IV. Subtyping in type theory

- Compare with set theory:
  \[ a \in A \quad a : A \]
  \[ A \subseteq B \quad A \leq B \]

  But, this is superficial, because typing and subtyping can only be more restrictive.

- Traditional notion: "subsumptive subtyping"
  \[ a : A \quad A \leq B \]
  \[ \implies \quad a : B \]

Subtyping: motivations

- TTs in programming languages, eg,
  - More concise and readable programs (eg, subtype polymorphism in OO-languages)

- TTs for proof assistants, eg,
  - Abbreviations (eg, more readable terms/scripts)

- TTs as modelling languages, eg,
  - More powerful modelling languages

Subtyping: basics

- Fundamental principle
  If \( A \leq B \) and, wherever a term of type \( B \) is required, we can use a term of type \( A \) instead.

  For example, the subsumption rule realises this.

- Basic laws
  - Reflexivity: \( A \leq A \)
  - Transitivity: \( A \leq B \quad B \leq C \implies A \leq C \)

Subtyping: examples (informal)

- Base examples
  - \( \text{Nat} \leq \text{Int} \)
  - \( \text{Book} \leq \text{Phy} \), where \( \text{Phy} \) is the type of physical objects

- Examples for type constructors
  - \( A \leq A' \quad B \leq B' \implies A \times B \leq A' \times B' \)
  - \( A' \leq A \quad B \leq B' \implies A \to B \leq A' \to B' \) (contravariance)
  - \( A \leq B \implies \text{List}(A) \leq \text{List}(B) \)
  - \( \langle l_1 : A_1, \ldots, l_n : A_n \rangle \leq \langle l_1 : A_{1'}, \ldots, l_{n-1} : A_{n-1}, A_n \rangle \) (record types)

Two different views of types

- Question: Is subsumptive subtyping adequate for type theories with canonical objects?

- Answer: No: why and then what?
Two corresponding views of subtyping

Subsumptive subtyping is suitable for type assignment:
- A term can be overloaded (has more than one type).
- Subsumption is simply another rule for type assignment.

What about TTs with canonical objects?

Incompatibility of subsumption & canonicity

Subsumption rule:
\[ \text{a : A} \quad A \leq B \]
\[ \text{a : B} \]

Incompatible with the view of canonical objects

Q: If \( A \leq B \) and \( \text{a : A} \) is canonical in \( A \), is it canonical in \( B \)?

Canonicity

Definition
- Any closed object of an inductive type is computationally equal to a canonical object of that type.
- This is a basis of TTs with canonical objects.
- This is why the elimination rule is adequate.
- Eg, Elimination rule for List(T):
  "For any family C, if C is inhabited for all canonical T-lists \( \text{nil}(T) \) and \( \text{cons}(T,a,l) \), then so is \( C \) for all T-lists."

Canonicity is lost in subsumptive subtyping.

Eg,
\[ A \leq B \]
\[ \text{List}(A) \leq \text{List}(B) \]

\( \text{nil}(A) : \text{List}(B) \), by subsumption;
- But \( \text{nil}(A) \neq \text{any canonical B-list nil}(B) \) or \( \text{cons}(B,b,l) \).
- The elim rule for \( \text{List}(B) \) is inadequate: it does not cover \( \text{nil}(A) \) ...

Amending the elimination rule?

Generalise it to cover all subtypes ...
- To take care of the objects introduced by subsumptive subtyping.

But
- This requires "bounded quantification" to quantify over all subtypes (of the form \( \forall A \leq B \) ...)
- Troublesome ...
- If not, then what?
Coercive subtyping

- Basic idea
  - $A \subseteq B$ if there is a coercion $c$ from $A$ to $B$.
  - Coercions are "implicit" -- they can be omitted!

Subtyping as abbreviation

Coercive subtyping: formal rules

- Formal presentation (Luo 1997/1999) includes

$$f : B \rightarrow D \quad a : A \quad A \subseteq B$$

$$f(a) = f(c(a)) : D$$

- Coercions between any two types are unique.  
  - Think of an implementation: if more than one, the computer does not know which to choose ...  
  - Incoherence leads to non-conservativity (and in most cases, inconsistency).

Formal defn of coherence:

$$A \lessdot c B$$

where $\lessdot$ is the computational equality.

Coherence

- Coherence: a key requirement
  - Coercions between any two types are unique.  
  - Think of an implementation: if more than one, the computer does not know which to choose ...  
  - Incoherence leads to non-conservativity (and in most cases, inconsistency).

- Formal defn of coherence:

$$A \lessdot B \quad A \lessdot B$$

$$c = c' : A \rightarrow B$$

where $\lessdot$ is the computational equality.

Coercive subtyping generalises/subsumes

- Injective/subset subtyping: Even $\leq$ N; Man $\leq$ Human.
- Projective/inheritance subtyping: $\Sigma$ (Man, handsome) $\leq$ Man.

Applications of coercive subtyping, including

- Proof development
- Dependently-typed programming
- Type-theoretical semantics in linguistics

Structural subtyping in coercive subtyping

- Example -- structural subtyping for lists:

$$A \subseteq B$$

$$\text{List}(A) \subseteq \text{map}(c) \text{ List}(B)$$

- Structural subtyping for all inductive types
  - $\Sigma$-types, types of vectors, ...
  - General rules and transitivity elimination
    [Luo & Adams 08, Luo & Luo 05]
Non-structural subtyping – examples

- Projective subtyping (c.f. record subtyping)
  - From Σ-types or record types to component types
  - Projections of (dependent) record types
  - Very useful in proofs/modelling
  - Eg, first projection as a coercion:
    \[ \sum (\text{Nat, positive}) \leq \pi_1 \text{Nat} \]

Projections of (dependent) record types

- Very useful in proofs/modelling
- Eg, proof development [Bailey 1998, ...]
- Type-theoretic model of linguistic semantics [Luo 2010]

Coercion concerning unit types

- Useful in various applications
- Eg., representation of manifest fields in module types (Σ-types or dependent record types) [Luo 08]

Coercive subtyping is an adequate theory of subtyping for type theories with canonical objects.

Views on types

<table>
<thead>
<tr>
<th>Views on types</th>
<th>Views on subtyping</th>
</tr>
</thead>
<tbody>
<tr>
<td>type assignment systems</td>
<td>subsumptive subtyping</td>
</tr>
<tr>
<td>TTs with canonical objects</td>
<td>coercive subtyping</td>
</tr>
</tbody>
</table>

Historical remarks

- Semantic interpretations for subtyping in PLs
  - Early papers: eg, Mitchell (1983/1991) for the simply typed λ-calculus
  - Both subsumptive and coercive interpretations, called subset interpretation and coercion interpretation, resp.
  - Later papers, eg, (Breazu-Tannen et al 1991) for recursive & record types in PLs

Subsumptive subtyping for dependent types

- Subtyping for Edinburgh LF (Aspinall & Compagnoni 2001)
- Subtyping for Edinburgh LF (Aspinall & Compagnoni 2001)
- Remarks on our previous treatment in coercive subtyping
  - Proof-theoretic
  - For T Ts with canonical objects

Revision Questions

- How can subtyping be useful in
  - Programming languages?
  - Proof assistants?
- Modelling?
- What are two different views of types/subtyping?
  - What is subsumptive subtyping?
  - What is coercive subtyping?
  - How are they related to the views of types?

Selected References