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 $T$.

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

Properties we expect of subtyping:

- $T <: T$ (Refl)
- $T_1 <: T_2 \quad T_2 <: T_3 \quad T_1 <: T_3$ (Trans)
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

\[
\text{int} <: \text{string}, \text{int} <: \text{real}, \text{real} <: \text{string}
\]

Then $3$ can be coerced to a string like

\[
3 \mapsto \text{"3"}, \quad 3 \mapsto 3.0 \quad \text{and} \quad 3.0 \mapsto \text{"3.0"}
\]

We require coherence, i.e. uniqueness of coercion.
Types and Terms

Types:

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

Terms:

\[ e ::= x \quad \text{variables} \]
\[ \quad e \ e \quad \text{applications} \]
\[ \quad \lambda x.\ e \quad \text{lambda-abstractions} \]
Function Types

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

\[ \text{int} \rightarrow \text{int} <: \text{int} \rightarrow \text{real} \]

and

\[ \text{real} \rightarrow \text{int} <: \text{int} \rightarrow \text{int} \]

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 $\Delta$ of (type-var, type)-pairs. Valid contexts are:

- Valid $\emptyset$
- Valid $\Delta \not\succeq \text{dom } \Delta$
- Valid $(X <: T), \Delta$

Subtyping judgements:

- Valid $\Delta$  \[ \Delta \vdash T <: \text{Top} \]  Top
- Valid $\Delta$  \[ \Delta \vdash X <: X \]  Refl
- $(X <: S) \in \Delta$  \[ \Delta \vdash S <: T \]  Trans
- \[ \Delta \vdash X <: T \]
- Valid $\Delta$  \[ \Delta \vdash S_1 <: T_1 \]  Funs
- Valid $\Delta$  \[ \Delta \vdash T_2 <: S_2 \]
- \[ \Delta \vdash T_1 \rightarrow T_2 <: S_1 \rightarrow S_2 \]
Properties (I)

Given

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

Top

\[
\frac{\text{valid } \Delta}{\Delta \vdash X <: X}
\]

Refl

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

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

Trans

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

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

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

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

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

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

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

### Variables

\[
\text{valid } \Gamma \quad \text{valid } \Delta \quad (x : T) \in \Gamma \quad \Rightarrow \quad \Delta; \Gamma \vdash x : T
\]

### Applications

\[
\Delta; \Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Delta; \Gamma \vdash e_2 : T_1 \\
\Delta; \Gamma \vdash e_1 \; e_2 : T_2
\]

### Lambdas

\[
\Delta; x : T_1, \Gamma \vdash e : T_2 \quad x \not\in \text{dom } \Gamma \\
\Delta; \Gamma \vdash \lambda x. e : T_1 \rightarrow T_2
\]

### Subtyping

\[
\Delta; \Gamma \vdash e : T' \quad \Delta \vdash T' <: T \\
\Delta; \Gamma \vdash e : T
\]
Typing Problem

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

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

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

$$\Delta; \Gamma \vdash e : T$$

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

Munich, 10. January 2007 - p.12 (1/1)
The rules for variables and lambdas are the same; delete the rule for transitivity.

The rule for applications

\[ \Delta; \Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Delta; \Gamma \vdash e_2 : T_1 \]

\[ \Delta; \Gamma \vdash e_1 \ e_2 : T_2 \]

is replaced by

\[ \Delta; \Gamma \vdash e_1 : T_1 \quad T_1 = T_{11} \rightarrow T_{12} \]

\[ \Delta; \Gamma \vdash e_2 : T_2 \quad \Delta \vdash T_2 <: T_{11} \]

\[ \Delta; \Gamma \vdash e_1 \ e_2 : T_{12} \]
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 $\Delta \vdash S \ll T$.

- Both properties by induction on the respective relation.
Joins

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

\[
\Delta; \Gamma \vdash e_1 : \text{bool} \quad \Delta; \Gamma \vdash e_2 : T \quad \Delta; \Gamma \vdash e_3 : T
\]

\[
\Delta; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\]

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

\[
\Delta; \Gamma \vdash e_1 : \text{bool} \quad \Delta; \Gamma \vdash e_2 : T_1 \quad \Delta; \Gamma \vdash e_3 : T_2 \\
\Delta; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_1 \lor T_2
\]

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

\[(\lambda x. e_1)e_2 \rightarrow e_1[x:=e_2]\] \hspace{1cm} \[\lambda x. e \rightarrow \lambda x. e'\]

\[e_1 \rightarrow e_1'\]

\[e_2 \rightarrow e_2'\]

\[e_1 e_2 \rightarrow e_1'e_2\]

\[e_1 e_2 \rightarrow e_1'e_2'\]

\[\text{then in general it is possible that } s : S \text{ and } t : T \text{ with } s \rightarrow^* t \text{ and } T <: S, \text{ but not } S <: T.\]
Explicit Casts

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

Terms:

\[ e ::= \ldots \]
\[ | \quad (S <: T) \ e \text{ casts} \]

with the rule

\[ \Delta; \Gamma \vdash e : T \quad \Delta \vdash S <: T \]
\[ \Delta; \Gamma \vdash (S <: T) \ e : S \]
Historical Points

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

\[
\text{class } C \text{ extends } D \\
C <: D
\]

There is a lot of research on 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:

\[ T ::= X \quad \text{type variables} \]
\[ \mid T \rightarrow T \quad \text{function types} \]
\[ \mid \forall X.T \quad \forall\text{-type} \]

Terms:

\[ e ::= x \quad \text{variables} \]
\[ \mid e e \quad \text{applications} \]
\[ \mid \lambda x.e \quad \text{lambda-abstractions} \]
\[ \mid \Lambda X.e \quad \text{type-abstractions} \]
\[ \mid e T \quad \text{type-applications} \]
Transitions in PLC

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

\[(\lambda x. e_1) e_2 \rightarrow e_1[x := e_2]\]

plus rules for type-abstractions and type-applications

\[(\Lambda X. e) T \rightarrow e[X := T]\]

Confluence and Termination holds for \[\rightarrow\].
Typing Rules

**Type-Generalisation**

\[ \Gamma \vdash e : T \quad X \not\in \text{ftv}(\Gamma) \]
\[ \frac{\Gamma \vdash \lambda X.e : \forall X.T}{\Gamma \vdash \lambda X.e : \forall X.T} \]

**Type-Specialisation**

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

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?)

- **Type-Specialisation**

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

  \[
  \text{bool} \overset{\text{def}}{=} \forall X. X \rightarrow (X \rightarrow X)
  \]

- **true** \(\overset{\text{def}}{=} \Lambda X. \lambda x_1 : X. \lambda x_2 : X. x_1\)

- **false** \(\overset{\text{def}}{=} \Lambda X. \lambda x_1 : X. \lambda x_2 : X. x_2\)

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

Lists can be represented as

\[ X \text{ list} \overset{\text{def}}{=} \forall Y. Y \to (X \to Y \to Y) \to Y \]

\[ \text{Nil} \overset{\text{def}}{=} \Lambda X Y. \lambda x : Y. \lambda f : X \to Y \to Y. x \]

\[ \text{Cons} \overset{\text{def}}{=} \ldots \]

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 $X \rightarrow X \text{ list} \rightarrow \text{ bool}$, 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.