Subtyping

Stefan Muller

CS 534: Types and Programming Languages, Spring 2024
Lecture 19

1 Subsumption

Consider the types \texttt{nat} and \texttt{int}. We want to be able to consider anything of \texttt{nat} to be of type \texttt{int}. We'll say \texttt{nat} is a "subtype" of \texttt{int}:

\[
\texttt{nat} <: \texttt{int}
\]

In general, we'll use \( \tau <: \tau' \) to say that \( \tau \) is a subtype of \( \tau' \). There are a few ways of interpreting this:

- Any \( \tau \) can behave like a \( \tau' \).
- Anything that expects a \( \tau' \) can accept a \( \tau \).
- Anything of type \( \tau \) can have type \( \tau' \).

The last point is explicitly allowed by the "subsumption rule":

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \quad \text{(T-SUB)}
\]

2 Example: Products

Consider \( n \)-ary products \( \tau_1 \times \cdots \times \tau_n \):

\[
\forall i, \Gamma \vdash e_i : \tau_i \quad (\times-I)
\]

\[
\frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \quad 1 \leq i \leq n}{\Gamma \vdash \pi_i e : \tau_i} \quad (\times-E)
\]

Let:

\[
\text{fst2} \quad \equiv \quad \lambda x : \text{int} \times \text{int}. \pi_1 x
\]

\[
\text{fst3} \quad \equiv \quad \lambda x : \text{int} \times \text{int} \times \text{int}. \pi_1 x
\]

The expression \( \text{fst2} (1, 2, 3) \) is perfectly safe but not well-typed. In general, \( \pi_i e \) is safe for \( e : \tau_1 \times \cdots \times \tau_n \) as long as \( i \leq n \). Since projection is the only thing that can "expect" (eliminate) something of product type, that should mean that

\[
\text{int} \times \text{int} \times \text{int} \quad <: \quad \text{int} \times \text{int}
\]

and in general

\[
\frac{\tau_1 \times \cdots \times \tau_n \times \cdots \times \tau_{n+k} \quad \text{SUB-WIDTH}}{\tau_1 \times \cdots \times \tau_n \times \cdots \times \tau_{n+k} \quad \text{SUB-WIDTH}}
\]

We call this "width subtyping".

With this and the subsumption rule, we can type \( \text{fst2} (1, 2, 3) \):
Note that it would not be safe to say
\[ \text{int} \times \text{int} <: \text{int} \times \text{int} \times \text{int} \]
because we cannot allow something like \( \pi_3 (1, 2) \).

What about
\[ (\text{int} \times \text{int} \times \text{int}) \times \text{int} <: (\text{int} \times \text{int}) \times \text{int} \]
\[ \text{int} \times \text{nat} <: \text{int} \times \text{int} \]

These should be fine, but we don’t have the rules to show them. Enter “depth subtyping”:

\[ \forall i, \tau \downarrow \in [1, n]. \tau_i <: \tau'_i \]
\[ \tau_1 \times \cdots \times \tau_n <: \tau'_1 \times \cdots \times \tau'_n \]

(SUB-DEPTH)

3 Properties of Subtyping

Note that to derive both of the subtyping relations above, we still need \( \text{int} <: \text{int} \). In general, we require that subtyping is reflexive and transitive. There are two ways to do this:

3.1 Set up the rules carefully so we can prove it.

We need a rule for products that combines width and depth subtyping.

\[ \text{unit} <: \text{unit} \text{ (Sub-Unit)} \]
\[ \text{int} <: \text{int} \text{ (Sub-Int)} \]
\[ \forall i \in [1, n]. \tau_i <: \tau'_i \]
\[ \tau_1 \times \cdots \times \tau_n <: \tau'_1 \times \cdots \times \tau'_n \text{ (Sub-Prod)} \]

Lemma 1. For all \( \tau, \tau <: \tau \).

Proof. By induction on the structure of \( \tau \).

- unit, int. By Sub-Unit, Sub-Int.
- \( \tau_1 \times \cdots \times \tau_n \). By induction, \( \tau_i <: \tau_i \). Apply Sub-Prod.

\[ \square \]

Lemma 2. If \( \tau_1 <: \tau_2 \) and \( \tau_2 <: \tau_3 \), then \( \tau_1 <: \tau_3 \).

Proof. By induction on the derivation of \( \tau_1 <: \tau_2 \) and \( \tau_2 <: \tau_3 \)

- If the first derivation is by Sub-Unit or Sub-Int, then \( \tau_1 = \tau_2 \), so we have \( \tau_1 <: \tau_3 \) by assumption.
- If the second derivation is by Sub-Unit or Sub-Int, then \( \tau_2 = \tau_3 \), so we have \( \tau_1 <: \tau_3 \) by assumption.
- Sub-Prod, Sub-Prod. We have

\[ \tau_1 \times \cdots \times \tau_{n+k} <: \tau'_1 \times \cdots \times \tau'_{n+k} \times \tau''_n \times \cdots \times \tau''_n \]

where \( \tau_i <: \tau'_i <: \tau''_i \). Apply Sub-Prod

\[ \square \]

This gets a lot harder as we add more rules.
3.2 Add explicit rules for reflexivity and transitivity

\[
\begin{align*}
\tau \lll \tau &\quad \text{(Sub-Refl)} \quad \frac{\tau_1 \lll \tau_2, \tau_2 \lll \tau_3}{\tau_1 \lll \tau_3} \quad \text{(Sub-Trans)}
\end{align*}
\]

This generally makes the theory cleaner and easier, but is a huge pain to actually implement in a type checker.

4 Other Subtyping Rules

What about n-ary sums?

\[
\begin{align*}
\tau_1 + \cdots + \tau_n \lll \tau_1 + \cdots + \tau_n + \cdots + \tau_{n+k} &\quad \text{(Sub-Width-Sum)} \\
\tau_1 + \cdots + \tau_n \lll \tau_1' + \cdots + \tau_n' &\quad \text{(Sub-Depth-Sum)}
\end{align*}
\]

How about functions?

\[
\frac{\tau_1' \lll \tau_1, \tau_2' \lll \tau_2}{\tau_1 \rightarrow \tau_2 \lll \tau_1' \rightarrow \tau_2'} \quad \text{(Sub-Fun)}
\]

Covariance and Contravariance Note that the argument types don’t go the way you expect! Rules Sub-Depth and Sub-Depth-Sum are covariant: the subtyping relations of the types go in the same direction as their components. Function subtyping is covariant in the result types but contravariant in the argument types!

Consider whether we should allow

\[
\text{int \times int \times int} \rightarrow \text{int} \lll \text{int \times int \rightarrow int}
\]

Take the function

\[
\text{thd} \triangleq \lambda x : \text{int \times int \times int.} \pi_3 x
\]

This would allow us to type \( \vdash \text{thd} : \text{int \times int \rightarrow int} \) and therefore type \( \text{thd} (1, 2) \), but this is clearly unsafe!

When can we use something of type \( \tau_1 \rightarrow \tau_2 \) when we expect a \( \tau_1' \rightarrow \tau_2' \)? If we expect a \( \text{int \rightarrow int} \), we might give it an int, like \(-1\), but this would be a problem if we got a \( \text{nat \rightarrow nat} \)! (On the other hand, if we got an \( \text{int \rightarrow nat} \), this would be fine: we’ll never know it’s only giving us nats back).

5 Examples

\[
\begin{align*}
\text{nat} &\lll \text{int} &\lll \text{int} \\
\text{int} &\lll \text{nat} &\lll \text{nat} \\
\text{int} \times \text{nat} &\lll \text{nat} \times \text{int} \\
\text{nat} + \text{nat} &\lll \text{int} + \text{int} \\
\text{int} \rightarrow \text{int} \times \text{int} \times \text{int} &\lll \text{int} \rightarrow \text{int} \times \text{int}
\end{align*}
\]

6 Progress and Preservation

Lemma 3. If \( \tau \lll \tau_1 \times \cdots \times \tau_n \) then \( \tau = \tau_1' \times \cdots \times \tau_m' \) for some \( m \geq n \) where for all \( i \in [1, n] \), we have \( \tau_i' \lll \tau_i \).

Proof. By induction on the derivation of \( \tau \lll \tau_1 \times \cdots \times \tau_n \).

- **Sub-Refl.** The result is clear.

- **Sub-Width.** Then \( \tau = \tau_1 \times \cdots \times \tau_m \) for \( m \geq n \). We have \( \tau_i \lll \tau_i \) by Sub-Refl.

- **Sub-Depth.** Then \( \tau = \tau_1' \times \cdots \times \tau_n' \) where \( \tau_i' \lll \tau_i \).
• SUB-TRANS. Then $\tau <: \tau'$ and $\tau' <: \tau_1 \times \cdots \times \tau_n$. By induction, $\tau' = \tau'_1 \times \cdots \times \tau'_m$ for some $m \geq n$ where for all $i \in [1, n]$, we have $\tau'_i <: \tau_i$. By another induction, $\tau = \tau''_1 \times \cdots \times \tau''_k$ for some $k \geq m$ where for all $i \in [1, m]$, we have $\tau''_i <: \tau'_i$. Apply transitivity of $\geq$ and $<:$. □

Lemma 4 (Canonical Forms (Old)). If $\bullet \vdash e : \tau_1 \times \cdots \times \tau_n$ and $e \text{ val}$, then $e = (v_1, \ldots, v_n)$ where $v_i \text{ val}$.

This is no longer true!

Lemma 5 (Canonical Forms (New)). If $\bullet \vdash e : \tau_1 \times \cdots \times \tau_n$ and $e \text{ val}$, then $e = (v_1, \ldots, v_m)$ where $v_i \text{ val}$ and $m \geq n$.

Proof. By induction on the derivation of $\bullet \vdash e : \tau_1 \times \cdots \times \tau_n$.

• x-I. Then $e = (v_1, \ldots, v_n)$. By inversion on the value rules, we have $v_i \text{ val}$.

• SUB. Then $\bullet \vdash e : \tau$ and $\tau <: \tau_1 \times \cdots \times \tau_n$. By Lemma ??, $\tau = \tau'_1 \times \cdots \times \tau'_m$ for some $m \geq n$ where for all $i \in [1, n]$, we have $\tau'_i <: \tau_i$. By induction, $e = (v_1, \ldots, v_k)$ where $v_i \text{ val}$ and $k \geq m$. We have $k \geq n$. □

This used to be obvious just by doing inversion on the typing rules; it’s not anymore!

Lemma 6 (Inversion on Typing).

1. If $\bullet \vdash (e_1, \ldots, e_n) : \tau$ then $\tau_1 \times \cdots \times \tau_n <: \tau$ and for all $i$, $\bullet \vdash e_i : \tau_i$.

2. If $\bullet \vdash \pi_i e : \tau$ then $\bullet \vdash e : \tau_1 \times \cdots \times \tau_n$ and $1 \leq i \leq n$ and $\tau_i <: \tau$.

Lemma 7 (Preservation). If $\bullet \vdash e : \tau$ and $e \mapsto e'$ then $\bullet \vdash e' : \tau$.

Proof. Consider the case for $\pi_i (e_1, \ldots, e_n) \mapsto e_i$. Then by Lemma ??, $\bullet \vdash (e_1, \ldots, e_n) : \tau_1 \times \cdots \times \tau_n$ and $1 \leq i \leq n$ and $\tau_i <: \tau$. By another application of Lemma ??, we have $\tau'_1 \times \cdots \times \tau'_n <: \tau_1 \times \cdots \times \tau_n$ and $\bullet \vdash e_i : \tau'_i$. By Lemma ??, we have $\tau'_i <: \tau_i$. By T-SUB, we have $\bullet \vdash e_i : \tau_i$ and $\bullet \vdash e_i : \tau$. □

Lemma 8 (Progress). If $\bullet \vdash e : \tau$ then either $e \text{ val}$ or $e \mapsto e'$.

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

• x-E. Then $e = \pi_i e_0$ and $\bullet \vdash e_0 : \tau_1 \times \cdots \times \tau_n$ and $1 \leq i \leq n$. By induction, either $e_0 \text{ val}$ or $e_0 \mapsto e'_0$. Consider the case where $e_0 \text{ val}$. Then, by Canonical Forms, $e_0 = (v_1, \ldots, v_m)$ where $v_i \text{ val}$ and $m \geq n$. We have $e \mapsto v_i$.

• T-SUB. Then $\bullet \vdash e : \tau'$ and $\tau' <: \tau$. By induction. □