Types in Programming Languages (8)

Christian Urban

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

Munich, 10. January 2007 - p.1 (1/1)

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.

$$rac{arGamma dash e : T \quad T <: T'}{arGamma dash e : T'}$$

Properties we expect of subtyping:

$$rac{T}{T <: T} \, {\sf Refl} \quad rac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3} \, {\sf Trans}_{{\sf Munich, 10. January 2007 - p.4 (1/1)}}$$

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

$$\blacksquare 3 \mapsto "3"$$

 \blacksquare 3 \mapsto 3.0 and 3.0 \mapsto "3.0"

We require coherence, i.e. uniqueness of coercion. Munich, 10. January 2007 - p.5 (1/1)

Types and Terms



$egin{array}{cccccc} T & ::= & X & ext{type variables} \ & & T ightarrow T & ext{function types} \ & & ext{Top} & ext{super-type of everything} \end{array}$



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

Function Types

Subtyping of functions (not obvious): e.g. int \rightarrow int <: int \rightarrow real

and

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

Therefore

$$rac{S_1 <: T_1 \quad T_2 <: S_2}{T_1
ightarrow T_2 <: S_1
ightarrow S_2}$$

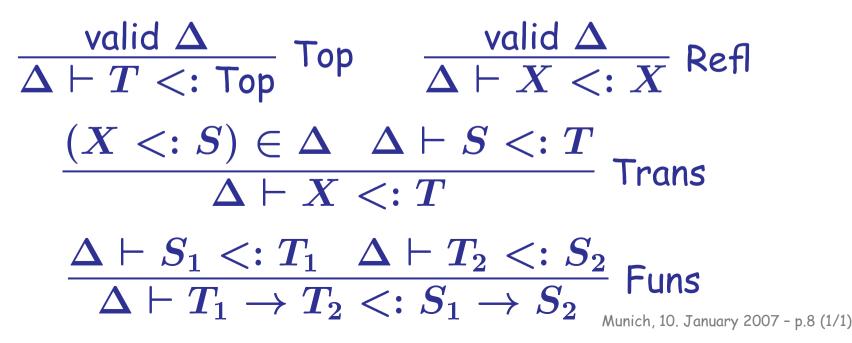
contra-variant in the argument, and
 co-variant in the result

Subtyping Judgement

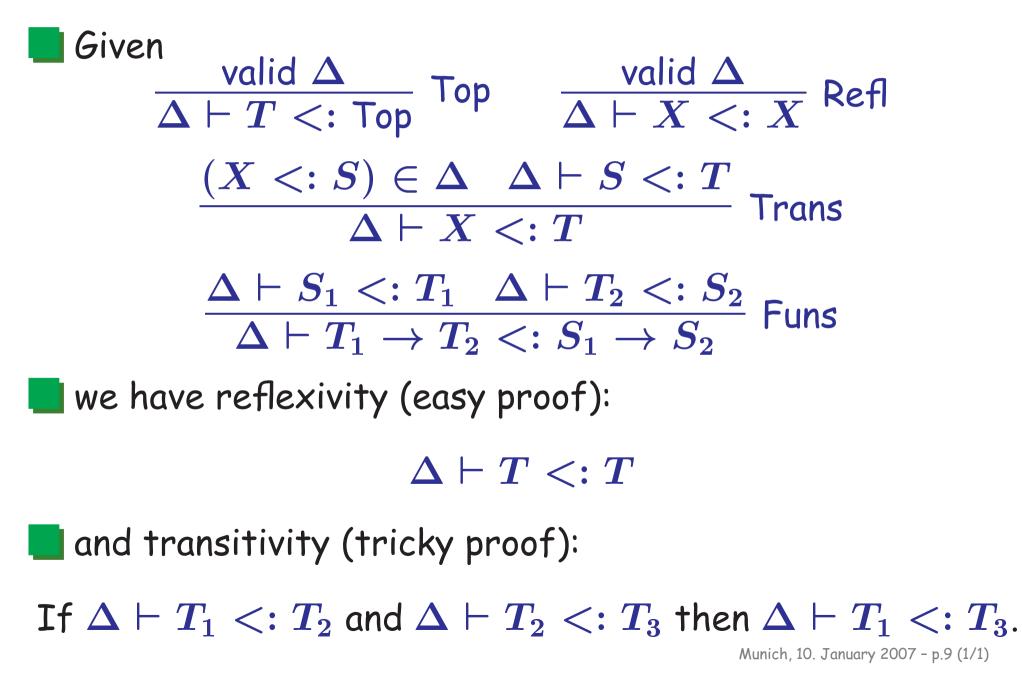
As usual we have contexts Δ of (type-var,type)-pairs. Valid contexts are:

 $rac{\operatorname{valid}\Delta \quad X
ot\in \operatorname{dom}\Delta}{\operatorname{valid} (X <: T), \Delta}$

Subtyping judgements:

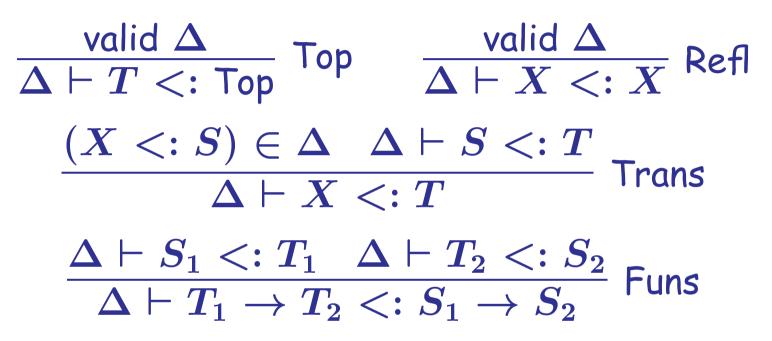


Properties (I)

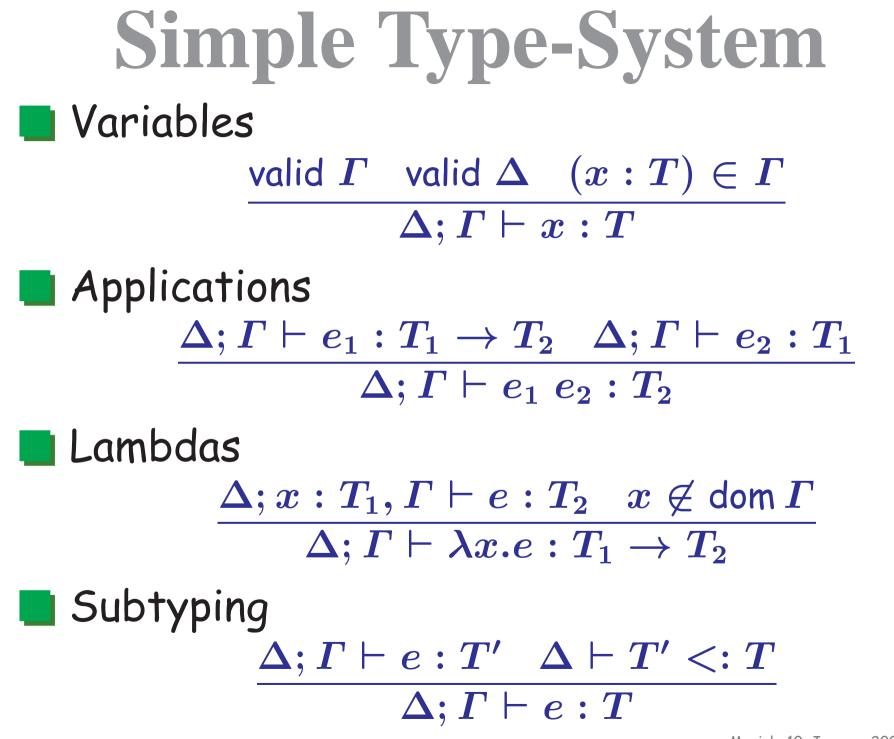


Properties (II)

Given



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



Munich, 10. January 2007 - p.11 (1/1)

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

$$rac{\Delta; arGamma dasherma e: T' \ \Delta dasherma T' <: T}{\Delta; arGamma dasherma e: T}$$
 Trans

This rule is always applicable and we have to guess T^{\prime} .

Algorithmic Type-System

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

The rule for applications $\begin{array}{c} \underline{\Delta; \Gamma \vdash e_1: T_1 \rightarrow T_2 \quad \Delta; \Gamma \vdash e_2: T_1} \\ \underline{\Delta; \Gamma \vdash e_1 \ e_2: T_2} \end{array}$ is replaced by

 $egin{aligned} \Delta; arGamma dash e_1 &: T_1 & T_1 = T_{11}
ightarrow T_{12} \ \Delta; arGamma dash e_2 &: T_2 & \Delta dash T_2 <: T_{11} \ \Delta; arGamma dash e_1 &: e_1 \, e_2 : T_{12} \end{aligned}$

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

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

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.) Munich, 10. January 2007 - p.15 (1/2)

Joins

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

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

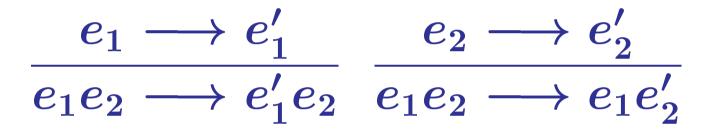
A type J is called a join of S and T if S <: J and T <: J, and for all types U, if S <: U and T <: U, then J <: U

(Depending on the system, calculation of joins is not always possible.) Munich, 10. January 2007 - p.15 (2/2)

Transition Rules

Given

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



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: $e ::= \dots$ $\mid (S <: T) \ e \ casts$ with the rule $\underline{\Delta; \Gamma \vdash e : T \ \Delta \vdash S <: T}$ $\underline{\Delta; \Gamma \vdash (S <: T) \ e : S}$

Historical Points

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

$$rac{class \ C \ extends \ D}{C <: D}$$

- 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: T ::= X type variables $\mid T ightarrow T$ function types $\mid orall X.T$ orall -type



e ::= x variables | e e applications $| \lambda x.e$ lambda-abstractions $| \Lambda X.e$ type-abstractions e T type-applications

Transitions in PLC

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

$$(\lambda x.e_1)e_2 \longrightarrow e_1[x\!:=\!e_2]$$

plus rules for type-abstractions and type-applications

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

Confluence and Termination holds for \longrightarrow .



Type-Generalisation

$$rac{arGamma dash e : T \quad X
ot\in \mathsf{ftv}(arGamma)}{arGamma dash \cdot A X. e : orall X. T}$$

Type-Specialisation

$$rac{arGamma dash e : orall X.T_1}{arGamma dash e \ T_2: T_1[X:=T_2]}$$

Interestingly, for PLC the problems of type-checking and type-inference are computationally equivalent and undecidable! Munich, 10, January 2007 - p.22 (1/2)



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! Munich, 10, January 2007 - p.22 (2/2)



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

Booleans with values true and false is represented by

$$\mathsf{bool} \stackrel{\mathsf{def}}{=} orall X. X o (X o X)$$

The true $\stackrel{\text{def}}{=} \Lambda X.\lambda x_1: X.\lambda x_2: X.x1$ folse $\stackrel{\text{def}}{=} \Lambda X.\lambda x_1: X.\lambda x_2: X.x2$

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

Lists

Lists can be represented as $X \text{ list} \stackrel{\text{def}}{=} \forall Y.Y \rightarrow (X \rightarrow Y \rightarrow Y) \rightarrow Y$ $\text{Nil} \stackrel{\text{def}}{=} \Lambda XY.\lambda x : Y.\lambda f : X \rightarrow Y \rightarrow Y.x$ $Cons \stackrel{\text{def}}{=} \dots$

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 $X \rightarrow X$ list \rightarrow 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.