Types in Programming Languages (8)

Christian Urban

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

Recap

- We ensured the property of control-flow safety of typed assembler programs:

  Property: A program cannot jump to an arbitrary address, but only to a well-defined subset of possible entry points.

- Type-inference was **not** employed: the compiler has to give enough information during the compilation process so that the bytecode only needs to be type-checked.

Kinds of Polymorphism

- Last year we considered parametric polymorphism, where functions can be used for different types, but the functions have to be independent of the types (e.g. reversing of lists).
- In practice however one also want the definition of functions to depend on types (for example addition over integers and floats behave differently).
- One solution: Ad-hoc polymorphism allows functions to work differently at different type (for example object-oriented programming languages and also OCaml).

  An example of ad-hoc polymorphism is subtyping.

Subtyping

- We wrote $T <: T'$ 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.

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

- Properties we expect of subtyping:

$$\frac{}{T <: T}\;\text{Refl} \qquad \frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3}\;\text{Trans}$$

Subtyping

- Another property: 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, i.e. uniqueness of coercion.

Types and Terms

- Types:

T	$::=$	X	type variables
	$\mid$	$T \to T$	function types
	$\mid$	Top	super-type of everything

- Terms:

e	$::=$	x	variables
	$\mid$	$e\ e$	applications
	$\mid$	$\lambda x.e$	lambda-abstractions

Function Types

- Subtyping of functions (not obvious): e.g.

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

and

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

Therefore

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

  - contra-variant in the argument, and
  - co-variant in the result

Subtyping Judgement

- As usual we have contexts Δ of (type-var,type)-pairs. Valid contexts are:

$$\frac{}{\text{valid } \varnothing} \qquad \frac{\text{valid } \Delta \quad X \not\in \text{dom } \Delta}{\text{valid } (X <: T), \Delta}$$

- 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 (I)

- Given

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

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

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

- we have reflexivity (easy proof):

$$\Delta \vdash T <: T$$

- and transitivity (tricky proof):

If $\Delta \vdash T_1 <: T_2$ and $\Delta \vdash T_2 <: T_3$ then $\Delta \vdash T_1 <: T_3$.

Properties (II)

- Given

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

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

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

- subtyping is decidable (with some priorities the rules are syntax-directed).

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?

- The rules are very not helpful: the problem is the Trans-rule

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

This rule is always applicable and we have to guess T'.

Algorithmic Type-System

- The rules for variables and lambdas are the same; delete the rule for transitivity.

- The rule for applications

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

is replaced by

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : T_1 \quad T_1 = T_{11} \to T_{12} \\ \Delta; \Gamma \vdash e_2 : T_2 \quad \Delta \vdash T_2 <: T_{11}\end{array}}{\Delta; \Gamma \vdash e_1\ e_2 : T_{12}}$$

Properties

- Soundness: If $\Delta; \Gamma \vdash e : T$ in the new system, then $\Delta; \Gamma \vdash e : T$ in the old system.
- Completeness: If $\Delta; \Gamma \vdash e : T$ in the old system, then $\Delta; \Gamma \vdash e : S$ for some S in the new system with $\Delta \vdash S <: T$.
- Both properties by induction on the respective relation.

Joins

- Type-checking expressions with multiple branches is a bit tricky: for example

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{bool} \quad \Delta; \Gamma \vdash e_2 : T \quad \Delta; \Gamma \vdash e_3 : T}{\Delta; \Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

- We need to calculate the minimal type of both branches - this is called the join.

  A type J is called a join of S and T if $S <: J$ and $T <: J$, and for all types U, if $S <: U$ and $T <: U$, then $J <: U$

  (Depending on the system, calculation of joins is not always possible.)

Joins

- Type-checking expressions with multiple branches is a bit tricky: for example

$$\frac{\Delta; \Gamma \vdash e_1 : \text{bool} \quad \Delta; \Gamma \vdash e_2 : T_1 \quad \Delta; \Gamma \vdash e_3 : T_2}{\Delta; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_1 \vee T_2}$$

- We need to calculate the minimal type of both branches - this is called the join.

  A type J is called a join of S and T if $S <: J$ and $T <: J$, and for all types U, if $S <: U$ and $T <: U$, then $J <: U$

  (Depending on the system, calculation of joins is not always possible.)

Transition Rules

Given

$$\frac{}{(\lambda x.e_1)e_2 \longrightarrow e_1[x:=e_2]} \qquad \frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1 e_2 \longrightarrow e_1 e_2'}$$

- then in general it is possible that $s : S$ and $t : T$ with $s \longrightarrow^* t$ and $T <: S$, but **not** $S <: T$.

Explicit Casts

Casting is often necessary in object-oriented languages. We can add a term-constructor for explicit castings.

- Terms:

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & (S <: T)\ e & \text{casts} \end{array}$$

with the rule

$$\frac{\Delta; \Gamma \vdash e : T \quad \Delta \vdash S <: T}{\Delta; \Gamma \vdash (S <: T)\ e : S}$$

Historical Points

- One of the main points of subtyping is to model class hierarchies:

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D}{C <: D}$$

- There is a lot of research how object-oriented languages can be understood in terms of subtyping (those languages are prone to problems with typing). One development is Featherweight Java.
- Even functional languages benefited from this research (Ocaml).

Data Types

- We next consider how to represent datatypes, such as
  - Booleans (either True or False)
  - Lists (either Nil or Cons)
  - Nats (either Zero or Successor)
  - Bin-trees (either Leaf or Node)
- The question is how to include them into the typing-system. Introducing them primitively is unsatisfactory. Why?
- We consider here the PLC.

Syntax of PLC

- Types:

$$\begin{array}{rcll} T & ::= & X & \text{type variables} \\ & \mid & T \to T & \text{function types} \\ & \mid & \forall X.T & \forall\text{-type} \end{array}$$

- Terms:

$$\begin{array}{rcll} e & ::= & x & \text{variables} \\ & \mid & e\ e & \text{applications} \\ & \mid & \lambda x.e & \text{lambda-abstractions} \\ & \mid & \Lambda X.e & \text{type-abstractions} \\ & \mid & e\ T & \text{type-applications} \end{array}$$

Transitions in PLC

- We have the same transitions as in the lambda-calculus, e.g.

$$\overline{(\lambda x.e_1)e_2 \longrightarrow e_1[x := e_2]}$$

plus rules for type-abstractions and type-applications

$$\overline{(\Lambda X.e)T \longrightarrow e[X := T]}$$

- Confluence and Termination holds for $\longrightarrow$.

Typing Rules

- Type-Generalisation

$$\frac{\Gamma \vdash e : T \quad X \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda X.e : \forall X.T}$$

- Type-Specialisation

$$\frac{\Gamma \vdash e : \forall X.T_1}{\Gamma \vdash e\ T_2 : T_1[X := T_2]}$$

- Interestingly, for PLC the problems of type-checking and type-inference are computationally equivalent and **undecidable**!

Typing Rules

- Type-Generalisation

Therefore we explicitly annotate the type in lambda-abstractions

$$\lambda x : T.e$$

Type-checking is then trivial. (But is it useful?)

- Typ

- Interestingly, for PLC the problems of type-checking and type-inference are computationally equivalent and **undecidable**!

Datatypes

We are now returning to the question of representing datatypes in PLC.

- Booleans with values true and false is represented by

$$\mathsf{bool} \stackrel{\mathsf{def}}{=} \forall X. X \to (X \to X)$$

- $\mathsf{true} \stackrel{\mathsf{def}}{=} \Lambda X. \lambda x_1 : X. \lambda x_2 : X. x1$
  $\mathsf{false} \stackrel{\mathsf{def}}{=} \Lambda X. \lambda x_1 : X. \lambda x_2 : X. x2$

  These are the only two closed normal terms of type bool.

Lists

- Lists can be represented as

$$X \text{ list} \stackrel{\text{def}}{=} \forall Y.Y \to (X \to Y \to Y) \to Y$$

- $\text{Nil} \stackrel{\text{def}}{=} \Lambda X Y.\lambda x : Y.\lambda f : X \to Y \to Y.x$

  $\text{Cons} \stackrel{\text{def}}{=} \ldots$

  These are infinitely closed normal terms of this type.

- We also have unit-, product- and sum-types. From this we can already build up all **algebraic types** (a.k.a. data types).

Possible Questions

- Question: A typed programming language is polymorphic if a term of the language may have different types (right or wrong)?
- PLC is at the heart of the immediate language in GHC: let-polymorphism of ML is compiled to (annotated) PLC.
- Describe the notion of beta-equality of terms in PLC. How can one decide that two typable PLC-terms are in this relation? Why does this fail for untypable terms?

Further Points

- Functional programming languages often allow bounds (constraints) on types:
  for example the membership functions of lists has type $X \to X$ list $\to$ bool, where X can only be a type with defined equality.

- Haskell generalises this idea by using type-classes
- This is in contrast to object-oriented programming languages which use subtyping for modelling this.