Abstract

Notes on a formal semantics for dependently typed X10, and generically typed X10.

1. Judgements

Here is the syntax for CFJ, converted to the new X10 1.7 style syntax. *(Note: we should convert the OOPSLA paper to this syntax as well.)*

Need to answer reviewer's question: Does the invariant associated with a class C change when that associated with the class D that it extends, changes? (Yes.) Need to figure out what this means for separate compilation. Should still be able to compile separately. It is just the case now that when you link a new class in, the invariant for this class will depend on the invariant that you have loaded for its superclasses. Java and X10 do nothing to keep compile-time types consistent with link-time types.

$$\begin{array}{rrcl}
\text{(C Term)} & \mathtt{t} & ::= & \mathtt{x} \mid \mathtt{self} \mid \mathtt{t.f} \\
 & & & \mid \mathtt{new\,C}(\overline{\mathtt{t}}) \mid \mathtt{g}(\overline{\mathtt{t}}) \\
\text{(Const.)} & \mathtt{c,d} & ::= & \mathtt{true} \mid \mathtt{p}(\overline{\mathtt{t}}) \mid \mathtt{t} = \mathtt{t} \\
 & & & \mid \mathtt{c,c} \mid \mathtt{val\,x:T;\ c} \\
\text{(Class)} & \mathtt{L} & ::= & \mathtt{class}\,C(\overline{f}:\overline{T})\{c\}\ \mathtt{extends}\ T\ \{\overline{M}\} \\
\text{(Method)} & \mathtt{M} & ::= & \mathtt{def\,m}(\bar{x}:\overline{T})\{c\}:T = e; \\
\text{(Exp.)} & \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\,C}(\overline{\mathtt{e}}) \\
 & & & \mid \mathtt{e\ as\ T} \\
\text{(Type)} & \mathtt{S,T,} & & \\
 & \mathtt{U,Z} & ::= & \mathtt{C}\{\mathtt{d}\}
\end{array}$$

We will use the abbreviation $\mathtt{val}x : S, y_1 : T_1, \ldots, y_n : T_n; T$ for $\mathtt{val}\ x : S; \mathtt{val}\ y_1 : T_1; \ldots; \mathtt{val}\ y_n : T_n; T$.

In the rest of this paper we will assume give some fixed but unknown constraint system $\mathcal{D}$. We will assume that the program P is written using constraints from $\mathcal{D}$, and further that classes defined in P do not have a cyclic inheritance struture. From P and $\mathcal{D}$ we also generate a new constraint system O, the constraint system of *objects* over P and $\mathcal{D}$ as follows.

Below, C,D range over names of classes in P, f over field names, m over method names, T over types, c over constraints in the underlying data constraint system $\mathcal{D}$.

$$\begin{array}{rrcl}
\text{(C Term)} & \mathtt{t} & ::= & \mathrm{C} \\
\text{(Constraint)} & \mathtt{c,d} & ::= & C\mathtt{<:}D \mid f:T \in C \mid \\
 & & & m(\bar{x}:\overline{T}), c \rightarrow T \in C
\end{array}$$

The constraint system satisfies the following axioms and inference rules: the subtype relation `<:` is reflexive and transitive and $C\mathtt{<:}D$ whenever C extends D in the given program.

For every field $f : T$ defined in class C, it is the case that $\vdash_O f : T \in C$. For every method $m(\mathtt{var}x : \overline{T})\{c\} : T = \{\ldots\}$ in class D, it is the case that $\vdash_O m(\bar{x} : \overline{T}), c \rightarrow T \in C$. Finally:

$$\frac{\Gamma \vdash_O \phi \in C \qquad \Gamma \vdash_O C\mathtt{<:}D}{\Gamma \vdash_O \phi \in D}\ (\textsc{Inh})$$

The constraint system $\mathcal{C}$ is the conjunction $\mathcal{D}, O$.

2. Rules

First, for a type environment Γ, we define the *constraint projection*, $\sigma(\Gamma)$ thus:

$\sigma(\varepsilon) = \mathtt{true}$
$\sigma(x : C\{c\}, \Gamma)$=(val x:C;(x=self,c)), $\sigma(\Gamma)$
$\sigma(c, \Gamma)$ = c, $\sigma(\Gamma)$

In general, existential quantification is more general than substitution, and since we have to use it anyway, we may as well avoid using substitutions.

2.1 Judgements

The following judgements will be defined. In all of them Γ is a well-typed context.

- Given Γ, the type T is well-formed: $\Gamma \vdash T\ \mathtt{type}$
- Given Γ, the type S is a subtype of T: $\Gamma \vdash S \mathtt{<:} T$
- Given Γ, the expression e is of type T: $\Gamma \vdash e : T$
- Given Γ, the method M is well-defined for the class C: $\Gamma \vdash M$ OK in C
- Given Γ, the field M is well-defined for the class C: $\Gamma \vdash f : T$ OK in C
- Given Γ, the class definition Cl is well-defined: $\Gamma \vdash Cl$ OK

In what follows we will sometimes think of the family of five judgements as a single judgement $\Gamma \vdash \phi$, where ϕ ranges over the formulas $T\ \mathtt{type}$, $e : T$, $S\mathtt{<:}T$, $f : T$ OK in C, M OK in C, and C OK.

In defining these judgements we will use $\Gamma \vdash_{\mathcal{C}} c$, the judgement corresponding to the underlying constraint system. For simplicity we define $\Gamma \vdash c$ to mean $\sigma(\Gamma) \vdash_{\mathcal{C}} c$.

Now, these judgements need to satisfy certain properties:

- $\Gamma \vdash T\mathtt{type}$ whenever $\Gamma \vdash e : T$.

  That is, if we can conclude that `e` has type `T` (under certain assumptions), then under those assumptions we must be able to conclude that `T` is well-defined.
- $\Gamma \vdash S\ \mathtt{type}$ and $\Gamma \vdash T\ \mathtt{type}$ whenever $\Gamma \vdash S\mathtt{<:}T$.
- If $\Gamma \vdash e : T$ and x is a variable occurring free in e then for some type U, $\Gamma \vdash x : U$. That is, all free variables on the RHS are actually defined by the LHS.

Keeping in mind these requirements, the rules are as follows. Below whever we use the assertion "x free" in the antecedent of a rule we mean that x is not free in the consequent of the rule.

2.2 Structural and Logical Rules

All the judgements are intuitionistic. In particular this means that all constraint systems satisfy the rules and axioms of inference below.

2.3 Well formedness rules

We use the judgement for well-typedness for expressions to represent well-typedness for constraints. That is, we posit a special type `o` (traditionally the type of propositions), and regard constraints as expressions of type `o`. Further, we change the formulation slightly so that there are no constraints of the form `p(t1,..., tn)`; rather instance method invocation syntax is used to express invocation of pre-defined constraints. This logically leads to the step of simply marking certain classes as "predicate" classes – all the (instance) methods of these classes whose return type is `o` then correspond to "primitive constraints."

Syntactically, we will continue to use the symmetric syntax `p(t1,..., tn)` rather than `t1.p(t2,..., tn)`.

(The alternative is to introduce static methods, and static method invocations in the expression language. This is not too difficult, but it is annoying to have to repeat most of the formulation of instance methods, and to have one more case to prove.)

So this means that the only cases left to handle are all the simple ones, expressing the availability of certain constants and operators at type `o` (see Figure 2).

$$\Gamma, c \vdash c \quad (\textsc{Id}) \qquad \frac{\Gamma \vdash c \qquad \Gamma, c \vdash d}{\Gamma \vdash d} \quad (\textsc{Cut})$$

$$\frac{\begin{array}{c}\Gamma \vdash \phi \qquad \Gamma \vdash \phi \\ \Gamma \vdash S\ \texttt{type} \qquad x \notin \texttt{var}(\Gamma)\end{array}}{\Gamma, x : S \vdash \phi}\ (\textsc{Weakening-1}) \qquad \frac{\Gamma \vdash \phi \qquad \Gamma \vdash c : o}{\Gamma, c \vdash \phi}\ (\textsc{Weakening-2}) \qquad \frac{\Gamma, \psi_0, \psi_1 \vdash \phi}{\Gamma, (\psi_0, \psi_1) \vdash \phi}\ (\textsc{And-L})$$

$$\frac{\Gamma \vdash \psi_0 \qquad \Gamma \vdash \psi_1}{\Gamma \vdash \psi_0, \psi_1}\ (\textsc{And-R}) \qquad \frac{\Gamma \vdash \phi[t/x] \qquad \Gamma \vdash t : S}{\Gamma \vdash \texttt{val}\ x; \phi}\ (\textsc{Exists-R}) \qquad \frac{\Gamma, x : S, \psi \vdash \phi \qquad (x \text{ fresh})}{\Gamma, \texttt{val}\ x : S; \psi \vdash \phi}\ (\textsc{Exists-L})$$

Figure 1. Logical rules

$$\Gamma \vdash \texttt{true} : o \quad (\texttt{true}) \qquad \frac{\Gamma \vdash t_0 : T_0 \qquad \Gamma \vdash t_1 : T_1 \qquad (\Gamma \vdash T_0 <: T_1 \vee \Gamma \vdash T_1 <: T_0)}{\Gamma \vdash t_0 = t_1 : o}\ (\textsc{Equals}) \qquad \frac{\Gamma \vdash c_0 : o \qquad \Gamma \vdash c_1 : o}{\Gamma \vdash (c_0, c_1) : o}\ (\textsc{And})$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash c[t/x] : o}{\Gamma \vdash \texttt{val} x : T; c : o}\ (\textsc{Some}) \qquad \frac{C \in \overline{C} \qquad \Gamma, \texttt{self} : C \vdash c : o}{\Gamma \vdash C\{c\}\ \texttt{type}}\ (\textsc{type})$$

Figure 2. Well formedness rules

3. Type inference rules

3.1 Expression typing judgement

Now we consider the rule for method invocation. Assume that in a type environment Γ the expressions $e_{0\ldots n}$ have the types $T_{0\ldots n}$. Since the actual values of these expressions are not known, we shall assume that they take on some fixed but unknown values $z_{0\ldots n}$ of type $T_{0\ldots n}$. Now for z_0 as receiver, let us assume that the type $T_0 \equiv C\{d\}$ has a method named `m` with signature $z_{1\ldots n} : Z_{1\ldots n}, c \rightarrow U$. If there is no method named `m` for the class `C` then this method invocation cannot be type-checked. Without loss of generality we may assume that the parameters of this method are named $z_{1\ldots n}$, since we are free to choose variable names as we wish.) Now in order for the method to be invocable, it must be the case that the types $T_{1\ldots n}$ are subtypes of $Z_{1\ldots n}$. (Note that there may be no occurrences of `this` in $Z_{1\ldots n}$; they have been replaced by z_0.) Further, it must be the case that for these parameter values, the constraint c is entailed. Given all these assumptions it must be the case that the return type is U — with all the parameters $z_{0\ldots n}$ existentially quantified.

3.2 Class OK judgement

The following rule is modified from what we had in the paper to ensure that all the types are well-formed (under the assumption `this:C`. Note that the variables $\overline{x}$ are permitted to occur in the types $T, \overline{T}$, hence their typing assertions must be added to Γ.

3.3 Subtype judgement

Note that `self` may be free in c, d and must be declared.

$$\frac{\begin{array}{c}\Gamma \vdash C <: D, C\{c\}\ \texttt{type}, D\{d\}\ \texttt{type} \\ \Gamma, \texttt{self} : C, c \vdash d\end{array}}{\Gamma \vdash C\{c\} <: D\{d\}}\ (\textsc{Dep type})$$

$$\Gamma, x : T \vdash x : T \qquad \text{(T-VAR)} \qquad \frac{\Gamma \vdash e : U \qquad \Gamma \vdash T \ \mathtt{type}}{\Gamma \vdash e \ \mathtt{as} \ T : T} \ \text{(T-CAST)} \qquad \frac{\Gamma \vdash e : S \qquad \mathtt{fields}(S)_i = f : U}{\Gamma \vdash e.f : \mathtt{val} \ \mathtt{this} : S ; U} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : T, \overline{e} : \overline{T} \\ \mathtt{mtype}(T, m, z) = \overline{z} : \overline{Z}, c \to U \qquad z, \overline{z} \text{fresh} \\ \Gamma, z : T, \overline{z} : \overline{T} \vdash \overline{T} <: \overline{Z} \\ \Gamma, z : T, \overline{z} : \overline{T} \vdash c \end{array}}{\Gamma \vdash e.m(\overline{e}) : \mathtt{val} z : T, \overline{z} : \overline{T} ; U} \qquad \text{(T-INVK)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \overline{e} : \overline{T} \qquad \mathtt{fields}(C, z) = \overline{f} : \overline{Z} \qquad z, \overline{z} \text{fresh} \\ \Delta \equiv \Gamma, z : C, \overline{z} : \overline{T}, z.\overline{f} = \overline{z} \\ \Delta \vdash \overline{T} <: \overline{Z} \qquad \Delta \vdash \mathit{inv}(C, z) \end{array}}{\Gamma \vdash \mathtt{new} \ C(\overline{e}) : C\{\mathtt{val} \ z : C, \overline{z} : \overline{T} ; z.\overline{f} = \overline{z}, \mathtt{self} = z, \mathit{inv}(C, z)\}} \ \text{(T-NEW)}$$

Figure 3. Type judgement

$$\frac{\begin{array}{c} \mathtt{this} : C, \overline{x} : \overline{T} \vdash T \ \mathtt{type}, \overline{T} \ \mathtt{type} \\ \mathtt{this} : C, \overline{x} : \overline{T}, c \vdash e : U, U <: T \end{array}}{\mathtt{def} \ m(\overline{x} : \overline{T})\{c\} : T = e; \ \text{OK in } C} \qquad \text{(METHOD OK)} \qquad \frac{\begin{array}{c} \overline{M} \ \text{OK in } C \qquad \mathtt{this} : C \vdash \overline{T} \ \mathtt{type} \\ \vdash T \ \mathtt{type} \qquad (\mathit{acyclicity}) \end{array}}{\mathtt{class} \ C(\overline{f} : \overline{T}) \ \mathtt{extends} \ T\{\overline{M}\} \ \text{OK in } C} \quad \text{(CLASS OK)}$$

Figure 4. Method and Class OK