} .

Figure 9 contains rules for deciding value-uninhabitation. The `nat`, `real`, and function types are always value-inhabited. Sum type $\tau_1 + \tau_2$ is value-uninhabited when both τ_1 and τ_2 are value-uninhabited, and product type $\tau_1 \times \tau_2$ is value-uninhabited when τ_1 or τ_2 is value-uninhabited. Finally, recursive type τ is value-uninhabited when the unrolled version of τ is value-uninhabited under the assumption that τ is value-uninhabited (because we can't make a value of type τ by relying on already having one).

Appendix A contains a preciseness proof for this subtyping relation. Along the way, the proof shows that the subtyping system is navigable (Lemma 13), value-uninhabitation is defined correctly ($\text{val}(\tau) = \emptyset$ iff no value of type τ exists), and the subtyping relation is indeed reflexive and transitive (without explicit rules stating so).

5.3. A Subtyping Algorithm

Because the subtyping system in Figure 9 is navigable, an algorithm exists for deciding whether subtyping judgments are derivable: simply search for a (possibly failing) derivation.

This simple subtyping algorithm can be optimized to prevent redundant computations. Figures 10–12 present one such implementation in Standard ML. This implementation is a complete but lightly edited version of the actual implementation posted online [Ligatti 2016b]. The actual implementation is 72 lines of code, not counting whitespace and comments. The comments in Figures 10–12 explain the optimized algorithm's operation and correctness.

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

— m is the number of μ-terms (i.e., variable declarations) in τ_1 or τ_2, whichever is greater (or 1 if neither contain μ-terms)
— n is the total size of τ_1 or τ_2, whichever is greater.

The main subtyping function, `sub` in Figure 12, first counts the number of variable declarations in its argument types `t1` and `t2`, and then allocates tables (`UT1`, `UT2`, `U1`, and `U2`) of these sizes, all in $O(n)$ time. The `sub` function next calls `init` (Figure 11) on each of `t1` and `t2`, in order to (1) build `CEtyp`-versions of `t1` and `t2`, and (2) properly initialize the previously allocated tables.

The `init` function implements the $\mathrm{val}(\tau){=}\emptyset$ judgment on all n component types of τ and runs in $O(mn)$ time. This function traverses a given type tree and commits to the value-uninhabitation of each of its m recursive types in turn, from outer recursive types to inner recursive types. This outer-to-inner ordering is important because the value-inhabitation of inner recursive types may depend on the value-inhabitation of outer recursive types. For example, with types of the form $X \equiv \mu x.((\mu y.x) + \tau')$, the value-inhabitation of the inner $Y \equiv \mu y.X$ depends on the value-inhabitation of the outer X (if τ' is `nat` then X is value-inhabited, causing Y to be value-inhabited, but if τ' is x then X is value-uninhabited, causing Y to be value-uninhabited). Each of the m commits in `init` requires traversing the recursive type's subtree in $O(n)$ time (all the other cases of `init`, which don't involve committing to the value-uninhabitation of a recursive type, run in time that's a constant plus the time required to `init` subtrees, for a total time that's proportional to the size of the subtree being considered). Note that `init` runs in $O(mn)$ time because it initializes tables for *all* the component types of its type argument; in applications where we only care to test whether one overall type is value-inhabited, we could simply call `init` with the final `b` argument set to `true`, in which case `init` decides value-inhabitation in $O(n)$ time.

After `init` has completed, all value-uninhabitation checks and recursive-type unrolling can be performed in constant time. At this point, `sub` allocates and initializes two tables (`S1` and `S2`) for storing recursive-type subtyping assumptions, in $O(m^2)$ time.

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

Thus, the total running time of `sub` is $O(n)$ (to allocate `UT1`, `UT2`, `U1`, and `U2`) plus $O(mn)$ (to run `init`) plus $O(m^2)$ (to allocate `S1` and `S2`) plus $O(mn)$ (to run `subh`). Because $0 < m < n$, the total running time of the subtyping algorithm is $O(mn)$.

```
(* Constructors for types.  Type variables are represented as
 * integers, which are assumed to be named 0, 1, etc., so for all
 * types T passed as arguments to the subtype-testing function
 * sub, the set of type variables in T is {0..n} for some n.
 * We also assume that T never uses undeclared variables and has
 * been alpha-converted to ensure the uniqueness of every
 * declared variable.
 * As an example, the type A from Figure 8 could be encoded as:
 * Rec(0,Sum(Sum(Rec(1,Sum(Sum(Var(1),Nat),Var(0))),Nat),Var(0)))
 *)
datatype typ = Nat | Real | Prod of typ * typ | Sum of typ * typ
             | Fun of typ * typ | Rec of int * typ | Var of int;

(* A CEtyp is a 'compressed' and 'extended' type.
 * 'compressed' means that all recursive types \mu n.t have been
 *    replaced by just the type variable n.  We'll still be able
 *    to look up the type to which n refers in an 'unroll table',
 *    an array that maps n to (the CEtyp-version of) t.
 *    Hence, CEtyp has no case for recursive types.
 * 'extended' means that the structure carries extra boolean
 *    flags to memoize whether types are value-uninhabited.
 *    Nat, real, and function types in this language are always
 *    value-inhabited, so their cases of CEtyp don't need the
 *    extra flag.  Variable types also don't need the flag; we'll
 *    instead use a separate array U to map type variables to
 *    bools indicating value-uninhabitation.
 *)
datatype CEtyp = CENat | CEReal | CEFun of CEtyp * CEtyp
               | CEVar of int | CEProd of CEtyp * CEtyp * bool
               | CESum of CEtyp * CEtyp * bool;

(* Returns the number of variables defined in a type. *)
fun numVars (Sum(t1,t2)) = numVars(t1) + numVars(t2)
  | numVars (Prod(t1,t2)) = numVars(t1) + numVars(t2)
  | numVars (Fun(t1,t2)) = numVars(t1) + numVars(t2)
  | numVars (Rec(_,t1)) = numVars(t1) + 1
  | numVars _ = 0;

(* Returns a bool indicating whether a given CEtyp is
 * value-uninhabited.  The second parameter is an array
 * mapping type variables to value-uninhabitation flags.
 * That is, U[n] iff the recursive type to which
 * type-variable n refers is value-uninhabited.
 *)
fun isUninhabited (CEProd(_,_,b)) _ = b
  | isUninhabited (CESum(_,_,b)) _ = b
  | isUninhabited (CEVar(n)) U = Array.sub(U,n)
  (* nat, real, and function types are value-inhabited *)
  | isUninhabited _ _ = false;
```

Fig. 10. Auxiliary definitions for the optimized subtyping algorithm.

```
(* This initialization function has 4 parameters:
 *   (1) a typ t
 *   (2) an unroll table UT (having size numVars(t))
 *   (3) an array U (also having size numVars(t)) mapping type
 *        variables to value-uninhabitation flags
 *   (4) a boolean b indicating whether we're trying to commit
 *        to the value-(un)inhabitation of some previously seen
 *        recursive type.
 * This function returns the CEtyp-version of t and properly
 * initializes the UT and U arrays (as side effects).
 *)
fun init Nat _ _ _ = CENat
  | init Real _ _ _ = CEReal
  | init (Fun(t1,t2)) UT U b =
      CEFun(init t1 UT U b, init t2 UT U b)
  | init (Sum(t1,t2)) UT U b =
      let val CEt1 = init t1 UT U b
          val CEt2 = init t2 UT U b
      in (* set the value-uninhabited flag based on rule U-Sum *)
        CESum(CEt1, CEt2,
          isUninhabited CEt1 U andalso isUninhabited CEt2 U)
      end
  | init (Prod(t1,t2)) UT U b =
      let val CEt1 = init t1 UT U b
          val CEt2 = init t2 UT U b
      in (* set the value-uninhabited flag based on rule U-Prod *)
        CEProd(CEt1, CEt2,
          isUninhabited CEt1 U orelse isUninhabited CEt2 U)
      end
  | init (Rec(n,t)) UT U b =
      (* Recursive type n is value-uninhabited iff t is
       * value-uninhabited under the assumption that n is
       * value-uninhabited (U-Rec).  Once we know whether t
       * is value-inhabited, we can properly set U[n].
       * Finally, if b=false then we're now committed to U[n] and
       * can move on to processing t, after which we can properly
       * set UT[n] and return the compressed version of \mu n.t,
       * which is just the variable n.
       *)
      (Array.update(U,n, true);
       Array.update(U,n, isUninhabited (init t UT U true) U);
       if b then () else Array.update(UT,n, init t UT U false);
       CEVar(n))
      (* We commit to the value-uninhabitation of recursive types
       * in this outer-to-inner fashion to properly handle types
       * like \mu 0.((\mu 1.0) + tau), in which the value-
       * uninhabitation of an outer type (here, 0) determines the
       * value-uninhabitation of an inner type (here, 1).
       *)
  | init (Var(n)) _ _ _ = CEVar(n);
```

Fig. 11. Computation of value-uninhabitation in the optimized subtyping algorithm.

```
fun sub t1 t2 =
let (* Allocate and initialize the unroll tables UT1 and UT2,
     * the uninhabitation arrays U1 and U2, and the compressed
     * and extended types CEt1 and CEt2. *)
  val m = numVars t1
  val n = numVars t2
  val UT1 = Array.array(m,CENat)
  val UT2 = Array.array(n,CENat)
  val U1 = Array.array(m,false)
  val U2 = Array.array(n,false)
  val CEt1 = init t1 UT1 U1 false
  val CEt2 = init t2 UT2 U2 false
  (* Now create arrays for storing subtyping assumptions.
   * S1[m][n] iff recursive type m in t1 is assumed to subtype
   * recursive type n in t2; similarly, S2[n][m] iff recursive
   * type n in t2 is assumed to subtype recursive type m in t1.*)
  val S1 = Array2.array(m,n,false)
  val S2 = Array2.array(n,m,false)
  (* The following helper subtyping function operates on CEtyp's
   * grouped with UT and U tables, and the S1 and S2 arrays.*)
  fun subh (CEt1,UT1,U1) (CEt2,UT2,U2) (S1,S2) =
      isUninhabited CEt1 U1 (* S-Bottom *)
    orelse (* S-Top *)
     (case CEt2 of CEFun(CEt2',_) => isUninhabited CEt2' U2
      | _ => false)
    orelse
      case (CEt1, CEt2) of
        (CENat, CEReal) => true (* S-Base *)
      | (CENat, CENat) => true (* S-Nat *)
      | (CEReal, CEReal) => true (* S-Real *)
      | (CEFun(t1, t2), CEFun(t1', t2')) => (* S-Fun *)
          subh (t1',UT2,U2) (t1,UT1,U1) (S2,S1) andalso
          subh (t2,UT1,U1) (t2',UT2,U2) (S1,S2)
      | (CESum(t1, t2, _), CESum(t1', t2', _)) => (* S-Sum *)
          subh (t1,UT1,U1) (t1',UT2,U2) (S1,S2) andalso
          subh (t2,UT1,U1) (t2',UT2,U2) (S1,S2)
      | (CEProd(t1, t2, _), CEProd(t1', t2', _)) => (* S-Prod *)
          subh (t1,UT1,U1) (t1',UT2,U2) (S1,S2) andalso
          subh (t2,UT1,U1) (t2',UT2,U2) (S1,S2)
      | (CEVar(m), CEVar(n)) => (* S-Rec *)
          (* Return true if m is assumed to subtype n; otherwise,
           * assume m subtypes n and return whether m-unrolled
           * subtypes n-unrolled *)
          Array2.sub(S1,m,n) orelse
          (Array2.update(S1,m,n,true);
           subh (Array.sub(UT1,m),UT1,U1)
                (Array.sub(UT2,n),UT2,U2) (S1,S2))
      | _ => false
in subh (CEt1,UT1,U1) (CEt2,UT2,U2) (S1,S2) end;
```

Fig. 12. The main function of the optimized subtyping algorithm.

6. DISCUSSION

A few remaining points are worth discussing.

6.1. Evaluation Contexts vs. General Contexts in the Definition of Preciseness

Definition 1 is based on evaluation contexts E rather than general (arbitrary-subexpression) contexts G. General contexts are the evaluation contexts used with the full-β evaluation strategy. For example, G is defined for λ as follows.

$$G ::= [\,] \mid \texttt{succ}(G) \mid \texttt{neg}(G) \mid \lambda x{:}\tau.G \mid G(e) \mid e(G)$$

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

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

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

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

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

Languages λ and λ_{ADT} satisfy the requirements of Proposition 8 and are therefore precise according to the general-context version of Definition 1.

6.2. Subtyping with Strict vs. Nonstrict Evaluation Strategies

The evaluation strategy remains fixed in Proposition 8; the proposition does not imply that a subtyping relation that's precise with one evaluation strategy will be precise with another. On the contrary, the choice of evaluation strategy may affect subtyping.

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

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

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

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

Similar analysis would show the same complications with other nonstrict evaluation strategies, such as the full-β strategy.

In practice, languages that are nonstrict by default may have constructs for switching to strict evaluation. For example, Haskell provides the special functions seq and deepSeq for forcing expressions to be evaluated. By enabling strict evaluation, such languages avoid the complications just described.

6.3. Iso-recursive vs. Equi-recursive Subtyping

This paper's rules for subtyping iso-recursive types are similar, at a high level, to the rules typically used for subtyping equi-recursive types [Amadio and Cardelli 1993; Brandt and Henglein 1998]. Specifically, S-REC follows the standard equi-recursive approaches of (1) considering as subtypes any pair of types previously considered, and (2) unrolling recursive types as they're encountered.

However, some substantial differences exist between this paper's treatment of iso-recursive subtyping and typical treatments of equi-recursive subtyping. One difference is that this paper considers arbitrary recursive types, without syntactic restrictions; concretely, the rules here can derive relationships like $\mu t.(t+t) \leq \mathtt{real} \leq (\mu t.t)\rightarrow\mathtt{nat}$, which have generally been beyond the scope of equi-recursive systems. The most common syntactic restriction on equi-recursive types has pertained to contractiveness [MacQueen et al. 1984], which requires recursive types to have specific shapes like $\mu t.(\overline{\tau}\rightarrow\overline{\tau}')$ rather than the more general $\mu t.\overline{\tau}$ (e.g., [Amadio and Cardelli 1993; Brandt and Henglein 1998; Gapeyev et al. 2002; Frisch et al. 2008; Im et al. 2013]). Contractive types can provide a useful constraint on the shapes of the type trees obtained by unrolling equi-recursive types. Iso-recursive types, on the other hand, do not represent such unrollings (e.g., $\mu t.\mathtt{nat}$ is nat in an equi-recursive, but not iso-recursive, system), so contractiveness does not seem useful in an iso-recursive setting (indeed, languages like ML support non-contractive recursive types). A different syntactic restriction on recursive types, specific to the domain of regular-expression types, is used in [Hosoya et al. 2005].

Another difference between iso- and equi-recursive subtyping relates to the "synchronous" unrolling used in this paper's rules (i.e., unrolling both types under consideration), versus the "asynchronous" unrolling commonly used for subtyping equi-recursive types (i.e., unrolling only one of the two types under consideration) [Amadio and Cardelli 1993; Brandt and Henglein 1998; Gapeyev et al. 2002]. Again, this difference stems from the implicit equality of an equi-recursive type with its unrolling (e.g., $\mu t.\mathtt{nat}\leq\mathtt{real}$ in an equi-recursive, but not iso-recursive, system). Intuitively, iso-recursive types are eliminated through explicit unroll operations, so matching μ's are required for subtyping iso-recursive types. Although beyond the scope of the present paper, it seems that equi-recursive subtypes could automatically be translated into iso-recursive subtypes by inserting any "missing μ's".

This difference between synchronous (iso-recursive) and asynchronous (equi-recursive) unrollings underlies the difference in the efficiency of subtyping algorithms. Although the subtyping-helper function `subh` in Figure 12 is similar to Brandt and Henglein's algorithm for subtyping equi-recursive types [Brandt and Henglein 1998], the most efficient known equi-recursive subtyping algorithms have $O(n^2)$ running time [Kozen et al. 1995; Brandt and Henglein 1998], while this paper's iso-recursive subtyping algorithm has $O(mn)$ running time. The $O(mn)$ bound improves on $O(n^2)$ because m is independent from, and smaller than, n; for example, the recursive type `typ` defined in Figure 10 has $n{\geq}17$ (its precise size depends on how variant types are represented) but $m{=}1$. The $O(mn)$ bound derives from synchronous unrolling (every type in a μ-term on one side of the $\leq$ may be compared at most once covariantly and at most once contravariantly to a corresponding type in each of the other side's μ-terms), while the $O(n^2)$ bound derives from asynchronous unrolling (every type in a μ-term on one side of the $\leq$ may be compared at most once covariantly and at most once contravariantly to a type τ on the other side, where τ is not limited to being a corresponding type within a μ-term). Because none of this paper's techniques address asynchronous type unrolling, we believe that none of this paper's techniques could be used to improve the $O(n^2)$ bound for subtyping equi-recursive types.

Acknowledgments. Many thanks to the anonymous reviewers for their insightful feedback. Special thanks to Michael Nachtigal for asking, after I'd presented object types similar to those in Section 4.2 to my 2011 PLs class, whether rules exist to show that they're subtypes. Thanks also to Michael and Jeremy for helping research related work, prepare a first version of the SML implementation, and typeset the paper.

REFERENCES

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993), 575–631.

Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2011. Union and Intersection Types for Secure Protocol Implementations. In *Proceedings of Theory of Security and Applications (TOSCA)*.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *The Journal of Symbolic Logic* 48, 4 (Dec. 1983), 931–940.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 8.

Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

Luca Cardelli. 1986. Amber. *Combinators and functional programming languages* (1986), 21–47.

D. Colazzo and G. Ghelli. 2005. Subtyping, Recursion and Parametric Polymorphism in Kernel Fun. *Information and Computation* 198, 2 (2005), 71–147.

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

Mariangiola Dezani-Ciancaglini and Silvia Ghilezan. 2014. Preciseness of Subtyping on Intersection and Union Types. In *Proceedings of Rewriting and Typed Lambda Calculi (RTA-TLCA)*, Gilles Dowek (Ed.). Lecture Notes in Computer Science, Vol. 8560. Springer International Publishing, 194–207.

Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. Ph.D. Dissertation. Université Paris 7.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4 (Sept. 2008), 19:1–19:64.

Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. 2002. Recursive subtyping revealed. *J. Funct. Program.* 12, 6 (2002), 511–548.

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

Robert Harper. 2013. *Practical Foundations for Programming Languages*. http://www.cs.cmu.edu/~rwh/plbook/ Version 1.33 of 05.07.2013, Working Draft.

Haruo Hosoya, Benjamin C. Pierce, and David N. Turner. 1998. Datatypes and Subtyping. (1998). Manuscript.

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.* 27, 1 (Jan. 2005), 46–90.

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

D. Kozen, J. Palsberg, and M.I. Schwartzbach. 1995. Efficient recursive subtyping. *Math. Structures in Comp. Sci.* 5, 1 (1995), 113–125.

Christopher League and Zhong Shao. 1998. *Formal semantics of the FLINT intermediate language*. Technical Report Yale-CS-TR-1171. Yale University.

Jay Ligatti. 2016a. *Induction on Failing Derivations*. Technical Report PL-Sep13. Univ. of South Florida. http://www.cse.usf.edu/~ligatti/papers/iotFdoJ.pdf

Jay Ligatti. 2016b. Subtyping-Algorithm Implementation. http://www.cse.usf.edu/~ligatti/projects/completeness/sub.sml. (Feb. 2016).

Barbara H. Liskov and Jeanette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16 (1994), 1811–1841.

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1984. An Ideal Model for Recursive Polymorphic Types. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 165–174.

Benjamin C. Pierce. 1991. *Programming with Intersection Types and Bounded Polymorphism*. Ph.D. Dissertation. Carnegie Mellon University.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Cees Pierik and Frank S. De Boer. 2005. On behavioral subtyping and completeness. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs*.

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60–61 (2004), 17–139.

Tatsurou Sekiguchi and Akinori Yonezawa. 1994. A Complete Type Inference System for Subtyped Recursive Types. In *Proceedings of Theoretical Aspects of Computer Software (TACS)*. 667–686.

Anthony J. H. Simons. 1994. *Adding Axioms to Cardelli-Wegner Subtyping*. Technical Report CS-94-6. University of Sheffield.

Anthony J. H. Simons. 2002. The Theory of Classification, Part 4: Object Types and Subtyping. *Journal of Object Technology* 1, 5 (2002), 27–35.

Christopher A. Stone and Andrew P. Schoonmaker. 2005. Equational Theories with Recursive Types. (2005). http://www.cs.hmc.edu/~stone/papers/stone-schoonmaker-long.pdf

Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

S. van Bakel, M. Dezani-Ciancaglini, U. deLiguoro, and Y. Motohama. 2000. *The Minimal Relevant Logic and the Call-by-Value Lambda Calculus*. Technical Report TR-ARP-05-2000. The Australian National University.

Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. 2003. Typed compilation of recursive datatypes. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*.

Jérôme Vouillon. 2004. Subtyping Union Types. In *Proceedings of the 18th International Workshop on Computer Science Logic*.

Jérôme Vouillon. 2006. Polymorphic Regular Tree Types and Patterns. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 103–114.

A. PROOF OF PRECISENESS FOR THE SUBTYPING RELATION IN λ_{ADT}

The following proof shows that the subtyping relation $\leq$ defined in Figure 9 is precise with respect to type safety.

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

The proof begins with many "sanity checks" on the `val` and $\leq$ relations (from Lemma 9 to Corollary 19). The first two lemmas are simple context-weakening results.

LEMMA 9. *Value-Uninhabitation Weakening.*

$$\forall\, U, \tau, U' \supseteq U : (U \vdash \mathrm{val}(\tau) = \emptyset \;\Rightarrow\; U' \vdash \mathrm{val}(\tau) = \emptyset)$$

PROOF. By straightforward induction on the derivation of $U \vdash \mathrm{val}(\tau) = \emptyset$. □

LEMMA 10. *Subtype Weakening.*

$$\forall S, \tau_1, \tau_2, S' \supseteq S : (S \vdash \tau_1 {\leq} \tau_2 \;\Rightarrow\; S' \vdash \tau_1 {\leq} \tau_2)$$

PROOF. By straightforward induction on the derivation of $S \vdash \tau_1 {\leq} \tau_2$. □

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

LEMMA 11. *Unrolled Value-Uninhabitation.*

$$\forall\, t, \overline{\tau} : (\mathrm{val}(\mu t.\overline{\tau}) {=} \emptyset \;\Rightarrow\; \mathrm{val}([\mu t.\overline{\tau}/t]\overline{\tau}) {=} \emptyset)$$

PROOF. The only rule deriving $\mathrm{val}(\mu t.\overline{\tau}) {=} \emptyset$ is U-REC, so by inversion of that rule, $\{\mu t.\overline{\tau}\} \vdash \mathrm{val}([\mu t.\overline{\tau}/t]\overline{\tau}) {=} \emptyset$. Hence, by Lemma 9, for all U there exists a derivation forest D_U such that $\dfrac{D_U}{U \cup \{\mu t.\overline{\tau}\} \vdash \mathrm{val}([\mu t.\overline{\tau}/t]\overline{\tau}) {=} \emptyset}$ is a valid derivation. Now construct a new derivation forest $D' = D_\emptyset$, except that D' (1) removes all $\mu t.\overline{\tau}$ value-uninhabitation assumptions from $D_\emptyset$, and then (2) replaces all leaf-node judgments of the form $U \cup \{\mu t.\overline{\tau}\} \vdash \mathrm{val}(\mu t.\overline{\tau}) {=} \emptyset$ in $D_\emptyset$ with the derivation tree $\dfrac{\dfrac{D_U}{U \cup \{\mu t.\overline{\tau}\} \vdash \mathrm{val}([\mu t.\overline{\tau}/t]\overline{\tau}) {=} \emptyset}}{U \vdash \mathrm{val}(\mu t.\overline{\tau}) {=} \emptyset}$. Then $\dfrac{D'}{\mathrm{val}([\mu t.\overline{\tau}/t]\overline{\tau}) {=} \emptyset}$ is a valid derivation tree because D' derives as does $D_\emptyset$, but without requiring an initial $\mu t.\overline{\tau}$ value-uninhabitation assumption. □

LEMMA 12. *Unrolled Subtyping.*

$$\forall t_1, t_2, \overline{\tau}_1, \overline{\tau}_2 : (\mu t_1.\overline{\tau}_1 {\leq} \mu t_2.\overline{\tau}_2 \;\Rightarrow\; [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1 {\leq} [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2)$$

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 $\tau_{2u} = [\tau_2/t_2]\overline{\tau}_2$. The only rules for deriving $\tau_1 {\leq} \tau_2$ are S-$\bot$ and S-REC. In the S-$\bot$ case, $\mathrm{val}(\tau_1) {=} \emptyset$, so by Lemma 11, $\mathrm{val}(\tau_{1u}) {=} \emptyset$, implying by S-$\bot$ that $\tau_{1u} {\leq} \tau_{2u}$, as required. In the S-REC case, we assume $\{\tau_1 {\leq} \tau_2\} \vdash \tau_{1u} {\leq} \tau_{2u}$, so by Lemma 10, for all S there exists a derivation-forest

If J has the form...	then $f(J)$ is...
$S{\vdash}\texttt{nat}{\leq}\texttt{real}$	$() \ \vee\ (\mathrm{val}(\texttt{nat}){=}\emptyset)$
$S{\vdash}\texttt{nat}{\leq}\texttt{nat}$	$() \ \vee\ (\mathrm{val}(\texttt{nat}){=}\emptyset)$
$S{\vdash}\texttt{real}{\leq}\texttt{real}$	$() \ \vee\ (\mathrm{val}(\texttt{real}){=}\emptyset)$
$S{\vdash}\tau_1{\to}\tau_2{\leq}\tau_1'{\to}\tau_2'$	$(\mathrm{val}(\tau_1{\to}\tau_2){=}\emptyset) \vee (\mathrm{val}(\tau_1'){=}\emptyset) \vee (S{\vdash}\tau_1'{\leq}\tau_1 \wedge S{\vdash}\tau_2{\leq}\tau_2')$
$S{\vdash}\tau{\leq}\tau_1'{\to}\tau_2' \quad (\tau \neq \tau_1{\to}\tau_2)$	$(\mathrm{val}(\tau){=}\emptyset) \ \vee\ (\mathrm{val}(\tau_1'){=}\emptyset)$
$S{\vdash}\tau_1{+}\tau_2{\leq}\tau_1'{+}\tau_2'$	$(\mathrm{val}(\tau_1{+}\tau_2){=}\emptyset) \ \vee\ (S{\vdash}\tau_1{\leq}\tau_1' \wedge S{\vdash}\tau_2{\leq}\tau_2')$
$S{\vdash}\tau{\leq}\tau_1'{+}\tau_2' \quad (\tau \neq \tau_1{+}\tau_2)$	$(\mathrm{val}(\tau){=}\emptyset)$
$S{\vdash}\tau_1{\times}\tau_2{\leq}\tau_1'{\times}\tau_2'$	$(\mathrm{val}(\tau_1{\times}\tau_2){=}\emptyset) \ \vee\ (S{\vdash}\tau_1{\leq}\tau_1' \wedge S{\vdash}\tau_2{\leq}\tau_2')$
$S{\vdash}\tau{\leq}\tau_1' \times \tau_2' \quad (\tau \neq \tau_1 \times \tau_2)$	$(\mathrm{val}(\tau){=}\emptyset)$
$S{\cup}\{\tau_\mu{\leq}\tau_\mu'\}{\vdash}\tau_\mu{\leq}\tau_\mu'$	$(\mathrm{val}(\tau_\mu){=}\emptyset) \ \vee\ ()$
$S{\vdash}\tau_\mu{\leq}\tau_\mu' \quad (\tau_\mu{\leq}\tau_\mu' \notin S)$	$(\mathrm{val}(\tau_\mu){=}\emptyset) \ \vee\ (S{\cup}\{\tau_\mu{\leq}\tau_\mu'\}{\vdash}\tau_{\mu u}{\leq}\tau_{\mu u}')$
$S{\vdash}\tau{\leq}\tau_\mu' \quad (\tau \neq \tau_\mu)$	$(\mathrm{val}(\tau){=}\emptyset)$
$U{\vdash}\mathrm{val}(\tau_1{+}\tau_2){=}\emptyset$	$(U{\vdash}\mathrm{val}(\tau_1){=}\emptyset \ \wedge\ U{\vdash}\mathrm{val}(\tau_2){=}\emptyset)$
$U{\vdash}\mathrm{val}(\tau_1{\times}\tau_2){=}\emptyset$	$(U{\vdash}\mathrm{val}(\tau_1){=}\emptyset) \ \vee\ (U{\vdash}\mathrm{val}(\tau_2){=}\emptyset)$
$U{\cup}\{\tau_\mu\}{\vdash}\mathrm{val}(\tau_\mu){=}\emptyset$	$()$
$U{\vdash}\mathrm{val}(\tau_\mu){=}\emptyset \quad (\tau_\mu \notin U)$	$(U{\cup}\{\tau_\mu\}{\vdash}\mathrm{val}(\tau_{\mu u}){=}\emptyset)$
anything else	ε

Fig. 13. Rule function f for the subtyping system of λ_{ADT}. Conjunctive clauses are always parenthesized. Symbol τ_μ denotes a type $\mu t.\overline{\tau}$, and $\tau_{\mu u}$ denotes the unrolled form of τ_μ.

D_S such that $\dfrac{D_S}{S \cup \{\tau_1{\leq}\tau_2\} \vdash \tau_{1u}{\leq}\tau_{2u}}$ is a valid derivation. Now construct a new derivation forest $D' = D_\emptyset$, except that D' (1) removes all $\tau_1{\leq}\tau_2$ subtyping assumptions from $D_\emptyset$, and then (2) replaces all leaf-node judgments of the form $S \cup \{\tau_1{\leq}\tau_2\} \vdash \tau_1{\leq}\tau_2$ in $D_\emptyset$ with the derivation tree $\dfrac{\dfrac{D_S}{S \cup \{\tau_1{\leq}\tau_2\} \vdash \tau_{1u}{\leq}\tau_{2u}}}{S \vdash \tau_1{\leq}\tau_2}$. Then $\dfrac{D'}{\tau_{1u}{\leq}\tau_{2u}}$ is a valid derivation tree because D' derives as does $D_\emptyset$, but without requiring an initial $\tau_1{\leq}\tau_2$ subtyping assumption. □

The next lemma shows that the subtyping and value-uninhabitation systems are navigable. Hence, for all value-uninhabitation and subtyping judgments J, J is underivable iff there exists a failing derivation of J.

LEMMA 13. *Navigability.*
The subtyping system, including the value-uninhabitation subsystem, is navigable.

PROOF. The subtyping system is navigable because it has a rule function and a well-founded relation of premise to conclusion judgments. Figure 13 presents the rule function (from conclusion to premise judgments) that follows immediately from the subtyping rules (Figure 9). The relation of premise to conclusion judgments is well-founded: all the rules' premises decrease the sizes of the types under consideration, except that recursive types may be unrolled a limited number of times—the value-uninhabitation rules may unroll every recursive type at most once (with rule U-REC), and the subtyping rules may unroll every pair of recursive types at most once (with rule S-REC), but no rules ever introduce new recursive types. □

A.2. Correctness of the Value-Uninhabitation Rules

Lemma 14 shows that subtypes of value-uninhabited types must be value-uninhabited. In other words, the bottom type truly is bottom.

LEMMA 14. *Value-Uninhabitation is Closed Under Subtyping.*

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

PROOF. By assumption, $U \vdash \mathrm{val}(\tau_1) = \emptyset$ is underivable, so by Lemma 13 and the definition of failing derivations, there is a failing derivation rooted at $U \vdash \mathrm{val}(\tau_1) = \emptyset$. Proceed by induction on that failing derivation. In all cases, the contrapositive of Lemma 9 ensures that $\mathrm{val}(\tau_1) = \emptyset$ is underivable, so $\tau_1 \leq \tau_2$ can't be derived with rule S-$\bot$.

As can be seen in Figure 13, the failing derivation's leaf judgments (i.e., where f returns ε) can only be of the form $U \vdash \mathrm{val}(\tau_1) = \emptyset$ such that τ_1 is `nat`, `real`, or function type. In these cases the lemma holds because $\tau_1 \leq \tau_2$ can only be derived with rules S-BASE, S-NAT, S-REAL, S-$\top$, or S-FUN, implying that τ_2 must also be `nat`, `real`, or function type, so $\mathrm{val}(\tau_2) = \emptyset$ is underivable.

Figure 13 shows three possible forms of inner judgments in a failing derivation of $U \vdash \mathrm{val}(\tau_1) = \emptyset$. We next consider each of these three inductive cases. Note that judgments of the form $U \cup \{\tau_1\} \vdash \mathrm{val}(\tau_1) = \emptyset$ (third row from the bottom in Figure 13) can't be inner judgments in failing derivations because inner judgments must have exactly one child judgment from each of the conjunctive clauses returned by f. In general, no J for which $f(J)$ contains $()$ can be an inner (or leaf) judgment in a failing derivation.

— Case $\tau_1 = \tau_1' + \tau_1''$:
In this case the current judgment's child in the failing derivation is either an underivable $U \vdash \mathrm{val}(\tau_1') = \emptyset$ or an underivable $U \vdash \mathrm{val}(\tau_1'') = \emptyset$, so by the inductive hypothesis, the lemma holds on one of these judgments.
Because $\tau_1 = \tau_1' + \tau_1''$, $\tau_1 \leq \tau_2$ may be derived with rule S-$\top$ or S-SUM. In the S-$\top$ subcase, τ_2 is a function type, so $\mathrm{val}(\tau_2) = \emptyset$ is not derivable. In the S-SUM subcase, $\tau_2 = \tau_2' + \tau_2''$, $\tau_1' \leq \tau_2'$, and $\tau_1'' \leq \tau_2''$. Because $\tau_1' \leq \tau_2'$ and $\tau_1'' \leq \tau_2''$, the inductive hypothesis implies that $\mathrm{val}(\tau_2') = \emptyset$ is not derivable or $\mathrm{val}(\tau_2'') = \emptyset$ is not derivable, so by the definition of value-uninhabitation (U-SUM), $\mathrm{val}(\tau_2) = \emptyset$ is not derivable.

— Case $\tau_1 = \tau_1' \times \tau_1''$:
In this case the current judgment's children in the failing derivation are $U \vdash \mathrm{val}(\tau_1') = \emptyset$ and $U \vdash \mathrm{val}(\tau_1'') = \emptyset$, so by the inductive hypothesis, the lemma holds on both of these underivable judgments.
Because $\tau_1 = \tau_1' \times \tau_1''$, $\tau_1 \leq \tau_2$ may be derived with rule S-$\top$ or S-PROD. In the S-$\top$ subcase, τ_2 is a function type, so $\mathrm{val}(\tau_2) = \emptyset$ is not derivable. In the S-PROD subcase, $\tau_2 = \tau_2' \times \tau_2''$, $\tau_1' \leq \tau_2'$, and $\tau_1'' \leq \tau_2''$. Because $\tau_1' \leq \tau_2'$ and $\tau_1'' \leq \tau_2''$, the inductive hypothesis implies that $\mathrm{val}(\tau_2') = \emptyset$ is not derivable and $\mathrm{val}(\tau_2'') = \emptyset$ is not derivable, so by the definition of value-uninhabitation (U-PROD), $\mathrm{val}(\tau_2) = \emptyset$ is not derivable.

— Case $\tau_1 = \mu t_1.\overline{\tau}_1$ (with $\tau_1 \notin U$):
In this case the current judgment's only child in the failing derivation is $U \cup \{\tau_1\} \vdash \mathrm{val}(\tau_{1u}) = \emptyset$, where τ_{1u} is $[\mu t_1.\overline{\tau}_1 / t_1]\overline{\tau}_1$, so by the inductive hypothesis, the lemma holds on this underivable judgment.
Because $\tau_1 = \mu t_1.\overline{\tau}_1$, $\tau_1 \leq \tau_2$ may be derived with rule S-$\top$ or S-REC. In the S-$\top$ subcase, τ_2 is a function type, so $\mathrm{val}(\tau_2) = \emptyset$ is not derivable. In the S-REC subcase, $\tau_2 = \mu t_2.\overline{\tau}_2$, so let $\tau_{2u} = [\mu t_2.\overline{\tau}_2 / t_2]\overline{\tau}_2$; then because $\tau_1 \leq \tau_2$, Lemma 12 provides that $\tau_{1u} \leq \tau_{2u}$. Given that $\tau_{1u} \leq \tau_{2u}$, the inductive hypothesis implies that $\mathrm{val}(\tau_{2u}) = \emptyset$ is not derivable, so by the contrapositive of Lemma 11, $\mathrm{val}(\tau_2) = \emptyset$ is not derivable.

In all cases, $\mathrm{val}(\tau_2) = \emptyset$ is not derivable, as required. □

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

LEMMA 15. *Value-Uninhabitation.*

$$\forall\tau : (\mathrm{val}(\tau){=}\emptyset \;\Leftrightarrow\; \neg\exists v : (v{:}\tau))$$

PROOF. We first prove that, for all U and τ, if $U\vdash\mathrm{val}(\tau){=}\emptyset$ is not derivable, then $\exists v : (v{:}\tau)$. The contrapositive of the lemma's if-direction ($\Leftarrow$) follows as a result. The proof is by induction on the failing derivation of $U\vdash\mathrm{val}(\tau){=}\emptyset$, which can only have leaf judgments when τ is nat, real, or $\tau_1{\rightarrow}\tau_2$. In every one of these base cases, there exists a v such that $v{:}\tau$ (when τ is nat or real, let v be 0, and when $\tau = \tau_1{\rightarrow}\tau_2$, let v be $\lambda x : \tau_1.(d(\mathtt{roll}_{\mu t.(t\rightarrow\tau_2)}(d)))$, where d is $\lambda x{:}\mu t.(t{\rightarrow}\tau_2).(\mathtt{unroll}(x)\ x)$). The inductive cases occur when τ is a sum, product, or recursive type. In the case where $\tau{=}\mu t.\overline{\tau}$, the inductive hypothesis implies that there exists a v' such that $v'{:}[\mu t.\overline{\tau}/t]\overline{\tau}$; let $v{=}\mathtt{roll}_\tau(v')$ to ensure that $v{:}\tau$. The cases where τ is a sum or product type are handled similarly (but instead of v being a rolled subvalue, it's either an injection of a subvalue or a pair of subvalues).

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

— Cases T-NAT, T-REAL, T-LAM: Here τ is nat, real, or a function type, so $\mathrm{val}(\tau){=}\emptyset$ is not derivable.
— Case T-PROD: Here $\tau = \tau_1 \times \tau_2$, $v = (v_1, v_2)$, $v_1{:}\tau_1$, and $v_2{:}\tau_2$. By the inductive hypothesis, $\mathrm{val}(\tau_1){=}\emptyset$ is not derivable and $\mathrm{val}(\tau_2){=}\emptyset$ is not derivable, so by rule U-PROD, $\mathrm{val}(\tau_1 \times \tau_2){=}\emptyset$ is not derivable.
— Case T-LEFT: Here $\tau = \tau_1 + \tau_2$, $v = \mathtt{inl}_\tau(v_1)$, and $v_1{:}\tau_1$. By the inductive hypothesis, $\mathrm{val}(\tau_1){=}\emptyset$ is not derivable, so by rule U-SUM, $\mathrm{val}(\tau_1 + \tau_2){=}\emptyset$ is not derivable.
— Case T-RIGHT: This case is similar to the previous one.
— Case T-ROLL: Here $\tau = \mu t.\overline{\tau}$, $v = \mathtt{roll}_\tau(v')$, and $v'{:}[\mu t.\overline{\tau}/t]\overline{\tau}$. Let $\tau_u = [\mu t.\overline{\tau}/t]\overline{\tau}$, so we have $v'{:}\tau_u$. By the inductive hypothesis, $\mathrm{val}(\tau_u){=}\emptyset$ is not derivable, so by the contrapositive of Lemma 11, $\mathrm{val}(\tau){=}\emptyset$ is not derivable.
— Case T-SUBSUME: Here $v{:}\tau'$ and $\tau'{\leq}\tau$. By the inductive hypothesis, $\mathrm{val}(\tau'){=}\emptyset$ is not derivable, so by Lemma 14, $\mathrm{val}(\tau){=}\emptyset$ is not derivable.

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

A.3. Subtyping Reflexivity and Transitivity

Although the subtyping relation in λ_{ADT} lacks explicit reflexive and transitive rules, this section's lemmas show that the relation is nonetheless reflexive and transitive.

LEMMA 16. *Strong Subtyping Reflexivity.*

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

PROOF. By induction on the failing derivation of $S{\vdash}\tau_1{\leq}\tau_2$, which exists by Lemma 13. We first show that the lemma holds on any $S{\vdash}\tau_1{\leq}\tau_2$ judgment in a failing derivation such that this judgment doesn't have a child of the form $S'{\vdash}\tau_1'{\leq}\tau_2'$. These cases occur when $\tau_1{=}$real and $\tau_2{=}$nat, or when exactly one of τ_1 and τ_2 is a function/product/sum/recursive type (recall, from the proof of Lemma 14, that no J for which $() \in f(J)$ can appear in a failing derivation). In all these base cases, $\tau_1 \neq \tau_2$, as required.

The remaining cases of $S{\vdash}\tau_1{\leq}\tau_2$ judgments in failing derivations occur when both τ_1 and τ_2 are function/sum/product/recursive types. In all these inductive cases, there exists an underivable child judgment of the form $S' \vdash \tau_1' \leq \tau_2'$, so the inductive hypothesis implies that $\tau_1' \neq \tau_2'$, which guarantees that $\tau_1 \neq \tau_2$. For example, the inductive hypothesis in the case of recursive types implies that the unrolled types are unequal, which guarantees that the rolled types must also be unequal. $\square$

Lemma 17 provides a standard subtyping-inversion result, though the result is complicated by consideration of value-uninhabitation.

LEMMA 17. *Subtyping Inversion.*

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

A. $\mathrm{val}(\tau_1) = \emptyset$*, or*
B. $\mathrm{val}(\tau_1) = \emptyset$ *is underivable,* $\tau_2 = \tau_2'{\to}\tau_2''$*, and* $\mathrm{val}(\tau_2') = \emptyset$*, or*
C. Neither A nor B hold, and all of the following hold:
- *i.* $\tau_1{=}\mathtt{real} \Rightarrow (\tau_2{=}\mathtt{real})$
- *ii.* $\tau_1{=}\mathtt{nat} \Rightarrow (\tau_2{=}\mathtt{real} \vee \tau_2{=}\mathtt{nat})$
- *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.* $\tau_2{=}\mathtt{real} \Rightarrow (\tau_1{=}\mathtt{nat} \vee \tau_1{=}\mathtt{real})$
- *viii.* $\tau_2{=}\mathtt{nat} \Rightarrow (\tau_1{=}\mathtt{nat})$
- *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 $S \vdash \tau_1{\leq}\tau_2$. $\square$

LEMMA 18. *Strong Subtyping Transitivity.*

$$\forall S, \tau_1, \tau_2, \tau_3 : ((S{\vdash}\tau_1{\leq}\tau_3 \textit{ not derivable} \wedge \tau_1{\leq}\tau_2) \;\Rightarrow\; (\tau_2{\leq}\tau_3 \textit{ not derivable}))$$

PROOF. By induction on the failing derivation of $S{\vdash}\tau_1{\leq}\tau_3$. Note that because $S{\vdash}\tau_1{\leq}\tau_3$ is underivable, $\mathrm{val}(\tau_1){=}\emptyset$ is underivable (by rule S-$\bot$), so by Lemma 14, $\mathrm{val}(\tau_2){=}\emptyset$ is underivable. Also because $S{\vdash}\tau_1{\leq}\tau_3$ is underivable, if $\tau_3{=}\tau_3'{\to}\tau_3''$ then $\mathrm{val}(\tau_3'){=}\emptyset$ is underivable (by rule S-$\top$). Now suppose that $\tau_2{=}\tau_2'{\to}\tau_2''$ and $\mathrm{val}(\tau_2'){=}\emptyset$; then the only rules for deriving $\tau_2{\leq}\tau_3$ would be S-$\bot$, S-$\top$, and S-FUN; however, S-$\bot$ can't apply because $\mathrm{val}(\tau_2){=}\emptyset$ is underivable, S-$\top$ can't apply because if $\tau_3{=}\tau_3'{\to}\tau_3''$ then $\mathrm{val}(\tau_3'){=}\emptyset$ is underivable, and S-FUN can't apply because it would violate Lemma 14 to have $\tau_3'{\leq}\tau_2'$ and $\mathrm{val}(\tau_2'){=}\emptyset$ when $\mathrm{val}(\tau_3'){=}\emptyset$ is underivable. It's therefore impossible to derive $\tau_2{\leq}\tau_3$ when $\tau_2{=}\tau_2'{\to}\tau_2''$ and $\mathrm{val}(\tau_2'){=}\emptyset$.

We now have that (1) $\mathrm{val}(\tau_1){=}\emptyset$ is underivable, (2) $\mathrm{val}(\tau_2){=}\emptyset$ is underivable, (3) if $\tau_2{=}\tau_2'{\to}\tau_2''$ then $\mathrm{val}(\tau_2'){=}\emptyset$ is underivable, and (4) if $\tau_3{=}\tau_3'{\to}\tau_3''$ then $\mathrm{val}(\tau_3'){=}\emptyset$ is underivable. In other words, neither τ_1 nor τ_2 is a $\bot$, and neither τ_2 nor τ_3 is a $\top$. The cases below therefore ignore these possibilities.

We first show that the lemma holds on any $S{\vdash}\tau_1{\leq}\tau_3$ judgment in a failing derivation such that this judgment doesn't have a child of the form $S'{\vdash}\tau_1'{\leq}\tau_3'$. These cases occur when $\tau_1{=}\mathtt{real}$ and $\tau_3{=}\mathtt{nat}$, or when exactly one of τ_1 and τ_3 is a function/product/sum/recursive type. If $\tau_1{=}\mathtt{real}$ and $\tau_3{=}\mathtt{nat}$, then $\tau_2{\leq}\tau_3$ is underivable because Lemma 17 (applied to $\tau_1{\leq}\tau_2$) ensures that $\tau_2{=}\mathtt{real}$. If exactly one of τ_1 and τ_3 is a function/product/sum/recursive type, then $\tau_2{\leq}\tau_3$ is again underivable because otherwise, with $\tau_1{\leq}\tau_2$ and $\tau_2{\leq}\tau_3$, Lemma 17 would ensure that both τ_1 and τ_3 are the same "kind" of type (i.e., both numeric/function/product/sum/recursive types).

The remaining cases of $S \vdash \tau_1 \leq \tau_3$ judgments in failing derivations occur when both τ_1 and τ_3 are function/sum/product/recursive types. In the function-types case, $\tau_1 = \tau_1' \rightarrow \tau_1''$, $\tau_3 = \tau_3' \rightarrow \tau_3''$, and the underivable judgment $S \vdash \tau_1 \leq \tau_3$ either has the underivable $S \vdash \tau_3' \leq \tau_1'$ or the underivable $S \vdash \tau_1'' \leq \tau_3''$ as one of its children in the failing derivation. Because $\tau_1 \leq \tau_2$, Lemma 17 ensures that $\tau_2 = \tau_2' \rightarrow \tau_2''$, $\tau_2' \leq \tau_1'$, and $\tau_1'' \leq \tau_2''$. Hence, the inductive hypothesis, applied to $S \vdash \tau_3' \leq \tau_1'$ or $S \vdash \tau_1'' \leq \tau_3''$ (whichever is the child of $S \vdash \tau_1 \leq \tau_3$), implies that at least one of $\tau_3' \leq \tau_2'$ and $\tau_2'' \leq \tau_3''$ is underivable, so $\tau_2 \leq \tau_3$ is underivable (by rule S-FUN). The proofs of the sum- and product-types cases are similar.

In the recursive-types case, $\tau_1 = \mu t_1.\overline{\tau}_1$, $\tau_3 = \mu t_3.\overline{\tau}_3$, and the underivable judgment $S \vdash \tau_1 \leq \tau_3$ has the underivable $\{\tau_1 \leq \tau_3\} \vdash \tau_{1u} \leq \tau_{3u}$ as one of its children in the failing derivation (where $\tau_{1u} = [\tau_1/t_1]\overline{\tau}_1$ and $\tau_{3u} = [\tau_3/t_3]\overline{\tau}_3$). Because $\tau_1 \leq \tau_2$, Lemma 17 ensures that $\tau_2 = \mu t_2.\overline{\tau}_2$, and Lemma 12 ensures that $\tau_{1u} \leq \tau_{2u}$ (where $\tau_{2u} = [\tau_2/t_2]\overline{\tau}_2$). The inductive hypothesis then implies that $\tau_{2u} \leq \tau_{3u}$ is underivable, so by the contrapositive of Lemma 12, $\tau_2 \leq \tau_3$ is underivable. □

COROLLARY 19. *$\leq$ is a Preorder.*
The subtyping relation is reflexive and transitive.

PROOF. Immediate by the contrapositives of Lemmas 16 and 18. □

A.4. Properties of the Static and Dynamic Semantics

Having completed the "sanity checks" on the val and $\leq$ relations, Lemmas 20–22 present standard weakening, variable-substitution, and canonical-forms lemmas, which are used to prove subtyping completeness and soundness.

LEMMA 20. *Weakening.*

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

PROOF. By induction on the derivation of $\Gamma \vdash e : \tau$. □

LEMMA 21. *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 22. *Canonical Forms.*

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

A. $\tau{=}\mathtt{nat} \Rightarrow v{=}\mathtt{n}$ *(for some* $\mathtt{n}$*)*
B. $\tau{=}\mathtt{real} \Rightarrow v{=}\mathtt{n}$ *or* $v{=}\mathtt{r}$ *(for some* $\mathtt{n}$ *or* $\mathtt{r}$*)*
C. $(\tau{=}\tau_1{\rightarrow}\tau_2 \wedge \mathrm{val}(\tau_1){=}\emptyset$ *not derivable*$) \Rightarrow v{=}\lambda x{:}\tau_3.e$ *(for some* x, τ_3, *and* e*)*
D. $\tau{=}\tau_1{+}\tau_2 \Rightarrow v{=}\mathtt{inl}_{\tau_1'+\tau_2'}(v')$ *or* $v{=}\mathtt{inr}_{\tau_1'+\tau_2'}(v')$ *(for some* τ_1', τ_2', *and* v'*)*
E. $\tau{=}\tau_1 \times \tau_2 \Rightarrow v{=}(v_1, v_2)$ *(for some* v_1 *and* v_2*)*
F. $\tau{=}\mu t.\overline{\tau} \Rightarrow v{=}\mathtt{roll}_{\mu t.\overline{\tau}}(v')$ *(for some* t, $\overline{\tau}$, *and* v'*)*

PROOF. By induction on the derivation of $v{:}\tau$. The only nontrivial case is T-SUBSUME, in which $v{:}\tau'$ and $\tau' \leq \tau$. Because $v{:}\tau'$, Lemma 15 ensures that $\mathrm{val}(\tau') = \emptyset$ is underivable. If $\tau = \mathtt{nat}$ then by Lemma 17, $\tau' = \mathtt{nat}$, so by the inductive hypothesis (applied to $v{:}\tau'$), $v = \mathtt{n}$. If $\tau = \mathtt{real}$ then by Lemma 17, $\tau' = \mathtt{nat}$ or $\tau' = \mathtt{real}$, so by the inductive hypothesis, $v = \mathtt{n}$ or $v = \mathtt{r}$. If $\tau = \tau_1 \rightarrow \tau_2$ and $\mathrm{val}(\tau_1) = \emptyset$ is not derivable, then by Lemma 17, $\tau' = \tau_1' \rightarrow \tau_2'$ and $\tau_1 \leq \tau_1'$. Because $\mathrm{val}(\tau_1) = \emptyset$ is not derivable and $\tau_1 \leq \tau_1'$, Lemma 14 ensures that $\mathrm{val}(\tau_1') = \emptyset$ is not derivable. Then applying the inductive hypothesis to $v{:}\tau'$, where $\tau' = \tau_1' \rightarrow \tau_2'$, we find that $v = \lambda x{:}\tau_3.e$. The remaining cases of τ are proved similarly. □

A.5. Subtyping Completeness

We're now ready to state and prove the key lemma used to show completeness, Lemma 23. As in λ, we first prove a slightly stronger version of the desired completeness result. Also as in λ, the proof of Lemma 23 is constructive (in part because the proof of Lemma 15 is constructive).

LEMMA 23. *Strong Completeness.*

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

PROOF. The proof is by induction on the failing derivation of $S \vdash \tau_1 \leq \tau_2$. In all cases, the underivability of $S \vdash \tau_1 \leq \tau_2$ implies that τ_1 is not a $\bot$ (i.e., $\mathrm{val}(\tau_1) = \emptyset$ is underivable) and τ_2 is not a $\top$.

We first show that the lemma holds on any $S \vdash \tau_1 \leq \tau_2$ judgment in the failing derivation such that this judgment doesn't have a child of the form $S' \vdash \tau_1' \leq \tau_2'$. These cases 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.

Case $\tau_1 = \texttt{real}$ and $\tau_2 = \texttt{nat}$:
This case's proof is the same as in the proof of Lemma 7.

Case $\tau_1 = \tau_1' \to \tau_1''$ and $\tau_2 \neq \tau_2' \to \tau_2''$:
Construct a lambda value $v{:}\tau_1$ as shown in the proof of Lemma 15, and define E and τ as follows:

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

$$\tau = \begin{cases} \texttt{real} & \texttt{if}\ \tau_2 = \texttt{nat}\ \texttt{or}\ \tau_2 = \texttt{real}\ \texttt{or}\ \tau_2 = \tau_2' + \tau_2'' \\ \tau_2'' & \texttt{if}\ \tau_2 = \tau_2' \times \tau_2'' \\ [\mu t.\overline{\tau}/t]\,\overline{\tau} & \texttt{if}\ \tau_2 = \mu t.\overline{\tau} \end{cases}$$

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 is a lambda value).

Case $\tau_1 \neq \tau_1' \to \tau_1''$, $\tau_2 = \tau_2' \to \tau_2''$, and both $\mathrm{val}(\tau_1) = \emptyset$ and $\mathrm{val}(\tau_2') = \emptyset$ are underivable:
By Lemma 15 there exist v and v_2' such that $v : \tau_1$ and $v_2' : \tau_2'$. With $\tau_1 \neq \tau_1' \to \tau_1''$ and $v : \tau_1$, Lemma 22 implies that v can't be a lambda value. 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 can't be a lambda value).

Case $\tau_1 = \mu t_1.\overline{\tau}_1$, $\tau_2 \neq \mu t_2.\overline{\tau}_2$, $\mathrm{val}(\tau_1) = \emptyset$ is underivable, and if $\tau_2 = \tau_2' \to \tau_2''$ then $\mathrm{val}(\tau_2') = \emptyset$ is underivable:
By Lemma 15 there exists a v such that $v{:}\mu t_1.\overline{\tau}_1$. Hence, by Lemma 22, v is a rolled value. Also by Lemma 15, if $\tau_2 = \tau_2' \to \tau_2''$ then there exists a v_2' such that $v_2'{:}\tau_2'$. Now define E and τ as follows:

$$E = \begin{cases} \texttt{neg}([\,]) & \texttt{if}\ \tau_2 = \texttt{nat}\ \texttt{or}\ \tau_2 = \texttt{real} \\ \texttt{case}_{\texttt{real}}[\,]\ \texttt{of inl}\ x \Rightarrow 2.718\ \texttt{else inr}\ y \Rightarrow 2.718 & \texttt{if}\ \tau_2 = \tau_2' + \tau_2'' \\ [\,].\texttt{snd} & \texttt{if}\ \tau_2 = \tau_2' \times \tau_2'' \\ [\,](v_2') & \texttt{if}\ \tau_2 = \tau_2' \to \tau_2'' \end{cases}$$

$$\tau = \begin{cases} \texttt{real} & \texttt{if}\ \tau_2 = \texttt{nat}\ \texttt{or}\ \tau_2 = \texttt{real}\ \texttt{or}\ \tau_2 = \tau_2' + \tau_2'' \\ \tau_2'' & \texttt{if}\ \tau_2 = \tau_2' \times \tau_2''\ \texttt{or}\ \tau_2 = \tau_2' \to \tau_2'' \end{cases}$$

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 is a rolled value).

Case $\tau_1 \neq \mu t_1.\overline{\tau}_1$, $\tau_2 = \mu t_2.\overline{\tau}_2$, and $\mathrm{val}(\tau_1){=}\emptyset$ is underivable:
There are two subcases to consider: either (1) τ_1 is a $\top$ or (2) not. In subcase (1), let $v = 0$, so $v{:}\tau_1$ by T-SUBSUME and S-$\top$. In subcase (2), Lemma 15 guarantees a v such that $v{:}\tau_1$, and Lemma 22 guarantees that v isn't a rolled value. Hence, in all subcases, $v{:}\tau_1$ and v isn't a rolled value. Now 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 other cases of $S{\vdash}\tau_1{\leq}\tau_2$ judgments having no child of the form $S'{\vdash}\tau_1'{\leq}\tau_2'$—that is, the cases where exactly one of τ_1 and τ_2 is a product/sum type—are proved similarly.

The remaining cases of $S{\vdash}\tau_1{\leq}\tau_2$ judgments in failing derivations occur when both τ_1 and τ_2 are function/sum/product/recursive types.

Case $\tau_1 = \tau_1'{\rightarrow}\tau_1''$ and $\tau_2 = \tau_2'{\rightarrow}\tau_2''$:
This case's proof almost matches that given for Lemma 7. All the logic remains the same, with only two nontrivial differences: (1) in the first subcase in Lemma 7, we obtained a $v_1''{:}\tau_1''$ by Lemma 3, but here we replace this v_1'' with an $e_1''{:}\tau_1''$ (which exists because all types in λ_{ADT} are inhabited), and (2) in the second subcase in Lemma 7, we obtained a $v_2'{:}\tau_2'$ by Lemma 3, but here we obtain the same by Lemma 15 and the assumption that τ_2 isn't a $\top$ (i.e., $\mathrm{val}(\tau_2'){=}\emptyset$ is underivable).

Case $\tau_1 = \mu t_1.\overline{\tau}_1$ and $\tau_2 = \mu t_2.\overline{\tau}_2$:
In this case, the underivable judgment $S{\vdash}\tau_1{\leq}\tau_2$ has $S{\cup}\{\tau_1{\leq}\tau_2\}{\vdash}\tau_{1u}{\leq}\tau_{2u}$ as a child in the failing derivation, where $\tau_{1u} = [\mu t_1.\overline{\tau}_1/t_1]\overline{\tau}_1$ and $\tau_{2u} = [\mu t_2.\overline{\tau}_2/t_2]\overline{\tau}_2$. By the inductive hypothesis, there exist E', τ', v', and e' such that $E'[\tau_{2u}]{:}\tau'$, $v'{:}\tau_{1u}$, $E'[v'] \mapsto^* e'$, and $\texttt{stuck}(e')$. Let $v = \texttt{roll}_{\tau_1}(v')$, $E = E'[\texttt{unroll}([\,])]$, $\tau = \tau'$, and $e = e'$. Then by rule T-ROLL, $v{:}\tau_1$. Also, $E'[\tau_{2u}]{:}\tau'$ means that $\{x'{:}\tau_{2u}\} \vdash E'[x']{:}\tau'$, which implies by Lemma 20 that $\{x{:}\tau_2, x'{:}\tau_{2u}\} \vdash E'[x']{:}\tau'$; then because $\{x{:}\tau_2\} \vdash \texttt{unroll}(x) : \tau_{2u}$, Lemma 21 ensures that $\{x{:}\tau_2\} \vdash E'[\texttt{unroll}(x)]{:}\tau'$, which means that $E'[\texttt{unroll}([\tau_2])]{:}\tau'$. Hence, $E[\tau_2]{:}\tau$. Also, by the definitions of E and v, we have $E[v] = E'[\texttt{unroll}(\texttt{roll}_{\tau_1}(v'))]$, so $E[v] \mapsto E'[v']$, where $E'[v'] \mapsto^* e'$. Thus, because $e' = e$ and $\texttt{stuck}(e')$, $E[v] \mapsto^* e$, and $\texttt{stuck}(e)$.

The remaining inductive cases, in which both τ_1 and τ_2 are product/sum types, are proved similarly. The product-types case constructs v as a pair expression and uses a `fst` or `snd` expression to eliminate the pair in E. The sum-types case constructs v as an `inl` or `inr` expression and uses a `case` expression to eliminate the injection in E. $\square$

Having proved a stronger version of completeness in Lemma 23, the weaker version follows as a corollary.

COROLLARY 24. *Completeness.*

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

PROOF. By Lemma 23, 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 $\texttt{stuck}(e')$. The corollary is the contrapositive of this result. $\square$

A.6. Subtyping Soundness

With completeness proved, we move on to proving the soundness of the subtyping relation using type-safety lemmas. Lemmas 25–27 are used to prove Preservation (Lemma 28), while Lemma 29 is used to prove Progress (Lemma 30).

LEMMA 25. *Typing Inversion.*

A. $\Gamma \vdash \mathtt{n}{:}\tau \Rightarrow (\mathtt{nat} \leq \tau)$
B. $\Gamma \vdash \mathtt{r}{:}\tau \Rightarrow (\mathtt{real} \leq \tau)$
C. $\Gamma \vdash \mathtt{succ}(e){:}\tau \Rightarrow (\Gamma \vdash e{:}\mathtt{nat} \ \wedge \ \mathtt{nat} \leq \tau)$
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. $\Gamma \vdash \mathtt{neg}(e){:}\tau \Rightarrow (\Gamma \vdash e{:}\mathtt{real} \ \wedge \ \mathtt{real} \leq \tau)$
F. $\Gamma \vdash \lambda x{:}\tau_1.e{:}\tau \Rightarrow \exists \tau_2 : (\Gamma \cup \{x{:}\tau_1\} \vdash e{:}\tau_2 \ \wedge \ \tau_1 \rightarrow \tau_2 \leq \tau)$
G. $\Gamma \vdash e_1(e_2){:}\tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e_1{:}\tau_1 \rightarrow \tau_2 \ \wedge \ \Gamma \vdash e_2{:}\tau_1 \ \wedge \ \tau_2 \leq \tau)$
H. $\Gamma \vdash \mathtt{inl}_{\tau_1' + \tau_2'}(e){:}\tau \Rightarrow (\Gamma \vdash e{:}\tau_1' \ \wedge \ \tau_1' + \tau_2' \leq \tau)$
I. $\Gamma \vdash \mathtt{inr}_{\tau_1' + \tau_2'}(e){:}\tau \Rightarrow (\Gamma \vdash e{:}\tau_2' \ \wedge \ \tau_1' + \tau_2' \leq \tau)$
J. $\Gamma \vdash (\mathtt{case}_{\tau'} \ e_1 \ \mathtt{of} \ \mathtt{inl} \, x \Rightarrow e_2 \ \mathtt{else} \ \mathtt{inr} \, y \Rightarrow e_3){:}\tau \Rightarrow$
$\quad \exists \tau_1, \tau_2 : (\Gamma \vdash e_1{:}\tau_1 + \tau_2 \ \wedge \ \Gamma \cup \{x{:}\tau_1\} \vdash e_2{:}\tau' \ \wedge \ \Gamma \cup \{y{:}\tau_2\} \vdash e_3{:}\tau' \ \wedge \ \tau' \leq \tau)$
K. $\Gamma \vdash e.\mathtt{fst} : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e{:}\tau_1 \times \tau_2 \ \wedge \ \tau_1 \leq \tau)$
L. $\Gamma \vdash e.\mathtt{snd} : \tau \Rightarrow \exists \tau_1, \tau_2 : (\Gamma \vdash e{:}\tau_1 \times \tau_2 \ \wedge \ \tau_2 \leq \tau)$
M. $\Gamma \vdash \mathtt{roll}_{\mu t.\overline{\tau}}(e){:}\tau \Rightarrow (\Gamma \vdash e{:}[\mu t.\overline{\tau}/t]\overline{\tau} \ \wedge \ \mu t.\overline{\tau} \leq \tau)$
N. $\Gamma \vdash \mathtt{unroll}(e){:}\tau \Rightarrow \exists t, \overline{\tau} : (\Gamma \vdash e{:}\mu t.\overline{\tau} \ \wedge \ [\mu t.\overline{\tau}/t]\overline{\tau} \leq \tau)$
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 two rules could apply: T-SUBSUME (in which case the result follows from an inductive argument) and another rule (in which case the result is immediate). For example, $\Gamma \vdash e.\mathtt{fst}{:}\tau$ 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 Corollary 19 then, $\tau_1 \leq \tau$, as required. If $\Gamma \vdash e.\mathtt{fst}{:}\tau$ is derived with T-FST, we can assume $\Gamma \vdash e : \tau_1 \times \tau_2$ and $\tau = \tau_1$. By Corollary 19 then, $\tau_1 \leq \tau$, as required. All the other cases are proved similarly. □

LEMMA 26. *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-NEG 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.

$$\text{Case } \frac{\mathtt{n}' = \mathtt{n} + 1}{\mathtt{succ}(\mathtt{n}) \mapsto_\beta \mathtt{n}'} \ \beta\text{-SUCC}$$

Because $\mathtt{succ}(n){:}\tau$, Lemma 25 ensures that $\mathtt{nat} \leq \tau$, while rule T-NAT ensures that $\mathtt{n}'{:}\mathtt{nat}$. Hence, $\mathtt{n}'{:}\tau$ by rule T-SUBSUME.

$$\text{Case } \frac{}{(\lambda x{:}\tau_1.e_1)(v) \mapsto_\beta [v/x]e_1} \ \beta\text{-APP}$$

By Lemma 25 and the assumption that $(\lambda x{:}\tau_1.e_1)(v){:}\tau$, we have $(\lambda x{:}\tau_1.e_1){:}\tau_1' \rightarrow \tau_2'$, $v{:}\tau_1'$, and $\tau_2' \leq \tau$. By Lemma 25 again and the result that $(\lambda x{:}\tau_1.e_1){:}\tau_1' \rightarrow \tau_2'$, we also have $\{x{:}\tau_1\} \vdash e_1{:}\tau_2$ and $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$. Because $\{x{:}\tau_1\} \vdash e_1{:}\tau_2$, rule T-LAM implies that $(\lambda x{:}\tau_1.e_1){:}\tau_1 \rightarrow \tau_2$. Given that $(\lambda x{:}\tau_1.e_1){:}\tau_1 \rightarrow \tau_2$ and $v{:}\tau_1'$, Lemma 15 implies that both $\mathrm{val}(\tau_1 \rightarrow \tau_2) = \emptyset$ and $\mathrm{val}(\tau_1') = \emptyset$ are underivable, so we can use Lemma 17 on the fact that $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ 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 with $\{x{:}\tau_1\} \vdash e_1{:}\tau_2$, Lemma 21 implies $[v/x]e_1{:}\tau_2$. Finally, with $[v/x]e_1{:}\tau_2$ and $\tau_2 \leq \tau_2' \leq \tau$, we have $[v/x]e_1{:}\tau$ by T-SUBSUME.

$$\text{Case } \frac{}{\texttt{unroll}(\texttt{roll}_{\mu t.\overline{\tau}}(v)) \mapsto_\beta v}\ \beta\text{-UNROLL}$$

By Lemma 25 and the assumption that $\texttt{unroll}(\texttt{roll}_{\mu t.\overline{\tau}}(v)){:}\tau$, we have $\texttt{roll}_{\mu t.\overline{\tau}}(v){:}\mu t'.\overline{\tau}'$ and $[\mu t'.\overline{\tau}'/t']\overline{\tau}'{\leq}\tau$. Then by Lemma 25 again and the result that $\texttt{roll}_{\mu t.\overline{\tau}}(v){:}\mu t'.\overline{\tau}'$, we find $v{:}[\mu t.\overline{\tau}/t]\overline{\tau}$ and $\mu t.\overline{\tau}{\leq}\mu t'.\overline{\tau}'$. Because $\mu t.\overline{\tau}{\leq}\mu t'.\overline{\tau}'$, Lemma 12 implies that $[\mu t.\overline{\tau}/t]\overline{\tau}{\leq}[\mu t'.\overline{\tau}'/t']\overline{\tau}'$. Hence, we have $v{:}[\mu t.\overline{\tau}/t]\overline{\tau}$ and $[\mu t.\overline{\tau}/t]\overline{\tau}{\leq}[\mu t'.\overline{\tau}'/t']\overline{\tau}'{\leq}\tau$, so $v{:}\tau$ by rule T-SUBSUME. □

LEMMA 27. *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 25 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. □

LEMMA 28. *Preservation.*

$$\forall e, \tau, e' : ((e{:}\tau \;\wedge\; e \mapsto e') \;\Rightarrow\; e'{:}\tau)$$

PROOF. Only one rule derives $e \mapsto e'$, 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 27 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 26 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 21 to imply that $E[e_2]{:}\tau$. Hence, $e'{:}\tau$ as required. □

LEMMA 29. *Decomposition.*

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

PROOF. By induction on the derivation of $e{:}\tau$. The proof is a standard progress proof using the canonical-forms Lemma 22 (and Lemma 15 in the T-APP case, to ensure that Case C of Lemma 22 applies). □

LEMMA 30. *Progress.*

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

PROOF. By assumption, $e{:}\tau$, so Lemma 29 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 $e_1 \mapsto_\beta e_2$, the dynamic semantics ensures that $e \mapsto E[e_2]$. □

Preservation and Progress imply type safety.

LEMMA 31. *Type Safety.*

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

PROOF. By induction on the derivation of $e \mapsto^* e'$, using Progress and Preservation (Lemmas 30 and 28) in the usual way. □

As with λ, the soundness of the subtyping relation follows from the variable-substitution and type-safety results (Lemmas 21 and 31).

LEMMA 32. *Soundness.*

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

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

A.7. Subtyping Preciseness

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

THEOREM 33. *Preciseness.*
The $\leq$ relation is precise with respect to type safety. Formally, for all types τ_1 and τ_2:

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

PROOF. Immediate by Corollary 24 and Lemma 32. □