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 Object)

Class Definitions

- For example

```
class Pair extends Object {
    Object fst;                                   (fields)
    Object snd;

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

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

Class Definitions

- For example

```
class Pair extends Object {
    Object fst;                                   (fields)
    Object snd;

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

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

- constructors need to be always present, e.g. A() { super(); } corresponds to "do nothing"

Class Definitions

- For example

```
class Pair extends Object {
    Object fst;                                   (fields)
    Object snd;

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

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

- constructors always take one argument for each field; super is always invoked

Class Definitions

- For example

```
class Pair extends Object {
    Object fst;                                   (fields)
    Object snd;

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

    Pair setfst (Object newf) {                   (method)
        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 $Pair(\ldots,\ldots)$
- method invocations, e.g. $-$.setfst(...)
- field access, e.g. $A.f$, this.snd
- variables, e.g. this, $newf$
- casts, e.g. $(A)t$, $(Pair)t$

Evaluation

Since we have no assignments, evaluation can be easily formalised, e.g.:

$$\text{new } Pair(\text{new } A(), \text{new } B()).\text{snd}$$

$$\longrightarrow \text{new } B()$$

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

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

$\longrightarrow$

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

$\longrightarrow$

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

$\longrightarrow$

$\text{new } B()$

Terms and Values

- Terms:

$$
\begin{array}{rcll}
T & ::= & x & \text{variables} \\
 & | & t.f & \text{field access} \\
 & | & t.m(t_1, \ldots, t_n) & \text{method invocation} \\
 & | & \mathsf{new}\, C(t_1, \ldots, t_n & \text{object creation} \\
 & | & (C)t & \text{cast}
\end{array}
$$

- Values:

$$v ::= \mathsf{new}\, C(v_1, \ldots, v_n)$$

Classes

- Classes:

$$C \quad ::= \quad \text{class } C \text{ extends } C \ \{\vec{C}\ \vec{f}; \vec{K}\ \vec{M}\}$$

- Constructors:

$$K \quad ::= \quad C(C\ \vec{x})\{\text{super}(\vec{f}); \text{this}.\vec{f} = \vec{f}\}$$

- Methods:

$$M \quad ::= \quad C\ m(\vec{C}\vec{x})\{\text{return } t\}$$

Subtyping

- $C <: C$
- $$\frac{C <: D \quad D <: E}{C <: E}$$
- $$\frac{CT(C) = \text{class } C \text{ extends } D \{\ldots\}}{C <: D}$$

where CT is the class-table, a mapping from class-names to class-declarations

Evaluation (I)

- new $C(v_1, \ldots, v_n).f_i \longrightarrow v_i$
- $$\frac{m \text{ is defined in } C \text{ as } B\ m(\vec{B}\ \vec{x})\{\text{return } t\} \text{ or so in a super-class of } C}{\text{new } C(\vec{v}).m(\vec{u}) \longrightarrow t[\vec{x} \mapsto \vec{u}, \text{this} \mapsto \text{new } C(\vec{v})]}$$

in t the $\vec{x}$ are instantiated by the $\vec{u}$ and this is associated with $C(\vec{v})$

Evaluation (II)

$$\frac{C <: D}{(D)(\text{new}\, C(\vec{v})) \longrightarrow \text{new}\, C(\vec{v})}$$

- the rest are "congruence"-rules

$$\frac{t \longrightarrow t'}{t.f \longrightarrow t'.f}$$

Typing (I)

- $$\frac{x : C \in \Gamma}{\Gamma \vdash x : C}$$

- $$\frac{\Gamma \vdash t : C \quad C \text{ contains field } C_i\ f_i}{\Gamma \vdash t.f_i : C_i}$$

- $$\frac{\Gamma \vdash \vec{u} : \vec{C} \quad \vec{C} <: \vec{D} \quad \Gamma \vdash t : C' \quad \text{and } m : \vec{D} \to C \text{ in } C'}{\Gamma \vdash t.m(\vec{u}) : C}$$

Typing (II)

- $$\frac{\Gamma \vdash \vec{t} : \vec{D} \quad \vec{C} <: \vec{D} \quad C \text{ consists of fields } \vec{D}\ \vec{f}}{\Gamma \vdash \text{new } C(\vec{t}) : C}$$

- $$\frac{\Gamma \vdash t : D \quad D <: C}{\Gamma \vdash (C)t : C}$$

- $$\frac{\Gamma \vdash t : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t : C}$$

- $$\frac{\Gamma \vdash t : D \quad C \not<: D \quad D \not<: C}{\Gamma \vdash (C)t : C}$$ stupid warning

Type-Safety

- If $\Gamma \vdash t : C$ and $t \longrightarrow t'$ 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 ...

  $(A)(\text{Object})\text{new } B() \longrightarrow (A)\text{new } B()$

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:

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

- Terms:

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

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
- Ty

Therefore we explicitly annotate the type in lambda-abstractions

$$\lambda x : T.e$$

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

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

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

Datatypes

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

- Booleans with values true and false is represented by

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

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

  $\text{false} \stackrel{\text{def}}{=} \Lambda X.\lambda x_1 : X.\lambda x_2 : X.x2$

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