Part VII: Types for Object-Oriented Programming

Object-oriented programming poses new challenges to type systems.

- Record types
- Subtyping
- Inheritance
- Classes
- Meaning of `this`.

Example Language: Mini-Funnel

Syntax of Mini-Funnel

Name	x, y, z		
Tyvar	a, b, c		
Ident	I, J	$=$	$x \mid I.x$
Term	E, F	$=$	$x \mid E.x \mid E[\overline{T}] \mid E\ F \mid \mathbf{def}\ D\ ;\ E$
		$\mid$	$E\ \&\ F \mid \mathbf{nil}$
Definition	D	$=$	$\epsilon \mid D, D \mid L = E$
Left-hand side	L	$=$	$I[\overline{a}](x : T) : U \mid L\ \&\ L'$
Type	S, T	$=$	$a \mid \{x_1 : T_1, \ldots, x_n : T_n\} \mid T \to U$
		$\mid$	$\overline{a}.T \mid \mu a.T$
Lifted Type	U	$=$	$T \mid \mathbf{nil}$
Environment	Γ, Σ, Δ	$=$	$\epsilon \mid \Gamma, a \mid \Gamma, x : T$

Remarks

- We use a minimal language subset, which exemplifies most of the important typing problems.
- Other constructs can be added by deriving typing rules for a construct from the construct's expansion into the minimal language subset.
- **nil** is used as a constant for the empty process and as a name for the types of processes (which no not return a result).
- Difference between types T and lifted types U: lifted types can be **nil**, normal types cannot.
- To keep the formal treatment simple, we require explicit type annotations everywhere. We expect that it's possible to use clever type inference for inserting most of these annotations automatically.

Type Equivalence

We assume the following equivalences between types.

- a-renaming: The names of $\forall$-bound type variables don't matter. A $\forall$-prefix which binds an empty list of type variables may be dropped.

$$(a)\ \frac{\bar{b} \notin \mathrm{tv}(T)}{\forall \bar{a}.T \equiv \forall \bar{b}.[\bar{b}/\bar{a}]T} \qquad (\textsc{Empty})\ \ \forall \epsilon.T \equiv T$$

- The order of fields in a record does not matter.

$$(\textsc{Permute})\ \frac{\{i_1, \ldots, i_n\} = \{1, \ldots, n\}}{\{x_1 : T_1, \ldots, x_n : T_n\} \equiv \{x_{i_1} : T_{i_1}, \ldots, x_{i_n} : T_{i_n}\}}$$

- A recursive type $\mu a.T$ is equivalent to its (one-step) unfolding $[\mu a.T/a].T$. Two recursive types are equivalent if their *infinite unfoldings* are equivalent.

Well-formedness

$\boxed{\Gamma \vdash U \text{ WF}}$ In environment Γ, type U is well-formed. That is, all of U's type variables are listed in Γ.

$$(\textsc{Tvar-wf})\ \ \Gamma, a \vdash a \text{ WF} \qquad (\textsc{Nil-wf})\ \ \Gamma \vdash \textbf{nil} \text{ WF}$$

$$(\textsc{Rec-wf})\ \frac{\Gamma \vdash T_1 \text{ WF} \quad \ldots \quad \Gamma \vdash T_n \text{ WF}}{\Gamma \vdash \{x_1 : T_1, \ldots, x_n : T_n\} \text{ WF}} \qquad (\mu\textsc{-wf})\ \frac{\Gamma, a \vdash T \text{ WF}}{\Gamma \vdash \mu a.T \text{ WF}}$$

$$(\textsc{Arrow-wf})\ \frac{\Gamma \vdash T \text{ WF} \qquad \Gamma \vdash U \text{ WF}}{\Gamma \vdash T \to U \text{ WF}} \qquad \frac{\Gamma, \bar{a} \vdash T \text{ WF}}{\Gamma \vdash \forall \bar{a}.T \text{ WF}}$$

Subtyping

Subtyping is defined by a deduction system for judgements of the form

$\boxed{S \le T'}$ Type S is a subtype of type T.

Meaning: "Whereever a term of type T is required, a value of type S can also be passed."

This is expressed in the *subsumption rule* for type assignments (repeated later).

$$(\textsc{Sub})\ \frac{\Gamma \vdash E : S \qquad S \le T}{\Gamma \vdash E : T}$$

Subtyping is reflexive and transitive:

$$(\textsc{Refl})\ \ U \le U \qquad (\textsc{Trans})\ \frac{T_1 \le T_2 \qquad T_2 \le T_3}{T_1 \le T_3}$$

Subtyping Rules Continued

Records with "more fields" are subtypes of records with fewer fields.

$$(\textsc{Record})\ \frac{T_1 \le T_1' \quad \ldots \quad T_m \le T_m'}{\{x_1 : T_1, \ldots, x_m : T_m, \ldots, x_n : T_n\} \le \{x_1 : T_1', \ldots, x_m : T_m'\}}$$

Rule for type schemes:

$$(\textsc{Forall})\ \frac{T \le T'}{\forall \bar{a}.T \le \forall \bar{a}.T'}$$

Two recursive types $\mu a.S$ and $\mu b.T$ are in a subtype relationship if their inifinite unfoldings are.

Note: This is non-trivial to formalize and to type-check (but it's possible).

Subtyping for Function Types

For function types, we have the following rule:

$$(\textsc{Arrow})\ \frac{T' \leq T \qquad U \leq U'}{T \to U \leq T' \to U'}$$

Note that the subtyping relationship is *reversed* in the function arguments!

Why is this rule required?

One (somewhat loosely) calls this rule the *contravariance rule* for function subtyping, because subtyping is reversed for function arguments.

Structural Subtyping vs Subtyping by Declaration

We have seen an instance of *structural subtyping*, where a type is a subtype of another purely because of the structure of the two types.

Many programming languages use instead *subtyping by declaration*, where the subtyping relationship is explicitly declared.

In these languages a type declaration introduces a new type, with a given set of fields and a given (set of) supertype(s).

Example: Objects with "head" field, infinite lists:

```
type Headed [a] = { head: a}
type Stream [a] extends Headed [a] = {
  head: a, tail: () → Stream [a]
}
```


Type Aliases

In Mini-Funnel, a type definition is seen just as an abbreviation which can always be replaced by its right-hand side. (That's why we don't need to have type definitions in the abstract syntax).

Example:

```
type Headed [a] = { head: a}
type Stream [a] = μs.{
  head: a, tail: () → s
}
```

Note that

- We express recursion through μ-types.
- Stream[a] is a subtype of Headed[a] even without an explicit extends clause.

Which scheme is preferable?

Type Assignment for Expressions

$\boxed{\Gamma \vdash E : U}$ In environment Γ, term E has type U (where U can be **nil**).

$$(\textsc{Taut})\ \Gamma, x : T \vdash x : T \qquad (\textsc{Sub})\ \frac{\Gamma \vdash E : S \qquad S \leq T}{\Gamma \vdash E : T}$$

$$(\textsc{Sel})\ \frac{\Gamma \vdash E : \{x : T\}}{\Gamma \vdash E.x : T} \qquad (\textsc{Tapp})\ \frac{\Gamma \vdash E : \forall \overline{a}.T \qquad \Gamma \vdash \overline{S}\ \textsc{wf}}{\Gamma \vdash E[\overline{S}] : [\overline{S}/\overline{a}]T}$$

$$(\textsc{Apply})\ \frac{\Gamma \vdash E : T \to U \qquad \Gamma \vdash F : T}{\Gamma \vdash E\ F : U}$$

$$(\textsc{Fork})\ \frac{\Gamma \vdash E : U \qquad \Gamma \vdash F : \mathbf{nil}}{\Gamma \vdash E \,\&\, F : U} \qquad (\textsc{Nil})\ \Gamma \vdash \mathbf{nil} : \mathbf{nil}$$

$$(\textsc{Def})\ \frac{\Gamma, \Sigma \vdash D : \Sigma \qquad \Gamma, \Sigma \vdash E : U}{\Gamma \vdash (\mathbf{def}\ D \,;\, E) : U}$$

Notation The , operator on environments and records is assumed to be associative and commutative.

Furthermore, the operands Γ, Σ of a composition Γ, Σ are required to have disjoint domains.

Type Assignment for Definitions

$\boxed{\Gamma \vdash D : \Sigma}$ In environment Γ, the definitions D are well-typed and generate environment Σ. Σ binds exactly the functions defined in D. Since definitions can be recursive, it is assumed that all functions in Σ are already in Γ.

$$(\textsc{Eqn})\ \frac{\Gamma \vdash L : \Sigma \,;\, \Delta \qquad \Sigma = I : \forall \overline{a}.T \to U, \Sigma' \qquad \Gamma, \Delta \vdash E : U}{\Gamma \vdash L = E : \Sigma}$$

$$(\textsc{Empty})\ \Gamma \vdash \epsilon : \epsilon \qquad (\textsc{Concat})\ \frac{\Gamma \vdash D_1 : \Sigma_1 \qquad \Gamma \vdash D_2 : \Sigma_2}{\Gamma \vdash D_1, D_2 : \Sigma_1 \wedge \Sigma_2}$$

Notation

- $\wedge$ is deep intersection, defined on types and environments consisting only of value bindings as follows:

$$\begin{array}{llll}
S \wedge T & = & S & \text{if } S = T \\
 & = & \{\Gamma \wedge \Sigma\} & \text{if } S = \{\Gamma\}, T = \{\Sigma\} \\
 & & \text{is undefined otherwise} & \\
(\Gamma \wedge \Sigma)(x) & = & \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Sigma) \\
 & = & \Sigma(x) & \text{if } x \in \mathrm{dom}(\Sigma) \setminus \mathrm{dom}(\Gamma) \\
 & = & \Gamma(x) \wedge \Sigma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Sigma)
\end{array}$$

- We sometimes write bindings of the form $I : T$, where I is a qualified identifier. The environment $I.x : T$ is taken to be equal to $I : \{x : T\}$.

Type Assignment for Left-hand Sides

$\boxed{\Gamma \vdash L : \Sigma \,;\, \Delta}$ In environment Γ, left-hand-side L generates function environment Σ and local environment Δ.

$$(\textsc{Fun})\ \frac{\Gamma, \overline{a} \vdash T\ \textsc{wf} \qquad \Gamma, \overline{a} \vdash U\ \textsc{wf}}{\Gamma \vdash (I[\overline{a}](x : T) : U) : (I : \forall \overline{a}.T \to U) \,;\, (\overline{a}, x : T)}$$

$$(\textsc{Join})\ \frac{\Gamma \vdash L : \Sigma \,;\, \Delta \qquad \Gamma \vdash L' : \Sigma' \,;\, \Delta' \qquad \Sigma' = (I : \forall \overline{a}.T \to \mathbf{nil})}{\Gamma \vdash L \,\&\, L' : \Sigma \wedge \Sigma' \,;\, \Delta, \Delta'}$$

Derived Constructs

Add **val** definitions to Mini-Funnel:

$$\text{Term} \quad E, F \quad = \quad ... \mid \textbf{val}\ x = E\ ;\ E'$$

Typing rule for **val**:

Add sequencing to Mini-Funnel:

$$\text{Term} \quad E, F \quad = \quad ... \mid E\ ;\ E'$$

Typing rule:

Type Inference

The presented type system needs a lot of explicit type information.

Can we infer this information?

Constraints, as in the Hindley/Milner system don't work here so well:

- Because of subsumption, we get subtyping constraints $S \leq T$ rather than equality constraints $T = U$. Hence, unification is not applicable.
- Even without subtyping, ∀-quantifiers would make unification undeciable.

Alternative: Use *local* type inference, to infer some type annotations.

Local type inference computes types by propagating information from some part of the tree to neighboring parts.

Example: Given

```
def f(x: Int) = x + x
```

infer f's result type to be Int.

Example: Given

```
x: Int, xs: List[Float]
Int ≤ Float
cons: ∀a.(a, List[a]) → List[a]
```

infer the missing type parameter in cons (x, List[xs]) to be [Float]:

```
cons [Float] (x, List[xs])
```

Generally, infer missing type parameters to be those types which make the actual arguments match the formal parameters and which minimize the result type.

An error results if no optimal type parameters exist.

Example:

Given

```
type ListVisitor [a,b] = {
  def Nil: b
  def Cons (x: a, xs: List [a]): b
}
type List [a] = {
  def match [b] (v: ListVisitor [a, b]): b
}
```

infer that in the definition

```
def append (xs: List[String], ys: List[String]) = {
  xs.match {
    def Nil = ys
    def Cons (x, xs1) = List.Cons (x, append (xs, ys))
  }
}
```

the following holds:

- The record argument of xs.match is a ListVisitor[String,b] for some unknown b.
- Therefore, the full types of the Nil and Cons fields are:

  Nil: b
  Cons: (x: String, xs: List [String]) → b

  (Parameter types are inferred).
- Therefore, the result type of the visitor is List[String], which is then also the type parameter for match.

Fully typed output of type inference:

```
def append (xs: List[String], ys: List[String]): List[String] = {
  xs.match [List[String]] {
    def Nil: List[String] = ys
    def Cons (x: String, xs1: List[String]): List[String] =
      List.Cons [String] (x, append (xs, ys))
  }
}
```


See also:

- Benjamin Pierce and David Turner; Local Type Inference; Proc. ACM Symposium on Principles of Programming Languages, 1996. (their techniques solve the first two examples).
- Our current research (solves all three).