SWEN430 - Compiler Engineering (2018)

Lecture 9 - Typing III : Formal Type Systems

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Subtyping?

- Many programming languages support some notation of *subtype*.

  E.g. — In Java, all class types are subtypes on `Object`.
  — In Ada, a subrange of integers is a subtype.

  But what does this mean?

- If we regard types as sets of values, then $T <: T'$ is equivalent to $|[T]| \subseteq |[T']|$, where $|[T]|$ is the set of values denoted by type T.

- But ... some argue that data types are not sets of values, but are distinguished by the operations that can be performed.

  E.g. consider the difference between stacks and queues.

  Better to define in terms of how values of a type can be used.

- If T is a subtype of T', written $T <: T'$ (or $T \leq T'$), then any term of type S can be safely used in a context where a term of type T is expected. (Paraphrased from Wikipedia.)

What is Subtyping?

- A stronger version is known as the *Liskov substitution principle* (aka behavioural subtyping):
  fIf S is a subtype of T, then objects of type T may be replaced with

  objects of type S without altering any of the desirable properties of the program (correctness, task performed, etc.).

- This is a semantic/correctness property not enforced by programming language type systems.

Subtyping in OO Languages

Suppose we have the following declarations:

```
Integer i = new Integer(1);
Float f = new Float(1.0);
Number n;
Object o;
```

What can be assigned to what?

```
n=i; // valid (because of subtyping)
n=f; // valid (because of subtyping)
n=o; // invalid (without cast)
```

- The expression on the rhs must be a subtype of the variable on the left.
- In Object-Oriented Languages (e.g. Java), *subtyping* is everywhere!
- In Java, subtyping and *subclassing* are the same thing
  - But, not in e.g. C++ (due to private inheritance)

$\lambda_{\leq} = \lambda_{\rightarrow}$ with subtyping

- Want subtype hierarchy where $\texttt{real} \leq \texttt{num}$ and $\texttt{int} \leq \texttt{num}$

- Want add operators:
  - $+ : (\texttt{real}, \texttt{real}) \rightarrow \texttt{real}$
  - $+ : (\texttt{int}, \texttt{int}) \rightarrow \texttt{int}$ (faster integer-only addition)

- Then, we could type "useful" functions such as:

$$\begin{array}{l} \textbf{let} \quad \texttt{f} = (\lambda \texttt{x} : \texttt{num},\ \texttt{y} : \texttt{num} \rightarrow \texttt{int}.\ 1 + (\texttt{y}\ \texttt{x}))) \\ \qquad \textbf{in}\ \texttt{f}\ \texttt{2.0}\ (\lambda \texttt{z} : \texttt{num}.\ 1) \end{array}$$

- Note: "**let** $\texttt{x} = \texttt{e}_1$ **in** $\texttt{e}_2$" is syntactic sugar for $(\lambda \texttt{x}.\texttt{e}_2)\ \texttt{e}_1$

Syntax and Semantics for $\lambda_{\leq}$

$\mathtt{t} ::=$	(Terms)
$\mid \mathtt{x}$	(Variables)
$\mid \mathtt{v}$	(Values)
$\mid \mathtt{t}\ \mathtt{t}$	(Apps)
$\mid \mathtt{t} + \mathtt{t}$	(Add)
$\mathtt{v} ::=$	(Values)
$\mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{t}$	(Function)
$\mid \mathtt{c}$	(Integer)
$\mid \mathtt{c.c}$	(Real)
$\mathtt{T} ::=$	(Types)
$\mid \mathtt{T} \to \mathtt{T}$	(Fun type)
$\mid \mathtt{int}$	(Int type)
$\mid \mathtt{real}$	(Real type)
$\mid \mathtt{num}$	(Num type)

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(App2)}$$

$$\frac{}{(\lambda \mathtt{x} : \mathtt{T}.\mathtt{t}_2)\ \mathtt{v}_1 \longrightarrow \mathtt{t}_2[\mathtt{x} \mapsto \mathtt{v}_1]} \quad \text{(FunApp)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1 + \mathtt{t}_2 \longrightarrow \mathtt{t}_1' + \mathtt{t}_2} \quad \text{(Add1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1 + \mathtt{t}_2 \longrightarrow \mathtt{v}_1 + \mathtt{t}_2'} \quad \text{(Add2)}$$

$$\frac{\mathtt{v}_1 + \mathtt{v}_2 = \mathtt{v}_3}{\mathtt{v}_1 + \mathtt{v}_2 \longrightarrow \mathtt{v}_3} \quad \text{(Add3)}$$

Note: The condition $v_1 + v_2 = v_3$ in rule (Add3) says that v_3 is the actual arithmetic sum of v_1 and v_2, where as $\mathtt{v}_1 + \mathtt{v}_2$ on the bottom is a syntactic term.

Subtyping relation for $\lambda_{\leq}$

$$\frac{}{\mathtt{T}_1 \leq \mathtt{T}_1} \quad \text{(S-Refl)} \qquad \frac{\mathtt{T}_1 \leq \mathtt{T}_2 \quad \mathtt{T}_2 \leq \mathtt{T}_3}{\mathtt{T}_1 \leq \mathtt{T}_3} \quad \text{(S-Trans)}$$

$$\frac{}{\mathtt{real} \leq \mathtt{num}} \quad \text{(S-Real)} \qquad \frac{}{\mathtt{int} \leq \mathtt{num}} \quad \text{(S-Int)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_3, \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2, \quad \mathtt{T}_2 \leq \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_3} \quad \text{(T-App)}$$

Subtyping relation is reflexive and transitive.

T-App is similar to $\lambda_{\to}$, but now supports subtyping.

Function Subtyping

$$\frac{\mathtt{T}_1 \geq \mathtt{T}_3 \quad \mathtt{T}_2 \leq \mathtt{T}_4}{\mathtt{T}_1 \rightarrow \mathtt{T}_2 \leq \mathtt{T}_3 \rightarrow \mathtt{T}_4} \quad \text{(S-Fun)}$$

- Subtyping of functions is **contravariant** in parameter position
  - Contravariant is when you can replace a type with a super type
  - Prevents this: $(\lambda \mathtt{x} : \mathtt{num} \rightarrow \mathtt{int}.\ \mathtt{x}\ 1.0)\ (\lambda \mathtt{y} : \mathtt{int}.\ \mathtt{y} + \mathtt{y})$

- Subtyping of functions is **covariant** in result position
  - Covariant is when you can replace a type with a subtype
  - Prevents this: $(\lambda \mathtt{x} : \mathtt{num} \rightarrow \mathtt{int}.\ (\mathtt{x}\ 1.0) + 1)\ (\lambda \mathtt{y} : \mathtt{num}.\ \mathtt{y})$

Subtyping for the While Language

- Subtyping in WHILE exists because of **union types!**
  - » $\texttt{int} \vee \texttt{string}$ is a type whose values are either `int` or `string`
  - » values of type $\texttt{int} \vee \texttt{null}$ are either `int` or `null`; i.e. optional `int`.
  - » $\texttt{int} \vee \texttt{int}$ is the same as `int`

  (Usually written `int|string`, etc.)

- Examples:
  - » $\texttt{int} \leq \texttt{int} \vee \texttt{null}$
  - » $\texttt{int} \vee \texttt{int} \leq \texttt{int}$
  - » $\{\,(\texttt{int} \vee \texttt{int[]})\ \texttt{f}\,\} \leq \{\,\texttt{int f}\,\} \vee \{\,\texttt{int[] f}\,\}$

    f(Note: $\{\,\texttt{T f}\,\}$ is the type of records with one field called `f` whose values are of type `T`.

Subtyping Relation (First Attempt)

So, how do we define subtyping for While?

$$\frac{}{\mathtt{T} \leq \mathtt{T}} \quad \text{(S-Reflex)} \qquad \frac{\forall i \in \{0 \ldots |n|\} \ : \ T_i \ \leq \ T_i'}{\{\overline{T}\ \overline{n}\} \ \leq \ \{\overline{T'}\ \overline{n}\}} \quad \text{(S-Record-Depth)}$$

$$\frac{\mathtt{T}_1 \leq \mathtt{T}_3 \quad \mathtt{T}_2 \leq \mathtt{T}_3}{\mathtt{T}_1 \vee \mathtt{T}_2 \leq \mathtt{T}_3} \quad \text{(S-Union1)} \qquad \frac{\mathtt{T}_1 \leq \mathtt{T}_2}{\mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_3} \quad \text{(S-Union2)}$$

(**NOTE:** this is the basic subtype algorithm you will implement)

- **Examples**:
  - » $\mathtt{int} \leq \mathtt{int} \vee \mathtt{null}$ (by S-Union2, S-Reflex)
  - » $\mathtt{int} \vee \mathtt{int} \leq \mathtt{int}$ (by S-Union1, S-Reflex)
  - » $\{(\mathtt{int} \vee [\mathtt{int}])\ \mathtt{f}\} \leq \{\mathtt{int}\ \mathtt{f}\} \vee \{[\mathtt{int}]\ \mathtt{f}\}$ (by ?)

Soundness & Completeness of Subtyping

Type Semantics

$$
\begin{aligned}
[\![\texttt{int}]\!] &= \mathbb{Z} \\
[\![\,\{\overline{\texttt{T f}}\}\,]\!] &= \left\{ \{\overline{\texttt{f : v}}\} \;\middle|\; \texttt{v}_1 \in [\![\texttt{T}_1]\!], \ldots, \texttt{v}_n \in [\![\texttt{T}_n]\!] \right\} \\
[\![\texttt{T}_1 \vee \ldots \vee \texttt{T}_n]\!] &= [\![\texttt{T}_1]\!] \cup \ldots \cup [\![\texttt{T}_n]\!]
\end{aligned}
$$

Subtyping Soundness

If $\texttt{T}_1 \leq \texttt{T}_2$ then $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$

Subtyping Completeness

If $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$ then $\texttt{T}_1 \leq \texttt{T}_2$

- Here, $\texttt{T}_1 \leq \texttt{T}_2$ represents outcome of **subtype algorithm** whilst $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$ represents **idealised solution**

Soundness & Completeness of Subtyping (in English)

- Again, in English:
  - **Soundness**: if subtype algorithm says $\mathtt{T}_1$ subtype of $\mathtt{T}_2$, then every value which could be in $\mathtt{T}_1$ must be permitted in $\mathtt{T}_2$
  - **Completeness**: if every value which could be in $\mathtt{T}_1$ is permitted in $\mathtt{T}_2$, then subtype algorithm should report that $\mathtt{T}_1$ is subtype of $\mathtt{T}_2$
- **Question:** *is our subtype relation sound and complete?*

  (how do we even go about trying to answer this?)
- **Answer:** *it's sound, but not complete*

Sound & Complete Subtyping Algorithm

- **See:** Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Proc. VMCAI*, David J. Pearce, 2013.