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.

Story So Far

- We studied a simple language of types and expressions:

$$
\begin{array}{rcll}
T & ::= & X & \text{type variables} \\
  & \mid & T \to T & \text{function types} \\
  \\
e & ::= & x & \text{variables} \\
  & \mid & e\ e & \text{applications} \\
  & \mid & \lambda x.e & \text{lambda-abstractions} \\
  & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{lets}
\end{array}
$$

- 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.

Story So Far

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

Story So Far

- Type-safety is then the combination of the preservation and progress property.

  - Preservation:
    If $\varnothing \vdash e : T$ and $e \Downarrow v$ then $\varnothing \vdash v : T$
  - "Progress":
    If $\varnothing \vdash e : T$ then either there exists a v such that $e \Downarrow v$.

Story So Far

- Type-safety is then the combination of the preservation and progress property.

  - Preservation:
    If $\varnothing \vdash e : T$ and $e \Downarrow v$ then $\varnothing \vdash v : T$
  - "Progress":
    If $\varnothing \vdash e : T$ 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).

Motivation

- Type-systems and type-safety are designed to prevent things like:

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

unsafe_union.f = 1.5
printf ("%d", unsafe_union.i)
```

Failures

- Sometimes functions need to indicate that they fail and have to handle failure.

$$
\begin{array}{lcll}
e & ::= & \ldots & \\
  & | & \mathsf{error} & \text{error value} \\
  & | & \mathsf{try}\ e_1\ \mathsf{with}\ e_2 & \text{error handling}
\end{array}
$$

$$
\frac{\mathsf{valid}\ \Gamma}{\Gamma \vdash \mathsf{error} : T}
\qquad
\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{try}\ e_1\ \mathsf{with}\ e_2 : T}
$$

Failures

- Evaluation rules:

$$\frac{e_1 \Downarrow \mathsf{error}}{e_1\ e_2 \Downarrow \mathsf{error}} \qquad \frac{e_2 \Downarrow \mathsf{error}}{e_1\ e_2 \Downarrow \mathsf{error}}$$

$$\frac{e_1 \Downarrow v}{\mathsf{try}\ e_1\ \mathsf{with}\ e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow \mathsf{error} \quad e_2 \Downarrow v}{\mathsf{try}\ e_1\ \mathsf{with}\ e_2 \Downarrow v}$$

Failure

- Preservation and progress in the presence of errors
  - Preservation:
    If $\varnothing \vdash e : T$ and $e \longrightarrow e'$ then $\varnothing \vdash e' : T$
  - Progress:
    If $\varnothing \vdash e : T$ then either there exists an e' with $e \longrightarrow e'$, 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

$$\mathbf{0}, \mathbf{1}, \mathbf{2} \ldots$$
$$\mathsf{nil}, \boldsymbol{e_1} :: \boldsymbol{e_2}$$
$$(\boldsymbol{e_1}, \boldsymbol{e_2})$$

- Difficulties arose with references - the naïve approach leads to problems in the let-rule. We needed to impose a restriction.

Recursion

- In a real programming language we need non-termination

$$
\begin{array}{lcll}
e & ::= & \ldots & \\
 & | & \mathsf{fix}\ e & \text{fixed point}
\end{array}
$$

- The following abbreviation is useful:

$$
\mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2 \stackrel{\mathsf{def}}{=} \mathsf{let}\ x = \mathsf{fix}(\lambda x.e_1)\ \mathsf{in}\ e_2
$$

Recursion

- Typing rule for recursions

$$\frac{\Gamma \vdash e : T \to T}{\Gamma \vdash \mathsf{fix}\ e : T}$$

- We specify the behaviour of recursion by reduction

$$\mathsf{fix}\ (\lambda x.e) \longrightarrow e[x := \mathsf{fix}\ (\lambda x.e)]$$

$$\frac{e \longrightarrow e'}{\mathsf{fix}\ e \longrightarrow \mathsf{fix}\ e'}$$

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.

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

- General principles of subtyping:

$$\frac{}{T <: T} \qquad \frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3}$$

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

  int <: string, int <: real, real <: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.

Other Types

- Products (clear)

$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{T_1 \times T_2 <: S_1 \times S_2}$$

- Functions (not so clear)

$$\mathsf{int} \to \mathsf{int} <: \mathsf{int} \to \mathsf{real}$$

and

$$\mathsf{real} \to \mathsf{int} <: \mathsf{int} \to \mathsf{int}$$

Therefore

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{T_1 \to T_2 <: S_1 \to S_2}$$

Co/Contra-Variance

- Function types

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{T_1 \to T_2 <: S_1 \to S_2}$$

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

- Lists can be co-variant:

$$\frac{T_1 <: T_2}{T_1\ \mathsf{list} <: T_2\ \mathsf{list}}$$

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:

$$\frac{T_1 <: T_2 \quad T_2 <: T_1}{T_1\,\mathsf{ref} <: T_2\,\mathsf{ref}}$$

- Similarly, arrays:

$$\frac{T_1 <: T_2 \quad T_2 <: T_1}{T_1\,\mathsf{array} <: T_2\,\mathsf{array}}$$

but Java allows (a flaw in the design):

$$\frac{T_1 <: T_2}{T_1\,\mathsf{array} <: T_2\,\mathsf{array}}$$

Formal Matters

More formally we have:

- Types:

$$
\begin{array}{rcll}
T & ::= & X & \text{type variables} \\
 & | & T \rightarrow T & \text{function types} \\
 & | & \mathsf{Top} & \text{super-type for everything}
\end{array}
$$

- Terms:

$$
\begin{array}{rcll}
e & ::= & x & \text{variables} \\
 & | & e\ e & \text{applications} \\
 & | & \lambda x.e & \text{lambda-abstractions}
\end{array}
$$

Subtyping Judgement

- We have contexts Δ of (type-variable,type)-pairs.
  Valid contexts are:

$$\frac{}{\text{valid } \varnothing} \qquad \frac{\text{valid } \Delta \quad X \notin \text{dom } \Delta}{\text{valid } (X <: T), \Delta}$$

- Subtyping judgements:

$$\frac{\text{valid } \Delta}{\Delta \vdash T <: \text{Top}}\ \text{Top} \qquad \frac{\text{valid } \Delta}{\Delta \vdash X <: X}\ \text{Refl}$$

$$\frac{(X <: S) \in \Delta \quad \Delta \vdash S <: T}{\Delta \vdash X <: T}\ \text{Trans}$$

$$\frac{\Delta \vdash S_1 <: T_1 \quad \Delta \vdash T_2 <: S_2}{\Delta \vdash T_1 \to T_2 <: S_1 \to S_2}\ \text{Funs}$$

Properties

- Given

$$\frac{\text{valid } \Delta}{\Delta \vdash T <: \mathsf{Top}}\ \mathsf{Top} \qquad \frac{\text{valid } \Delta}{\Delta \vdash X <: X}\ \mathsf{Refl}$$

$$\frac{(X <: S) \in \Delta \quad \Delta \vdash S <: T}{\Delta \vdash X <: T}\ \mathsf{Trans}$$

$$\frac{\Delta \vdash S_1 <: T_1 \quad \Delta \vdash T_2 <: S_2}{\Delta \vdash T_1 \to T_2 <: S_1 \to S_2}\ \mathsf{Funs}$$

- 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

$$\frac{\mathsf{valid}\ \Gamma \quad \mathsf{valid}\ \Delta \quad (x : T) \in \Gamma}{\Delta; \Gamma \vdash x : T}$$

- Applications

$$\frac{\Delta; \Gamma \vdash e_1 : T_1 \to T_2 \quad \Delta; \Gamma \vdash e_2 : T_1}{\Delta; \Gamma \vdash e_1\ e_2 : T_2}$$

- Lambdas

$$\frac{\Delta; x : T_1, \Gamma \vdash e : T_2 \quad x \notin \mathsf{dom}\ \Gamma}{\Delta; \Gamma \vdash \lambda x.e : T_1 \to T_2}$$

- Subtyping

$$\frac{\Delta; \Gamma \vdash e : T' \quad \Delta \vdash T' <: T}{\Delta; \Gamma \vdash e : T}$$

Typing Problem

- Given contexts Δ and Γ, and an expression e what should the subtyping algorithm calculate?

Typing Problem

- Given contexts Δ and Γ, and an expression e what should the subtyping algorithm calculate?

- Returning Top is probably not a good idea.

Typing Problem

- Given contexts Δ and Γ, 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 θ 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.