Formalizing Dependent Types for X10

(Draft Version 0.04)
(Please do not cite)
(Send comments to vsaraswa@us.ibm.com.)

April 19 2006

Abstract

We formalize the basic ideas of dependent-types for Java like languages (introduced in [**??DependentTypes??**]), in the context of FX10, a Featherweight version of X10 [**??X10??**]. For now, we focus on a sequential language; distribution and concurrency constructs *a la* X10 are to be added later. This note should be taken as a companion to [**??X10-concur??**] which formalizes the dynamic semantics for FX10. It is intended to provide the basis for a place-based type-system for scalars and arrays.

1 Introduction

2 Basic FX10

The abstract syntax for FX10 is specified in Table **??Table:AST??**. We consider a parametric version of the language, with an underlying constraint system $\mathcal{C}$ [**??cccc??**] being used to specify dependent types.

We follow [**??FJ??**,**??MJ??**] in our treatment. Meta-variables `C`, `D`, `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; `x`, `y`, `z` range over parameter and local variable names; other meta-variables are specified in Table **??Table:AST??**.

We write $\bar{e}$ as shorthand for $\mathtt{e}_1, \ldots, \mathtt{e}_\mathtt{n}$ (comma-separated sequence); this sequence may be empty (`n=0`). Similarly for $\bar{x}$. $\bar{\mathtt{M}}$ and $\bar{\mathtt{K}}$ are similar except that no commas separate the items in the sequence. We use the obvious abbreviation: $\bar{\mathtt{f}} : \bar{\mathtt{X}}$, abbreviates $\mathtt{f}_1 : \mathtt{X}_1, \ldots, \mathtt{f}_\mathtt{n} : \mathtt{X}_\mathtt{n}$ (`n` may be zero). $\mathtt{var}\ \bar{\mathtt{g}} : \bar{\mathtt{Y}}$ abbreviates the sequence $\mathtt{var}\ \mathtt{g}_1 : \mathtt{Y}_1, \ldots, \mathtt{g}_\mathtt{n} : \mathtt{Y}_\mathtt{n}$ if $\mathtt{n} > \mathtt{1}$ and the empty sequence otherwise. Empty parameter sequences may be omitted (like Scala, unlike Java).

(Classes)	CL	::=	$\mathtt{class\ C(\bar{f} : \bar{X},\ var\ \bar{g} : \bar{Y})extends\ D\{K\ \bar{M}\}}$
(Constructor)	K	::=	$\mathtt{def\ C(\bar{u} : \bar{U}, \bar{f} : \bar{X}, \bar{g} : \bar{Y}\ \&c):T\ =\ \{super(\bar{u});\ this.\bar{f}, \bar{g} = \bar{f}, \bar{g};\}}$
(Method)	M	::=	$\mathtt{def\ m(\bar{x} : \bar{X}\ \&c):T\ =\ e}$
(Expression)	e,r,s	::=	$\mathtt{null} \mid \mathtt{n} \mid \mathtt{x} \mid \mathtt{new\ C(\bar{e})}$
			$\mid \{\mathtt{val}\ \bar{\mathtt{x}} : \bar{\mathtt{X}} = \bar{\mathtt{e}};\ \mathtt{e}\} \mid \mathtt{e;e}$
			$\mid \mathtt{x.f} \mid \mathtt{x.f{=}e} \mid \mathtt{x.m(\bar{x})} \mid \mathtt{(T)\ e}$
			$\mid \mathtt{c?e:e} \mid \ldots$
(Type)	T,U,V,X,Y	::=	$\mathtt{C(\&c)} \mid \mathtt{nullable\ C(\&c)}$
(Constraint)	c,d	::=	$\mathtt{true} \mid \mathtt{c\&c} \mid \mid \text{ce==ce} \mid \text{ce !=ce} \mid (\exists \mathtt{x} : \mathtt{T})\,\mathtt{c}\ldots$
(Term)	ce	::=	$\mathtt{self} \mid \mathtt{f} \mid \mathtt{x} \mid \ldots$

Table 1: Abstract syntax for FX10.

The phrase “&c” is called a *where clause*. We abbreviate &true to the empty string. Similarly $\mathtt{this.\bar{f}=\bar{e};}$ abbreviates $\mathtt{this.f_1} = \mathtt{e_1};\ldots\mathtt{this.f_n} = \mathtt{e_n}$. $\{\mathtt{val}\ \bar{\mathtt{x}} : \bar{\mathtt{X}} = \bar{\mathtt{e}};\}$ abbreviates $\{\mathtt{val}\ \mathtt{x_1} : \mathtt{X_1} = \mathtt{e_1};\ldots;\mathtt{x_n} : \mathtt{X_n} = \mathtt{e_n};\}$. Sequences of field declarations, parameter declarations, local variable declarations, are assumed to not contain any duplicates. Sequences of constructors in a class must not contain two constructors with the same sequence of parameters types; similarly for methods. (FX10 permits *ad hoc* polymorphism.)

Expressions. For expressions, we assume the following precedence order (from less tight to more tight): sequencing, type-cast, assignment, conditional, field invocation, method invocation.

We also reserve the local variable name “this” and “self”. That is, no program may define a local variable or parameter named this or self.

We note that, somewhat unusually, field selection, assignment, method invocation and constructor invocation take constants as arguments, rather than expressions. This is necessary because we need a name for the arguments so that the name can be substituted for the formal argument in the resulting type. The version of these operations which takes arbitrary expressionas as arguments can be obtained by combining with the local variable combinator. Thus, e.m(e1) is simply {val x:X=e;x}.m({val y:Y=e1;y}), where x and y are new local variables, and the type of the expressions e and e1 is X and Y respectively. Below, when writing actual programs we shall feel free to use the e.m(e1) form.

The two-armed conditional expression c?e:e does not take an arbitrary expression as a test; rather it takes as argument a constraint which must satisfy the property that its negation is also a constraint. (Note that we do not require that constraints are closed under negation in general.) c?e:e1 is best thought of as a “typecase” expression. It permits the compiler to reason conditionally about the expression, by propagating the constraint down the positive branch and its negation down the negative branch.

Types. We reserve the class names “Object” and “int”.[1]

A type is of the form C(&c) where c is a constraint. Intuitively, a type C(&c) is the type of all objects that are instances of C and satisfy the condition c. Note that if the condition c is unsatisfiable, then the type is empty. Variables/parameters cannot be declared at empty types.

From the abbreviation rules above, the type C(&true) may be abbreviated to C. c may contain references to parameters and variables visible at the point of declaration of the type (including this), and the special variable self which refers to the object to which the type applies. The fields of C may occur unqualified in c, they are presumed to be qualified with the special variable self.[2]

The special variable this may *not* be referenced in a type in the definition of a constructor (e.g. in the constraint in the parameter list of the constructor, or in the return type of the constructor). This recognizes the fact that an object does not exist until it is created.

The type nullable C(&c) is the type C(&c) together with the special value null. Thus the type nullable C(&c) is never empty: if c is inconsistent it permits precisely the value null.

The terms ce in a constraint are drawn from an underlying constraint system, $\mathcal{C}$ [**??cccc??**]. *For now we take the constraint system to be fixed, but it makes sense to permit the programmer to extend the constraint system with new value types, and operations over them, provided that a constraint solver is supplied for these types.*

Recall that for readability, we permit the expression self.f to be abbreviated to f in the constraint c of a type C(&c). self is often absent in types. It is particularly useful in *singleton types*, e.g. Point(&self=p) which is satisfied by any object that is an instance of Point and is the same as p.

[1] In a subsequent version of this document we will introduce value types, and then int will be just another value type, defined with native methods.

[2] Note that in general this is different from self. For instance the type B(& f = this.g) appearing in the body of the definition of the class A is the type of all instances of B whose f field has the same value as the g field of the current A object. (It is the same as the type B(& self.f = this.g).)

We also permit the shorthand $\mathtt{C(\overline{ce})}$ for $\mathtt{C(\&\overline{f} = \overline{ce})}$ where $\overline{\mathtt{f}}$ is the textual order enumeration of the `val` fields of `C`. If the class has no field, then we use the shorthand `C(ce)` for `C(&self=ce)`. Thus `int(0)` is satisfied precisely by the value `0`.

Finally, we permit the shorthand $(\exists \mathtt{x} : \mathtt{T})\mathtt{C}(\&\mathtt{c})$ for $\mathtt{C}(\exists \mathtt{x} : \mathtt{T})\mathtt{c})$.

In a later version of this document we will introduce type definitions. This will let us use, for instance, the abbreviation `nat` *for the type* `int(&self >= 0)`.

Program. A *program* is a pair of a set of classes and an expression. For simplicity we shall leave the set of classes implicit. We shall assume that every class name used in the program (except `Object`) is defined exactly once in the program. We assume that the class hierarchy, defined by $\leq$ in the next section is acyclic (anti-symmetric).

2.1 Remarks on the syntax

In a class declaration, $\overline{\mathtt{f}}$ represents the val (final) fields of the class, $\overline{\mathtt{g}}$ the var (mutable) fields.[3]

After Scala [**??scala??**], we have chosen the ML-style variable declaration (type comes after the variable, and is separated by a colon), as opposed to conventional Java-style declarations, so that the return type of a method may appear after the declaration of the parameters of the method. This permits parameters to appear in the return type, while respecting the "define before use" meta-rule. This syntax also permits types (for parameters, variables and methods) to be optional, while still retaining readability. Note that the language above does not permit mutable constructor or method parameters.

In a constructor declaration, `c` is a constraint on the constructor parameters $\overline{\mathtt{x}}$ that must be true for the constructor to be invoked. A constructor may specify the return type (useful for specifying constraints on the fields of the object returned by the constructor). In a method definition, `c` is a constraint on the parameters $\overline{\mathtt{x}}$ and the final fields of the object on which the method is being invoked that must be true for this method to be invoked. In a type specification, `c` is a constraint on the final fields of the class `C`. A constraint is a boolean valued expression over final fields and local variables written using the operations provided in the underlying constraint system.

The abstract syntax differs from the syntax in [**??DependentTypes??**] in many notational respects. It differs substantively in that there is no distinction between parameters of classes and fields, and parameters of methods and arguments to the method. All arguments to a method are considered final. There are no implicit parameters for classes. All final fields of a class may be used in defining a dependent type on that class.

2.2 Example

Example 2.1 (List) Consider the class `List`. We shall use generic syntax for type parameters; this will be formalized in a subsequent version of this note in a fashion similar to [**??FJ??**]. For now, the reader should understand generic syntax in the spirit of [**??FJ??**].

```
class List<X>( n:int(&self>=0),
             var node: nullable X,
                 rest: nullable List<X>(n-1)) {
  /** Returns the empty list. Defined only when the parameter n
      has the value 0. Invocation: new List(0)<X>().
   */
  def List(n:int(&self>=0), node: nullable X, rest: nullable List<X>(n-1)) = {
    super(); this.n=n; this.node=node; this.rest=rest;
  }
```

[3] The addition of `var` constructor and method parameters and local variables is routine and omitted for brevity. Similarly for initializers of fields, static fields and methods. We may add exceptions later since they play an integral part in the semantics of concurrent constructs.

```
  def makeList:List<X>(0)=new List(0,null, null)
  def makeList(node:X):List<X>(1)=new List(1,node,makeList)
  def makeList{node:X, rest:nullable List<X>):List<X>(rest.n+1)=
   rest==null ? makeList(node):new List(rest.n+1,node,rest)

  def append(arg:List<X>):List<X>(n+arg.n)  =
       (n == 0) ? arg : new List<X>(node, rest.append(arg))
  def rev : List<X>(n) = rev(new List<X>())
  def rev(arg:List<X>):List<X>(n+arg.n) =
       (n == 0) ? arg : rest.rev(new List<X>( node, arg))
  /** Return a list of compile-time unknown length, obtained by filtering
      this with f. */
  def filter(f: fun<X,boolean>):List<X> =
       (n==0) ? this
       : (f(node) ?
         new List<X>(node, rest.filter(f));
          : rest.filter(f))
  /** Return a list of m numbers from o..m-1. */
  def gen(m:int(&self>= 0)):List<int(&self>=0)>(m) = gen(0,m)
  /** Return a list of (m-i) elements, from i to m-1. */
  def gen(i:int(&self>=0), m:int(&self>=i)):List<int(&self>=0)>(m-i) =
     (i == m)? new List<nat>() : new List<nat>(i, gen(i+1,m))
}
```

The class `List` has three fields, the `val` field `n`, and the `var` fields `node` and `rest`. `rest` is required to be a `List` whose `n` field has the value `this.n-1`. `n` is required to be non-negative – this is checked statically for each constructor.

Three constructors are provided. The first takes no arguments and returns instances of `List` which satisfy the constraint `n=0`. The second takes a single argument and returns a list of size `1`. The third takes two arguments, `node` (of type `X`) and `rest` of type `List<X>`, and returns a list of size `rest.n+1`. That is, the length of the list returned depends on the value of a parameter to the constructor.

Functions that append one list to another or that reverse a list can be defined quite naturally. In both cases the size of the list returned is known statically (as a function of the size of the list and the argument to the method). The example also illustrates a method `filter` which returns a list whose size cannot be known statically (it depends on properties of the argument function `f` which are not captured statically).

The `gen` methods[4] illustrate "`self`"-constraints. The first `gen` method takes a single argument `m` that is required to be a non-negative `int`. The second `gen` method illustrates that the type of a parameter can depend on the value of another parameter: `m` is required to be no less than `i`. This assumption are necessary in order to guarantee that the result type of the method is not empty, that is, to guarantee `m-i >= 0`. □

Note: Language extensions to be made to get a fuller subset of X10*:*

- *Add multiple constructors.*
- *Permit arbitrary code in constructors.*
- *Add static state – or at least distinguish between objects and traits.*
- *Permit fields to be overridden?*
- *Add functions.*
- *Add exceptions.*

[4] These should really be `static`.

- *Add places.*
- *Add async, finish, future.*
- *Add regions, distributions and arrays.*
- *Add generics with declaration-time covariance/contravariance/invariance, a la Scala. Generics should be instantiable with arbitrary value types.*
- *Consider adding generic values (universal quantification).*
- *Arrays should be generic.*

2.3 Type system

A *typing environment*, Γ, is a collection of variable typings $x : T$ together with zero or more constraints on variables. The constraint system satisfies the axiom:

$$\Gamma, \mathtt{x} : \mathtt{C}(\&\mathtt{c}) \vdash \mathtt{c}[\mathtt{x}/\mathtt{self}]$$

Thus, for instance, we have:

$$\mathtt{x} : \mathtt{List}(\&\mathtt{n} = \mathtt{3}) \vdash \mathtt{x.n} = \mathtt{3}$$

(asuming `n` is a field in `List`, and hence an abbreviation for `self.n`).

Typing judgements for expressions are of the form $\Gamma \vdash \mathtt{e} : \mathtt{T}$, read as "Under the assumptions Γ, `e` has type `T`." We use sequence notation in the obvious way: $\Gamma \vdash \bar{\mathtt{x}} : \bar{\mathtt{X}}$ is shorthand for the collection of typing judgements $\Gamma \vdash \mathtt{x}_1 : \mathtt{X}_1 \ldots \Gamma \vdash \mathtt{x}_n : \mathtt{X}_n$.

2.3.1 Subtyping rules

We will also use the sub-typing judgement $\Gamma \vdash \mathtt{X} \leq \mathtt{Y}$ on types, read as "Under the assumptions Γ the type `X` is a sub-type of `Y`.

$$\Gamma \vdash \mathtt{C} \leq \mathtt{C} \qquad \frac{\Gamma \vdash \mathtt{C} \leq \mathtt{D} \quad \Gamma \vdash \mathtt{D} \leq \mathtt{E}}{\Gamma \vdash \mathtt{C} \leq \mathtt{E}} \qquad \frac{\mathtt{class\ C(\ldots)\ extends\ D\ \ldots}}{\Gamma \vdash \mathtt{C} \leq \mathtt{D}} \qquad \frac{\Gamma \vdash \mathtt{C} \leq \mathtt{D} \quad \Gamma, \mathtt{c} \vdash \mathtt{d}}{\Gamma \vdash \mathtt{C}(\&\mathtt{c}) \leq \mathtt{D}(\&\mathtt{d})}$$

Under the assumptions of well-formed programs, this relation is acyclic.

2.3.2 Static semantics rules

$$\frac{}{\Gamma \vdash \mathtt{null} : \mathtt{nullable}\ \mathtt{T}}\ (\texttt{T-Null}) \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{e} : \mathtt{nullable}\ \mathtt{T}}\ (\texttt{T-Null-2})$$

$$\frac{}{\Gamma \vdash \mathtt{n} : \mathtt{int}(\mathtt{n})}\ (\texttt{T-Lit}) \qquad \frac{}{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}}\ (\texttt{T-Var})$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{x} : \mathtt{T}\\ \mathtt{f} : \mathtt{U} \in \mathtt{fields}(\mathtt{T})\\ \theta = [\mathtt{x}/\mathtt{this}]\end{array}}{\Gamma \vdash \mathtt{x.f} : \mathtt{U}\theta}\ (\texttt{T-Field-r}) \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{x} : \mathtt{T}\\ \mathtt{f} : \mathtt{U} \in \mathtt{varfields}(\mathtt{T})\\ \theta = [\mathtt{x}/\mathtt{this}]\\ \Gamma \vdash \mathtt{g} : \mathtt{V} \quad \Gamma \vdash \mathtt{V} \leq \mathtt{U}\theta\end{array}}{\Gamma \vdash \mathtt{x.f} = \mathtt{g} : \mathtt{U}\theta}\ (\texttt{T-Field-w})$$

$$\frac{\begin{array}{l}\Gamma \vdash \bar{\mathtt{y}} : \bar{\mathtt{Y}}\\ \mathtt{C}(\bar{\mathtt{z}} : \bar{\mathtt{Z}}\&\mathtt{c}) : \mathtt{T} \in \mathtt{ctor}(\mathtt{C})\\ \theta = [\bar{\mathtt{y}}/\bar{\mathtt{z}}]\\ \Gamma \vdash \bar{\mathtt{Y}} \leq \bar{\mathtt{Z}}\theta \quad \Gamma \vdash \mathtt{c}\theta\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\bar{\mathtt{y}}) : \mathtt{T}\theta}\ (\texttt{T-New}) \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{x}, \bar{\mathtt{y}} : \mathtt{X}, \bar{\mathtt{Y}}\\ \mathtt{m}(\bar{\mathtt{z}} : \bar{\mathtt{Z}}\&\mathtt{c}) : \mathtt{T} \in \mathtt{mType}(\mathtt{X})\\ \theta = [\mathtt{x}/\mathtt{this}, \bar{\mathtt{y}}/\bar{\mathtt{z}}]\\ \Gamma \vdash \bar{\mathtt{Y}} \leq \bar{\mathtt{Z}}\theta \quad \Gamma \vdash \mathtt{c}\theta\end{array}}{\Gamma \vdash \mathtt{x.m}(\bar{\mathtt{y}}) : \mathtt{T}\theta}\ (\texttt{T-Invoke})$$

$$\frac{\begin{array}{l}\Gamma[\bar{\mathtt{x}} : \bar{\mathtt{X}}] \vdash \bar{\mathtt{d}} : \bar{\mathtt{Y}}\\ \Gamma[\bar{\mathtt{x}} : \bar{\mathtt{X}}] \vdash \bar{\mathtt{Y}} \leq \bar{\mathtt{X}}\\ \Gamma[\bar{\mathtt{x}} : \bar{\mathtt{X}}] \vdash \mathtt{e} : \mathtt{Z}\end{array}}{\Gamma \vdash \{\mathtt{val}\ \bar{\mathtt{x}} : \bar{\mathtt{X}} = \bar{\mathtt{d}}; \mathtt{e}\} : \exists \bar{\mathtt{x}} : \bar{\mathtt{X}}.\mathtt{Z}}\ (\texttt{T-Local}) \qquad \frac{}{\Gamma \vdash (\mathtt{T})\mathtt{e} : \mathtt{T}}\ (\texttt{T-Cast})$$

$$\frac{\Gamma \vdash \mathtt{d} : \mathtt{S} \quad \Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{d}; \mathtt{e} : \mathtt{T}}\ (\texttt{T-Seq}) \qquad \frac{\begin{array}{c}\Gamma \vdash \mathtt{c} : \mathtt{boolean}\\ \Gamma, \mathtt{c} \vdash \mathtt{r} : \mathtt{T} \quad \Gamma, !\mathtt{c} \vdash \mathtt{s} : \mathtt{T}\end{array}}{\Gamma \vdash (\mathtt{c}?\mathtt{r} : \mathtt{s}) : \mathtt{T}}\ (\texttt{T-Cond})$$

$$\frac{\begin{array}{l}\mathtt{class}\ \mathtt{C}(\bar{\mathtt{f}} : \bar{\mathtt{X}}, \mathtt{var}\ \bar{\mathtt{g}} : \bar{\mathtt{Y}})\ \mathtt{extends}\ \mathtt{D}\{\mathtt{K}\ \bar{\mathtt{M}}\}\\ \mathtt{M}_{\mathtt{i}} = \mathtt{def}\ \mathtt{m}(\bar{\mathtt{z}} : \bar{\mathtt{Z}}\&\mathtt{d}) : \mathtt{T} = \mathtt{e}\\ \Gamma, \bar{\mathtt{z}} : \bar{\mathtt{Z}}, \mathtt{this} : \mathtt{C}, \mathtt{d} \vdash \mathtt{e} : \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{M}_{\mathtt{i}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}}\ (\texttt{T-Method})$$

$$\frac{\begin{array}{l}\mathtt{K} = \mathtt{def}\ \mathtt{C}(\overline{\mathtt{u}, \mathtt{f}, \mathtt{g}} : \overline{\mathtt{U}, \mathtt{X}, \mathtt{Y}}\&\mathtt{c}) : \mathtt{T} = \{\mathtt{super}(\bar{\mathtt{u}}); \mathtt{this}.\overline{\mathtt{f}, \mathtt{g}} = \overline{\mathtt{f}, \mathtt{g}}; \}\\ \mathtt{D}\ \mathtt{OK} \qquad \mathtt{K}_{\mathtt{d}} = \mathtt{ctor}(\mathtt{D}) = \mathtt{def}\ \mathtt{D}(\bar{\mathtt{u}} : \bar{\mathtt{U}}\&\mathtt{d}) : \mathtt{Z} = \ldots\\ \overline{\mathtt{u}, \mathtt{f}, \mathtt{g}} : \overline{\mathtt{U}, \mathtt{X}, \mathtt{Y}}, \mathtt{c} \vdash \mathtt{d}\\ \overline{\mathtt{u}, \mathtt{f}, \mathtt{g}} : \overline{\mathtt{U}, \mathtt{X}, \mathtt{Y}}, \mathtt{c}, \mathtt{this} : \mathtt{Z}, \mathtt{this}.\overline{\mathtt{f}, \mathtt{g}} = \overline{\mathtt{f}, \mathtt{g}} \vdash \mathtt{this} : \mathtt{T}\\ \bar{\mathtt{M}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\end{array}}{\mathtt{class}\ \mathtt{C}(\bar{\mathtt{f}} : \bar{\mathtt{X}},\ \mathtt{var}\,\bar{\mathtt{g}} : \bar{\mathtt{Y}})\ \mathtt{extends}\ \mathtt{D}\{\mathtt{K}\ \bar{\mathtt{M}}\}\ \mathtt{OK}}\ (\texttt{T-Class})$$

Note that (T-Cond) applies to conditionals c?r:s where the test is a constraint term. This means it can be built up using the primitive functions supplied with the constraint system, and must not reference any mutable variable.

2.3.3 Auxiliary definitions

The rules use some auxiliary definitions. For any type T, fields(T) is the set of typed fields for that type:

$$\mathtt{fields}(\mathtt{Object}) = \emptyset \quad \mathtt{fields}(\mathtt{C}(\&\mathtt{c})) = \mathtt{fields}(\mathtt{C})$$
$$\mathtt{fields}(\mathtt{C}) = \mathtt{fields}(\mathtt{D}), \bar{\mathtt{x}} : \bar{\mathtt{X}}, \bar{\mathtt{y}} : \bar{\mathtt{Y}} \ \ \textit{if}\ \mathtt{class}\ \ \mathtt{C}(\bar{\mathtt{x}} : \bar{\mathtt{X}}, \mathtt{var}\ \ \bar{\mathtt{y}} : \bar{\mathtt{Y}}\&\mathtt{c})\ \ \mathtt{extends}\ \ \mathtt{D}\ldots$$

For any type T, `varfields(T)` is the set of mutable fields for that type:

$$\texttt{varfields}(\texttt{Object}) = \emptyset \quad \texttt{varfields}(\texttt{C}(\&\texttt{c})) = \texttt{varfields}(\texttt{C})$$
$$\texttt{varfields}(\texttt{C}) = \texttt{varfields}(\texttt{D})[\bar{\texttt{y}} : \bar{\texttt{Y}}] \;\; \textit{if}\; \texttt{class C(}\bar{\texttt{x}} : \bar{\texttt{X}}, \texttt{var } \bar{\texttt{y}} : \bar{\texttt{Y}}\texttt{\&c) extends D...}$$

The set of constructors of a type `T=C(&c)` (`ctor(C)`) is the set of all elements $\texttt{C}(\bar{\texttt{x}} : \bar{\texttt{X}}\&\texttt{c}) : \texttt{T}$ such that the constructor $\texttt{def}\,\texttt{C}(\bar{\texttt{x}} : \bar{\texttt{X}}\&\texttt{c}) : \texttt{T}{=}\ldots$ occurs in the definition of the class C. We omit a formal definition. The set of methods of a type `T=C(&c)`, `mType(T)`, is the set of all elements $\texttt{m}(\bar{\texttt{z}} : \bar{\texttt{Z}}\&\texttt{c}) : \texttt{T}$ such that the method $\texttt{defm}(\bar{\texttt{z}} : \bar{\texttt{Z}}\&\texttt{c}) : \texttt{T} = \texttt{e}$ occurs in the definition of the class C or an ancestor class. We omit a formal definition.

2.4 Dynamic semantics

*Establish theorems: Subject reduction, well-typed program can do no wrong. Adapt from [**??X10-concur??**].*

2.5 Example revisited

Here we consider a concrete constraint system with the type `int` (with arithmetic operations).

Show the List example type-checks.

3 FX10 with places

- *Follow the outline of my recent set of slides.*
- *Every reference object now has a final* `place` *field,* `loc`*.*
- *The constant* `here` *evaluates to the current place.*
- `place` *is a value type, with a fixed but unknown set of operations, we can assume* `.next:place`*.*
- *The type* `T@!` *is represented by* `T(&loc=here)`*), and the type* `T@x` *by* `T(&loc=x.loc)`*.*
- *The rules for field read/write and method invocation are changed to require that the subject be of type* `_(&loc=here)`*.*
- *The new place-shifting control construct* `eval(p)e` *is introduced.*