Types in Programming Languages (9)

Christian Urban

http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/
Recap from last Week

We reformulated the inference rules for subtyping and typing so that one could read off a typing-algorithm.

The language we considered contained variables, applications and lambda-abstractions (briefly also looked at casts). Main point of subtyping is to analyse typing-systems for object-oriented languages.
Featherweight Java

- small language to study Java proposed by Igarashi, Pierce and Wadler

- contains only: object creation, method invocation, field access, casting and variables (no side-effects, which means it behaves almost like a functional language)

- one design motivation is the type-safety proof; for example since no assignment is possible, one does not need an environment to evaluate an FJ-program (still, FJ is Turing-complete)
Syntax

- an FJ-program consists of
  - a class-table, $CT$, which is a collection of class definitions
  - and a term, which corresponds to the “main-method” in Java

- a class definition has the form

  \[
  \text{class } A \text{ extends } B \{ \ldots \}
  \]

  where super-class is always included (where $B$ is possibly $\text{Object}$)
For example

```java
class Pair extends Object {
    Object fst;
    Object snd;

    Pair (Object f, Object s) {
        super(); this.fst = f; this.snd = s;
    }

    Pair setfst (Object newf) {
        return new Pair(newf, this.snd);
    }
}
```

For example

class Pair extends Object {
    Object fst;
    Object snd;

    Pair (Object f, Object s) {
        super(); this.fst = f; this.snd = s;
    }

    Pair setfst (Object newf) {
        return new Pair(newf, this.snd);
    }
}

constructors need to be always present, e.g. A() { super(); } corresponds to “do nothing”
For example

class Pair extends Object {
    Object fst;
    Object snd;

    Pair (Object f, Object s) {
        super(); this.fst = f; this.snd = s;
    }

    Pair setfst (Object newf) {
        return new Pair(newf, this.snd)
    }
}

collectors always take one argument for each field; super is always invoked
For example

```java
class Pair extends Object {
    Object fst;
    Object snd;

    Pair (Object f, Object s) {
        super(); this.fst = f; this.snd = s;
    }

    Pair setfst (Object newf) {
        return new Pair(newf, this.snd);
    }
}
```

**Method-bodies are always of the form**

```
return t  where t is a term
```
Terms

Terms are:

- **object constructions**, e.g. `new A()`, `new Pair(...)`
- **method invocations**, e.g. `... .setfst(...)`
- **field access**, e.g. `A.f`, `this.snd`
- **variables**, e.g. `this`, `newf`
- **casts**, e.g. `(A)t`, `(Pair)t`
Since we have no assignments, evaluation can be easily formalised, e.g.:

\[
\text{new } Pair(\text{new } A(), \text{new } B()).\text{snd} \\
\text{→ new } B() 
\]

A computation may get stuck if

- a field is accessed which is not declared
- a method is invoked which does not exists
- a cast to something other than a super-class
Reduction Sequence

$$((P' r) \new P' r (\new P' r (\new A(), \new B()),\new A())). \text{fst}.\text{snd}$$

$$\rightarrow$$

$$((P' r) \new P' r (\new A(), \new B())).\text{snd}$$

$$\rightarrow$$

$$\new P' r (\new A(), \new B()).\text{snd}$$

$$\rightarrow$$

$$\new B()$$
Terms and Values

Terms:

\[
T ::= \begin{array}{ll}
\ v \ & \ x \ \\
\ t.f \ & \ t.m(t_1, \ldots, t_n) \\
\ & \ \text{variables} \\
\ & \ \text{field access} \\
\ & \ \text{method invocation} \\
\ & \ \text{object creation} \\
\ & \ \text{cast} \\
\end{array}
\]

Values:

\[
v ::= \text{new } C(v_1, \ldots, v_n)
\]
Classes:

\[ C ::= \text{class } C \text{ extends } C \{ \tilde{C} \ f; \ K \ M \} \]

Constructors:

\[ K ::= C(C \tilde{x})\{\text{super}(\tilde{f}); \ this.\tilde{f} = \tilde{f} \} \]

Methods:

\[ M ::= C m(\tilde{C}\tilde{x})\{\text{return } t \} \]
Subtyping

\[ C <: C \]

\[ C <: D \quad D <: E \]
\[ \therefore C <: E \]

\[ CT(C) = \text{class } C \text{ extends } D \{ \ldots \} \]
\[ \therefore C <: D \]

where \( CT \) is the class-table, a mapping from class-names to class-declarations
Evaluation (I)

\[
\text{new } C(v_1, \ldots, v_n). f_i \rightarrow v_i
\]

\[
m \text{ is defined in } C \text{ as } B m(\bar{B} \bar{x}) \{\text{return } t\}
\]

or so in a super-class of \(C\)

\[
\text{new } C(\bar{v}). m(\bar{u}) \rightarrow t[\bar{x} \mapsto \bar{u}, \text{this } \mapsto \text{new } C(\bar{v})]
\]

in \(t\) the \(\bar{x}\) are instantiated by the \(\bar{u}\) and \text{this} is associated with \(C(\bar{v})\)
Evaluation (II)

\[ C <: D \]

\[
\frac{(D)\text{(new } C(\vec{v}))}{\rightarrow \text{ new } C(\vec{v})}
\]

the rest are “congruence”-rules

\[
\frac{t \rightarrow t'}{t.f \rightarrow t'.f}
\]
Typing (I)

\[
\begin{align*}
  x : C & \in \Gamma \\
  \Gamma \vdash x : C \\
  \Gamma \vdash t : C & \quad C \text{ contains field } C_i f_i \\
  \Gamma \vdash t.f_i : C_i \\
  \Gamma \vdash \vec{u} : \vec{C} & \quad \vec{C} <: \vec{D} \\
  \Gamma \vdash t : C' & \quad \text{and } m : \vec{D} \rightarrow C \text{ in } C' \\
  \Gamma \vdash t.m(\vec{u}) : C
\end{align*}
\]
Typing (II)

\[ \Gamma \vdash t : \tilde{D} \quad \tilde{C} <: \tilde{D} \quad C \text{ consists of fields } \tilde{D} \]

\[ \Gamma \vdash \text{new } C(t) : C \]

\[ \Gamma \vdash t : D \quad D <: C \]

\[ \Gamma \vdash (C)t : C \]

\[ \Gamma \vdash t : D \quad C <: D \quad C \neq D \]

\[ \Gamma \vdash (C)t : C \]

\[ \Gamma \vdash t : D \quad C \not<: D \quad D \not<: C \]

\[ \Gamma \vdash (C)t : C \]
Type-Safety

If $\Gamma \vdash t : C$ and $t \rightarrow t'$ then $\Gamma \vdash t' : C'$ for some $C' <: C$

stupid casts are rejected, but needed for the property above, e.g.

class $A$ extends Object . . .
class $B$ extends Object . . .

$(A)(Object)new B() \rightarrow (A)new B()$
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:

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

Terms:

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

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

\[\Gamma \vdash \forall X. e : \forall X.T\]

Type-Specialisation

\[\Gamma \vdash e : \forall X.T_1\]

\[\Gamma \vdash e \ T_2 : T_1[X := T_2]\]

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?)

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.

Boolean 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. \lambda x_1
false \overset{\text{def}}{=} \lambda X. \lambda x_1 : X. \lambda x_2 : X. \lambda 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.