(A Quick Intro to)
A Technique for Proving Subtyping Completeness, with an Application to Iso-recursive Types

Jay Ligatti

Joint work with:
Jeremy Blackburn   Ivory Hernandez   Michael Nachtigal
Suppose you’re defining a subtyping relation for a type-safe PL

What should your basic goals be?
Subtyping Relation Goals

• Soundness

\[ \tau_1 \leq \tau_2 \Rightarrow \text{\(\tau_1\)-type terms can always safely stand in for \(\tau_2\)-type terms} \]

• Completeness

\[ \text{\(\tau_1\)-type terms can always safely stand in for \(\tau_2\)-type terms} \Rightarrow \tau_1 \leq \tau_2 \]
Subtyping Relation Goals

- Preciseness is a standard goal when defining $\leq$
  
  Idea: $\leq$ is as complete as possible without sacrificing soundness

$\forall \tau_1, \tau_2 : \tau_1 \leq \tau_2$

precise = sound and complete

$\tau_1 \leq \tau_2$ iff $\tau_1 = \tau_2$

- Trivially complete

- Trivially sound
Proving Preciseness

• **Soundness** of $\leq$ can be proved with standard type-safety proofs
  – An unsound definition of $\leq$ would break type safety
Proving Preciseness

• **Soundness** of $\leq$ can be proved with standard type-safety proofs
  – An unsound definition of $\leq$ would break type safety

• **Completeness** of $\leq$ can be proved with

Problem
To Fill in the Blank,

• Need to state completeness property formally
• Then hopefully we can figure out how to prove it
To Fill in the Blank,

• Need to state completeness property formally
• Then hopefully we can figure out how to prove it
• Actually, let’s try to state the preciseness property formally...
Preciseness

• Intuition:
  \( \tau_1 \leq \tau_2 \) iff any term of type \( \tau_2 \) could be replaced by any term of type \( \tau_1 \) without breaking type safety
Preciseness

• Intuition:
  \( \tau_1 \leq \tau_2 \) iff any term of type \( \tau_2 \) could be replaced by any term of type \( \tau_1 \) without breaking type safety

• In other words:
  \( \tau_1 \leq \tau_2 \) iff \( \tau_2 \)-type expressions can—in any context—be replaced by \( \tau_1 \)-type expressions without causing well-typed programs to “get stuck”
Preciseness

\( \tau_1 \leq \tau_2 \) iff \( \tau_2 \)-type expressions can—in any context—be replaced by \( \tau_1 \)-type expressions without causing well-typed programs to “get stuck”

Definition: A subtyping relation \( \leq \) is precise wrt type safety when for all \( \tau_1, \tau_2 \):

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

Filling evaluation context \( E \)’s hole with a \( \tau_2 \)-type expression produces a well-typed program
Soundness

\( \tau_1 \leq \tau_2 \) iff \( \tau_2 \)-type expressions can—in any context—be replaced by \( \tau_1 \)-type expressions without causing well-typed programs to “get stuck”

Definition: A subtyping relation \( \leq \) is **sound** wrt type safety when for all \( \tau_1, \tau_2 \):

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

Filling evaluation context \( E \)'s hole with a \( \tau_2 \)-type expression produces a well-typed program
Completeness

$\tau_1 \leq \tau_2$ iff $\tau_2$-type expressions can—in any context—be replaced by $\tau_1$-type expressions without causing well-typed programs to “get stuck”

Definition: A subtyping relation $\leq$ is complete wrt type safety when for all $\tau_1, \tau_2$:

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

Filling evaluation context $E$’s hole with a $\tau_2$-type expression produces a well-typed program
Soundness of $\leq$ is a Corollary of Type Safety

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

• Proof idea:

Assume $\tau_1 \leq \tau_2$, $E[\tau_2]:\tau$, $e:\tau_1$, $E[e] \rightarrow ^* e'$, and stuck$(e')$

By subsumption and the definition of well-typed contexts, $E[e]:\tau$

But $E[e]:\tau$, $E[e] \rightarrow ^* e'$, and stuck$(e')$ combine to contradict type safety
Proving Completeness

\[ \neg \exists e, E, \tau, e': \]
\[ E[\tau_2]:\tau \land e:\tau_1 \land E[e] \rightarrow *e' \land \text{stuck}(e') \Rightarrow \tau_1 \leq \tau_2 \]
Proving Completeness

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

hmm…
Proving Completeness

(contrapositive)

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

- Approach: Define the subtyping relation in an algorithmic deductive system
  - i.e., the inference rules are deterministic, and all “attempted” derivations of \( \tau_1 \leq \tau_2 \) succeed/fail at a finite height
Proving Completeness

(contrapositive)

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

• Approach: Define the subtyping relation in an algorithmic deductive system
  – i.e., the inference rules are deterministic, and all “attempted” derivations of \(\tau_1 \leq \tau_2\) succeed/fail at a finite height

• Hence, because \(\tau_1 \not\leq \tau_2\), there exists a unique, finite “failing derivation” of \(\tau_1 \leq \tau_2\)
Example Failing Derivation

\[
\begin{align*}
\text{real} \leq \text{real} & \quad \checkmark \\
\text{int} \leq \text{real} & \quad \checkmark \\
\text{real} \rightarrow \text{int} \leq \text{real} \rightarrow \text{real} & \quad \checkmark \\
\text{int} \rightarrow \text{real} & \quad \times \\
\text{real} \rightarrow \text{int} \rightarrow \text{real} & \quad \times \\
\text{real} \rightarrow \text{real} \rightarrow \text{real} & \quad \times
\end{align*}
\]

\[(\text{real} \rightarrow \text{real}) \rightarrow (\text{int} \rightarrow \text{real}) \leq (\text{real} \rightarrow \text{int}) \rightarrow (\text{real} \rightarrow \text{real})\]

Types \( \tau ::= \text{int} \mid \text{real} \mid \tau_1 \rightarrow \tau_2 \)

\[
\begin{align*}
\tau_1 \leq \tau_2 \\
\text{int} \leq \text{int} \\
\text{real} \leq \text{real} \\
\text{int} \leq \text{real} \\
\tau_3 \leq \tau_1 & \quad \tau_2 \leq \tau_4 \\
\tau_1 \rightarrow \tau_2 & \leq \tau_3 \rightarrow \tau_4
\end{align*}
\]
Proving Completeness

$\tau_1 \nleq \tau_2 \Rightarrow \exists e, E, \tau, e': E[\tau_2]:\tau \land e:\tau_1 \land E[e] \rightarrow *e' \land stuck(e')$

By induction on the unique, finite, failing derivation of $\tau_1 \leq \tau_2$
Proving Completeness

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

We’ll trace the failure from a leaf to the root of the failing derivation tree, showing that completeness holds on each failing judgment along the way.

\[
\begin{array}{ccc}
\text{real} \leq \text{real} & \text{int} \leq \text{real} & \text{real} \leq \text{int}^* \text{real} \leq \text{real} \\
\text{real}\rightarrow\text{int} \leq \text{real}\rightarrow\text{real} & \text{int}\rightarrow\text{real} \leq \text{real}\rightarrow\text{real} & \\
(\text{real}\rightarrow\text{real})\rightarrow(\text{int}\rightarrow\text{real}) \leq (\text{real}\rightarrow\text{int})\rightarrow(\text{real}\rightarrow\text{real})
\end{array}
\]
Proving Completeness

\[ \tau_1 \not\preceq \tau_2 \Rightarrow \exists e, E, \tau, e' : \\
E[\tau_2] :\tau \land e : \tau_1 \land E[e] \rightarrow *e' \land \text{stuck}(e') \]

We’ll trace the failure from a leaf to the root of the failing derivation tree, showing that completeness holds on each failing judgment along the way.

\[
\begin{align*}
\text{real} \leq \text{real} & \quad \text{int} \leq \text{real} & \quad \text{real} \leq \text{int} \\
\text{real} \rightarrow \text{int} \leq \text{real} \rightarrow \text{real} & \quad \text{int} \rightarrow \text{real} \leq \text{real} \rightarrow \text{real} & \quad \text{real} \rightarrow \text{int} \\
(\text{real} \rightarrow \text{real}) \rightarrow (\text{int} \rightarrow \text{real}) \leq (\text{real} \rightarrow \text{int}) \rightarrow (\text{real} \rightarrow \text{real})
\end{align*}
\]
Proving Completeness

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

We’ll trace the failure from a leaf to the root of the failing derivation tree, showing that completeness holds on each failing judgment along the way

\[
\begin{align*}
\text{real} \leq \text{real} & \quad \text{int} \leq \text{real} & \quad \text{real} \leq \text{int} & \quad \text{real} \leq \text{real} \\
\text{real} \rightarrow \text{int} \leq \text{real} \rightarrow \text{real} & \quad \text{int} \rightarrow \text{real} \leq \text{real} \rightarrow \text{real} & \quad \text{real} \rightarrow \text{int} \rightarrow \text{real} \rightarrow \text{real}
\end{align*}
\]

\[
\text{(real} \rightarrow \text{real}) \rightarrow (\text{int} \rightarrow \text{real}) \leq (\text{real} \rightarrow \text{int}) \rightarrow (\text{real} \rightarrow \text{real})
\]
Base Cases of Completeness Proof

• 5 possible failing leaf judgments here:
  1. real ≤ int
  2. real ≤ τ₃ → τ₄
  3. int ≤ τ₃ → τ₄
  4. τ₃ → τ₄ ≤ real
  5. τ₃ → τ₄ ≤ int
Base Cases of Completeness Proof

• 5 possible failing leaf judgments here:
  1. real ≤ int
  2. real ≤ τ\_3 → τ\_4
  3. int ≤ τ\_3 → τ\_4
  4. τ\_3 → τ\_4 ≤ real
  5. τ\_3 → τ\_4 ≤ int

• In every case, an e, E, τ, e’ can be constructed such that E[τ\_2]:τ, e:τ\_1, E[e]→*e’, and stuck(e’


Inductive Step of Completeness Proof

• One case here: \( \frac{\tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4} \)

• Assuming the completeness property holds on some failing premise, prove that it also holds on the failing conclusion
Inductive Step of Completeness Proof

• One case here: \[ \frac{\tau_3 \leq \tau_1}{\tau_1 \rightarrow \tau_2} \leq \frac{\tau_2 \leq \tau_4}{\tau_3 \rightarrow \tau_4} \]

• Assuming the completeness property holds on some failing premise, prove that it also holds on the failing conclusion

• Again, it can be done; please see tech report for details
Another Interesting Problem

• Let’s apply these techniques (for proving subtyping preciseness) to the problem of subtyping iso-recursive types
Quick Refresher on Recursive Types

- Are fundamental for typing aggregate data structures

- Heavily used in functional and object-oriented PLs
  - datatype list = Empty of unit | Node of int * list
  - class Integer {... public void add(Integer i) ...}
Quick Refresher on Recursive Types

• There are 2 primary varieties of recursive types:

  – **Iso-recursive** systems require programmers to manually roll & unroll the recursion
    • ML and Haskell support iso-recursive types

  – **Equi-recursive** systems rely on type checkers to roll and unroll as needed, so programmers don’t have to
    • Modula-3 supports equi-recursive types
Amber Rules [Cardelli, 1986]

- Standard, textbook rules for subtyping iso-recursive types
- These rules are elegant and sound

\[
S \cup \{ t_1 \leq t_2 \} \vdash \tau_1 \leq \tau_2
\]

\[
S \vdash \mu t_1.\tau_1 \leq \mu t_2.\tau_2
\]

\[
S \cup \{ t_1 \leq t_2 \} \vdash t_1 \leq t_2
\]
Incompleteness of the Amber Rules

• Define:

\[ \tau_1 \equiv \mu L.\{\text{add}:(\mu i.\{\text{add}:i\to\text{unit}\})\to\text{unit}, \text{min}:\text{unit}\to\text{int}\} \]

\[ \tau_2 \equiv \mu i'.\{\text{add}:i'\to\text{unit}\} \]
Incompleteness of the Amber Rules

• Define:
  \[ \tau_1 \equiv \mu L.\{\text{add}: (\mu i.\{\text{add}: i \rightarrow \text{unit}\}) \rightarrow \text{unit}, \text{min}: \text{unit} \rightarrow \text{int} \} \]
  \[ \tau_2 \equiv \mu i'.\{\text{add}: i' \rightarrow \text{unit}\} \]

• \(\tau_1\) and \(\tau_2\) are natural encodings of class types

```java
class GreatInteger extends Integer {
    ...
    public void add(Integer i) {...}
    public int min() {...}
    ...
}
```

```java
class Integer {
    ...
    public void add(Integer i) {...}
    ...
}
```
Incompleteness of the Amber Rules

• Define:
  \[ \tau_1 \equiv \mu L.\{\text{add:}(\mu i.\{\text{add:i\rightarrow}\text{unit}\})\rightarrow\text{unit}, \text{min:}\text{unit}\rightarrow\text{int}\} \]
  \[ \tau_2 \equiv \mu i'.\{\text{add:i'\rightarrow}\text{unit}\} \]

• \( \tau_1 \) and \( \tau_2 \) are natural encodings of class types

  class GreatInteger extends Integer {
    ...
    public void add(Integer i) {...}
    public int min() {...}
    ...
  }

  class Integer {
    ...
    public void add(Integer i) {...}
    ...
  }

• GreatInteger \((\tau_1)\) is a subclass of Integer \((\tau_2)\) 
  \( \Rightarrow \) we should be able to derive \( \tau_1 \leq \tau_2 \)
Incompleteness of the Amber Rules

\[
\begin{align*}
\{L \leq i'\} & \vdash i' \leq \mu i.\{\text{add}:i \rightarrow \text{unit}\} \\
\{L \leq i'\} & \vdash \text{unit} \leq \text{unit} \\
\{L \leq i'\} & \vdash (\mu i.\{\text{add}:i \rightarrow \text{unit}\}) \rightarrow \text{unit} \leq i' \rightarrow \text{unit} \\
\{L \leq i'\} & \vdash \{\text{add}:\mu i.\{\text{add}:i \rightarrow \text{unit}\} \rightarrow \text{unit}, \text{min}:\text{unit} \rightarrow \text{int}\} \leq \{\text{add}:i' \rightarrow \text{unit}\} \\
\emptyset & \vdash \mu L.\{\text{add}:\mu i.\{\text{add}:i \rightarrow \text{unit}\} \rightarrow \text{unit}, \text{min}:\text{unit} \rightarrow \text{int}\} \leq \mu i'.\{\text{add}:i' \rightarrow \text{unit}\}
\end{align*}
\]
Incompleteness of the Amber Rules

\[
\begin{align*}
\{ L \leq i' \} & \vdash i' \leq \mu i.\{ \text{add}:i \to \text{unit} \} \quad \text{✓} \\
{ L \leq i' \} & \vdash \mu i.\{ \text{add}:i \to \text{unit} \} \to \text{unit} \leq i' \to \text{unit} \\
{ L \leq i' \} & \vdash \{ \text{add}:(\mu i.\{ \text{add}:i \to \text{unit} \}) \to \text{unit} , \text{min}:\text{unit} \to \text{int} \} \leq \{ \text{add}:i' \to \text{unit} \} \\
\emptyset & \vdash \mu L.\{ \text{add}:(\mu i.\{ \text{add}:i \to \text{unit} \}) \to \text{unit} , \text{min}:\text{unit} \to \text{int} \} \leq \mu i'.\{ \text{add}:i' \to \text{unit} \}
\end{align*}
\]

Problem: Amber rules don’t unroll recursive types in their premises, so their conclusions aren’t based on how iso-recursive types actually get used (i.e., eliminated).
New Iso-recursive Subtyping Rules

\[ \mu t_1.\tau_1 \leq \mu t_1.\tau_1 \notin S \]
\[ S \cup \{ \mu t_1.\tau_1 \leq \mu t_2.\tau_2 \} \vdash [\mu t_1.\tau_1/t_1]\tau_1 \leq [\mu t_2.\tau_2/t_2]\tau_2 \]

\[ S \vdash \mu t_1.\tau_1 \leq \mu t_2.\tau_2 \]

\[ S \cup \{ \mu t_1.\tau_1 \leq \mu t_2.\tau_2 \} \vdash \mu t_1.\tau_1 \leq \mu t_2.\tau_2 \]
New Rules Enable Desired Derivation

{L ≤ I', I' ≤ I, I ≤ I'} ⊢ I' ≤ I

{L ≤ I', I' ≤ I, I ≤ I'} ⊢ unit ≤ unit

{L ≤ I', I' ≤ I, I ≤ I'} ⊢ I → unit ≤ I' → unit

{L ≤ I', I' ≤ I, I ≤ I'} ⊢ {add:I → unit} ≤ {add:I' → unit}

{L ≤ I', I' ≤ I, I ≤ I'} ⊢ I ≤ I'

{L ≤ I', I' ≤ I} ⊢ unit ≤ unit

{L ≤ I', I' ≤ I} ⊢ I' → unit ≤ I → unit

{L ≤ I', I' ≤ I} ⊢ {add:I' → unit} ≤ {add:I → unit}

{L ≤ I'} ⊢ I' ≤ I

{L ≤ I'} ⊢ unit ≤ I' → unit

{L ≤ I'} ⊢ {add:I → unit, min:unit → int} ≤ {add:I' → unit}

∅ ⊢ μL.{add:(μi.{add:i → unit}) → unit, min:unit → int} ≤ μI'.{add:i' → unit}
New Rules are Precise

• Proof uses the techniques described earlier

• Proof also shows that the standard subtyping rules for function and (binary) sum and product types are precise as well
More Information

• Technical report:
  “Completely Subtyping Iso-recursive Types”

• Project webpage:
  http://www.cse.usf.edu/~ligatti/projects/completeness/