Types in Programming Languages (6)

Christian Urban

http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/
Story So Far

We started with a simple expression language where every expression (if it is typable at all) has a unique type.

There are functions (identity functions, sorting, list operations) which are the same for any type:

\[
\lambda x. x : T \Rightarrow T \\
\lambda x. x : (T \Rightarrow T) \Rightarrow (T \Rightarrow T)
\]

Therefore we considered polymorphism and type-schemes.
We studied a simple language of types and expressions:

\[
T ::= X \quad \text{type variables}
\]
\[
T \to T \quad \text{function types}
\]
\[
e ::= x \quad \text{variables}
\]
\[
e e \quad \text{applications}
\]
\[
\lambda x.e \quad \text{lambda-abstractions}
\]
\[
\text{let } x = e \text{ in } e \quad \text{lets}
\]

We looked at two algorithms that given a (valid) context and an expression, calculate the type (if there exists one); they even calculated a principal type scheme for a typable expression.
Type-safety is then the combination of the preservation and progress property.

**Preservation:**
If $\emptyset \vdash e : T$ and $e \rightarrow e'$ then $\emptyset \vdash e' : T$

**Progress:**
If $\emptyset \vdash e : T$ then either there exists an $e'$ with $e \rightarrow e'$, or $e$ is a value.
Type-safety is then the combination of the preservation and progress property.

**Preservation:**
If $\emptyset \vdash e : T$ and $e \downarrow v$ then $\emptyset \vdash v : T$

**“Progress”:**
If $\emptyset \vdash e : T$ then either there exists a $v$ such that $e \downarrow v$. 
Type-safety is then the combination of the preservation and progress property.

**Preservation:**
If $\emptyset \vdash e : T$ and $e \downarrow v$ then $\emptyset \vdash v : T$

**“Progress”:**
If $\emptyset \vdash e : T$ then either there exists a $v$ such that $e \downarrow v$.

In order to establish them we need to do several proofs by induction (some of them are quite tricky).
Type-systems and type-safety are designed to prevent things like:

```c
union {
    float f;
    int i;
} unsafe_union

unsafe_union.f = 1.5
printf ("%d", unsafe_union.i)
```
Sometimes functions need to indicate that they fail and have to handle failure.

\[
e ::= \ldots \quad \text{error} \quad \text{error value} \quad \text{error handling}
\]

\[
\begin{align*}
\text{valid } \Gamma & \quad \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{try } e_1 \text{ with } e_2 : T} \\
\Gamma \vdash \text{error} : T & \quad \text{error}
\end{align*}
\]
Failures

Evaluation rules:

\[
\begin{align*}
& e_1 \Downarrow \text{error} \\
& \quad \quad e_1 e_2 \Downarrow \text{error} \\
& \quad \quad \quad \quad \quad \quad \quad e_1 \Downarrow v \\
& \quad \quad \quad \quad \quad \quad \quad \text{try } e_1 \text{ with } e_2 \Downarrow v \\
& e_1 \Downarrow \text{error} \\
& \quad \quad e_2 \Downarrow v \\
& \quad \quad \quad \quad \quad \quad \quad \text{try } e_1 \text{ with } e_2 \Downarrow v
\end{align*}
\]
Failure

Preservation and progress in the presence of errors

**Preservation:**
If $\emptyset \vdash e : T$ and $e \rightarrow e'$ then $\emptyset \vdash e' : T$

**Progress:**
If $\emptyset \vdash e : T$ then either there exists an $e'$ with $e \rightarrow e'$, or $e$ is a value, or $e$ is an error.
Extending the Language

- Adding new types, such as `unit`, `nat`, `\( T \)` `list` `\( T \times T \)`, does not pose any difficulties.

- Same with simple expressions such as
  
  \[
  0, 1, 2 \ldots \\
  \text{nil}, e_1 :: e_2 \\
  (e_1, e_2)
  \]

- Difficulties arose with references - the naïve approach leads to problems in the `let-rule`. We needed to impose a restriction.
In a real programming language we need non-termination

\[ e ::= \ldots \]
\[ \mid \text{fix } e \quad \text{fixed point} \]

The following abbreviation is useful:

\[
\text{letrec } x = e_1 \text{ in } e_2 \overset{\text{def}}{=} \text{let } x = \text{fix}(\lambda x. e_1) \text{ in } e_2
\]
Recursion

- Typing rule for recursions

\[ \Gamma \vdash e : T \rightarrow T \]
\[ \Gamma \vdash \text{fix } e : T \]

- We specify the behaviour of recursion by reduction

\[ \text{fix } (\lambda x. e) \rightarrow e[x := \text{fix } (\lambda x. e)] \]

\[ e \rightarrow e' \]
\[ \text{fix } e \rightarrow \text{fix } e' \]
Kinds of Polymorphism

So far we considered **parametric** polymorphism:

Functions can be used at different type, but they have to be independent of the type.

This allows one to forget about types during run-time (in theory — in practice one can at least minimise the need of types, an example is equality).

**Ad-hoc** polymorphism allows function to compute differently at different type (for example + over integers and reals). Here we have coercions and overloading.
Subtyping

- We write $T <: T'$ to indicate that $T$ is a subtype of $T'$.

- If $T <: T'$, then whenever an expression of type $T'$ is needed then we can use an expression of type $T$.

$$
\Gamma \vdash e : T \quad T <: T' \\
\overline{\quad \Gamma \vdash e : T'}
$$

- General principles of subtyping:

$$
T <: T' \\
\overline{T_1 <: T_2 \quad T_2 <: T_3} \\
\overline{T_1 <: T_3}
$$
Subtyping

If $T <: T'$, then an expression of type $T$ can be coerced to be an expression of type $T'$ (in a unique way).

Problem with uniqueness: assume

$\text{int} <: \text{string}$, $\text{int} <: \text{real}$, $\text{real} <: \text{string}$

Then 3 can be coerced to a string like

- $3 \mapsto "3"
- 3 $\mapsto 3.0$ and $3.0 \mapsto "3.0"

We require coherence - only a unique way.

Munich, 29. November 2006
Other Types

Products (clear)

\[
\begin{align*}
T_1 &\ <: S_1 \quad T_2 &\ <: S_2 \\
T_1 \times T_2 &\ <: S_1 \times S_2
\end{align*}
\]

Functions (not so clear)

\[
\begin{align*}
\text{int} \rightarrow \text{int} &\ <: \text{int} \rightarrow \text{real} \\
\text{and} \\
\text{real} \rightarrow \text{int} &\ <: \text{int} \rightarrow \text{int}
\end{align*}
\]

Therefore

\[
\begin{align*}
S_1 &\ <: T_1 \quad T_2 &\ <: S_2 \\
T_1 \rightarrow T_2 &\ <: S_1 \rightarrow S_2
\end{align*}
\]
Function types

\[ S_1 <: T_1 \quad T_2 <: S_2 \]
\[ T_1 \to T_2 <: S_1 \to S_2 \]

are contra-variant in their arguments, and co-variant in their result

Lists can be co-variant:

\[ T_1 <: T_2 \]
\[ T_1 \text{ list} <: T_2 \text{ list} \]
Interesting Cases

In order to maintain type-safety, references cannot be co- or contra-variant, but have to be non-variant. We achieve this by:

\[
\begin{align*}
T_1 &<: T_2 \\
T_2 &<: T_1 \\
T_1 \text{ ref} &<: T_2 \text{ ref}
\end{align*}
\]

Similarly, arrays:

\[
\begin{align*}
T_1 &<: T_2 \\
T_2 &<: T_1 \\
T_1 \text{ array} &<: T_2 \text{ array}
\end{align*}
\]

but Java allows (a flaw in the design):

\[
\begin{align*}
T_1 &<: T_2 \\
T_1 \text{ array} &<: T_2 \text{ array}
\end{align*}
\]
More formally we have:

**Types:**

\[ T ::= X \quad \text{type variables} \]
\[ T \to T \quad \text{function types} \]
\[ \text{Top} \quad \text{super-type for everything} \]

**Terms:**

\[ e ::= x \quad \text{variables} \]
\[ e e \quad \text{applications} \]
\[ \lambda x . e \quad \text{lambda-abstractions} \]
We have contexts $\Delta$ of (type-variable,type)-pairs. Valid contexts are:

- $\Delta \vdash X \not\in \text{dom } \Delta$
- $\Delta \vdash (X <: T), \Delta$

Subtyping judgements:

- $\Delta \vdash T <: \text{Top}$
- $\Delta \vdash X <: X$
- $(X <: S) \in \Delta \Rightarrow \Delta \vdash S <: T$
- $\Delta \vdash X <: T$
- $\Delta \vdash S_1 <: T_1 \Rightarrow \Delta \vdash T_2 <: S_2$
- $\Delta \vdash T_1 \rightarrow T_2 <: S_1 \rightarrow S_2$
Properties

**Given**

\[
\frac{\text{valid } \Delta}{\Delta \vdash T <: \text{Top}} \quad \text{Top} \quad \frac{\text{valid } \Delta}{\Delta \vdash X <: X} \quad \text{Refl}
\]

\[
(X <: S) \in \Delta \quad \Delta \vdash S <: T
\]

\[
\frac{\Delta \vdash X <: T}{\Delta \vdash X <: T}
\]

\[
\Delta \vdash S_1 <: T_1 \quad \Delta \vdash T_2 <: S_2
\]

\[
\frac{\Delta \vdash T_1 \rightarrow T_2 <: S_1 \rightarrow S_2}{\Delta \vdash T_1 \rightarrow T_2 <: S_1 \rightarrow S_2}
\]

**Do we have reflexivity:**

\[
\Delta \vdash T <: T
\]

**What about transitivity:**

If \( \Delta \vdash T_1 <: T_2 \) and \( \Delta \vdash T_2 <: T_3 \) then \( \Delta \vdash T_1 <: T_3 \).
Simple Type-System

- **Variables**
  \[
  \begin{align*}
  \text{valid } \Gamma & \quad \text{valid } \Delta \quad (x : T) \in \Gamma \\
  \hline
  \Delta \vdash x : T
  \end{align*}
  \]

- **Applications**
  \[
  \begin{align*}
  \Delta ; \Gamma \vdash e_1 : T_1 \rightarrow T_2 & \quad \Delta ; \Gamma \vdash e_2 : T_1 \\
  \hline
  \Delta ; \Gamma \vdash e_1 \ e_2 : T_2
  \end{align*}
  \]

- **Lambdas**
  \[
  \begin{align*}
  \Delta ; x : T_1, \Gamma \vdash e : T_2 & \quad x \not\in \text{dom } \Gamma \\
  \hline
  \Delta ; \Gamma \vdash \lambda x . e : T_1 \rightarrow T_2
  \end{align*}
  \]

- **Subtyping**
  \[
  \begin{align*}
  \Delta ; \Gamma \vdash e : T' & \quad \Delta \vdash T' <: T \\
  \hline
  \Delta ; \Gamma \vdash e : T
  \end{align*}
  \]
Typing Problem

Given contexts $\Delta$ and $\Gamma$, and an expression $e$ what should the subtyping algorithm calculate?
Typing Problem

Given contexts $\Delta$ and $\Gamma$, and an expression $e$ what should the subtyping algorithm calculate?

Returning Top is probably not a good idea.
Typing Problem

Given contexts $\Delta$ and $\Gamma$, and an expression $e$, what should the subtyping algorithm calculate?

- Returning Top is probably not a good idea.

- We like to have a minimal type (according to the subtyping relation).
Possible Question

What should the subtyping rule(s) look like for records?

Explain what is meant by capture-avoiding substitution.

Give a definition for what it means when $\theta$ unifies $T$ and $S$. 
More Next Week

- Slides at the end of

http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/

There is also an appraisal form where you can complain **anonymously**.

- You can say whether the lecture was too easy, too quiet, too hard to follow, too chaotic and so on. You can also comment on things I should repeat.