Formalizing Dependent Types for X10

(Draft Version 0.09)
(Please do not cite)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 [6]), in the context of FX10, a Featherweight version of X10 [2]. 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 [7] 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

TO BE DONE: Add the ability to specify a constraint on super's fields when subclassing `class C(f:X, var g:Y) extends D(&c) {M}` *Here* `c` *is a constraint on* `f`,`g` *and the fields of* `D`. The abstract syntax for FX10 is specified in Table 1. We consider a parametric version of the language, with an underlying constraint system $\mathcal{C}$ [5] being used to specify dependent types.

We follow [3,1] in our treatment. Meta-variables `C`, `D`, `E` range over class names (including the "built-in" classes `int`, `boolean` and `Object`); `f` and `g` range over field names; `m` ranges over method names; `x`, `y`, `z` range over parameter and local constant names; other meta-variables are specified in Table 1.

We write $\bar{\mathtt{e}}$ as shorthand for $\mathtt{e_1}, \ldots, \mathtt{e_n}$ (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.

(Classes)	CL	::=	`class C(`$\bar{\mathtt{f}}:\bar{\mathtt{X}}$`, var `$\bar{\mathtt{g}}:\bar{\mathtt{Y}}$`)`	
			`extends D {`$\bar{\mathtt{M}}$`}`	
(Method)	M	::=	`def m(`$\bar{\mathtt{x}}:\bar{\mathtt{X}}$`&c):T = e`	
(Expression)	e,r,s	::=	`new C(`$\bar{\mathtt{x}}$`)`	(New object)
			| `{val x:X=e; e}`	(Local variable declaration)
			| `x.m(`$\bar{\mathtt{x}}$`)`	(Method Invocation)
			| `(T) e`	(Cast)
			| `x.f=e` | `e;e`	(Assignment, Sequencing)
			| `ce` | `ce => e : e`	
(Type)	T,U,...	::=	`C(&c)` | `nullable C(&c)`	
(Constraint)	c,d	::=	ce | $(\exists \mathtt{x}:\mathtt{T})\,\mathtt{c}$	
(Constraint Term)	ce	::=	`null` | `true` | `false` | `n`	
			| `x` | `x.f` | `ce op ce` | `op ce`	

Table 1: Abstract syntax for FX10.

We use the obvious abbreviation: $\bar{\mathtt{f}} : \bar{\mathtt{X}}$, for $\mathtt{f}_1 : \mathtt{X}_1, \ldots, \mathtt{f}_n : \mathtt{X}_n$ (n may be zero). var $\bar{\mathtt{g}} : \bar{\mathtt{Y}}$ abbreviates the sequence var $\mathtt{g}_1 : \mathtt{Y}_1, \ldots, \mathtt{g}_n : \mathtt{Y}_n$ if $\mathtt{n} > 1$ and the empty sequence otherwise. Empty parameter sequences may be omitted (like Scala, unlike Java).

Sequences of field declarations, parameter declarations, local variable declarations, are assumed to not contain any duplicates. Sequences of methods in a class must not contain two methods with the same sequence of parameters types. (FX10 permits *ad hoc* polymorphism.)

Class definition. A *class definition* specifies `val` (immutable) fields, `var` (mutable fields) and their types, its super-class, its list of methods, and an (implicit) constructor. Modulo minor syntactic changes (and dependent types), one may view the declaration

$$\mathtt{class\ C}(\bar{\mathtt{f}} : \bar{\mathtt{X}}, \mathtt{var}\ \bar{\mathtt{g}} : \bar{\mathtt{Y}})\ \mathtt{extends\ D}\ \{\bar{\mathtt{M}}\}$$

as shorthand for the Java class definition:

```
class C extends D {
  final X̄f̄;
  Ȳ ḡ;
  C(X̄ f̄,Ȳ ḡ,Z̄ h̄) {
    super(h̄);
    this.f̄=f̄;
    this.ḡ=ḡ;
  }
  M̄
}
```

where $\bar{\mathtt{h}} : \bar{\mathtt{Z}}$ is the list of argument types of the constructor for the super class D.

Methods. In a method definition, `c` is a constraint on the parameters $\bar{\mathtt{x}}$ and the final fields of the current object. This constraint must be true (statically) for the method to be invoked. One can think of `c` as allowing the programmer to specify conditions on the object which must be true for the method to be invoked. This flexibility is particularly valuable for nullary methods – for such methods the constraint cannot be folded into the type of arguments.

As usual for references to a field in the body of a class (not within a dependent type, see below), the occurrence of a field `f` in `c` stands for `this.f`.

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 constant names "`this`" and "`self`". That is, no program may define a local constant 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;` {`val y:Y=e1; x.m(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 abbreviated `e.m(e1)` form.

The two-armed conditional expression `ce => e : e` is 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`", "`int`" and "`boolean`". That is, the user may not define these classes. [1]

A *dependent type* is of the form `C(&c)` where `c` is a constraint. The phrase "`&c`" is called a *where clause*. We abbreviate `&true` to the empty string. Since empty parameter lists can be omitted, this means that `C(&true)` can be abbreviated to `C`.

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; this is checked statically.

`c` may contain references to parameters and local constants visible at the point of declaration of the type (including `this`), and the special constant `self`. `self` refers to the object whose type is being specified. A field `f` of `C` may occur unqualified in `c`, such a reference is supposed to be shorthand for `self.f`. References to the `f` field of the current object must be explicitly preceded by `this`.[2]

`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`, and in *subrange types*, e.g. `int(&self >= 0)`.

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, $\mathcal{C}$ [5]. *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.* $\mathcal{C}$ specifies a collection of value types (e.g. `int`, `bool`, constants of those types and operations on those types.

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

After Scala [4], 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.

The abstract syntax differs from the syntax in [6] in many notational respects. It differs substantively in that there is no distinction between parameters of classes and fields, and parameters of methods

[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.f` is different from `self.f`. 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)`.)

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 [3]. For now, the reader should understand generic syntax in the spirit of [3].

```
class List<X>( n:int(&self>=0),
               var node: nullable X,
                   rest: nullable List<X>(n-1)) extends Object {
  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 EmptyListInt:List<int(&self>=0)>(0)=new List<int(&self>=0)>(0,null,null)

  def append(arg:List<X>):List<X>(n+arg.n)  =
       (n == 0) => arg : new List<X>(node, rest.append(arg))
  def rplacd(arg:List<X>(n-1)):List<X>(n)  = {
       this.rest=arg;
       this
  }
  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))
  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) => EmptyListInt : EmptyListInt.makeList(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. Note that `rest` is forced to be `null` if `n=0`, since `List<X>(n-1)` will be empty in this case.

Three make methods are provided that call the implicit constructor with different arguments. While the first two return lists of constant size, the third takes returns a list of a size that depends on one of its arguments.

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[3] 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`. □

[3]These should really be `static`.

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.*
- *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 constraints. Constraints are taken from the underlying constraint system $\mathcal{C}$, and extended to include tokens of the form `var x.g:T` (read as: `x` *has a field* `g` *of type* `T`) and `x:T` (read as: `x` has type `T`).

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:

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

(asuming `n` is a field in `List`, and hence an abbreviation for `self.n`). We also remind the reader that for any constraint system $\Gamma \vdash c \;\&\; d$ iff $\Gamma \vdash c$ and $\Gamma \vdash d$.

Given a program (collection of classes and an expression), we shall also assume that the entailment relationship is closed under subtyping (see below):

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{C} \quad \mathtt{C} \leq \mathtt{D}}{\Gamma \vdash \mathtt{x} : \mathtt{D}}$$

and field selection. That is,

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{C} \quad \theta = [\mathtt{x}/\mathtt{this}] \quad \mathtt{class}\ \mathtt{C}(\bar{\mathtt{f}} : \bar{\mathtt{X}}, \mathtt{var}\ \bar{\mathtt{g}} : \bar{\mathtt{Y}}) \ldots}{\begin{array}{c}\Gamma \vdash \mathtt{val}\ \mathtt{x}.\bar{\mathtt{f}} \;\&\; \mathtt{x}.\bar{\mathtt{f}} : \bar{\mathtt{X}}\theta \\ \Gamma \vdash \mathtt{var}\ \mathtt{x}.\bar{\mathtt{g}} \;\&\; \mathtt{x}.\bar{\mathtt{g}} : \bar{\mathtt{Y}}\theta\end{array}}$$

(These assumptions enable us to state in a simple way that the constraints on fields of a class should be consistent. For, by requiring that `x:C` be consistent, we require that the conjunction of all the constraints on the fields of `x` is consistent, since a constraint is consistent iff its completion under entailment is consistent.)

Typing judgements for expressions are of the form $\Gamma \vdash \texttt{e} : \texttt{T}$, read as "Under the assumptions Γ, e has type T", and thought of as defining an inference relation that extends the underlying constraint system. We use sequence notation in the obvious way: $\Gamma \vdash \bar{\texttt{x}} : \bar{\texttt{X}}$ is shorthand for the collection of typing judgements $\Gamma \vdash \texttt{x}_1 : \texttt{X}_1 \ldots \Gamma \vdash \texttt{x}_n : \texttt{X}_n$.

2.3.1 Type well-formedness rules

We shall need the judgement $\Gamma \vdash \texttt{T}\star$, read as "Under the assumptions Γ, T is a valid type", and the judgement $\Gamma \vdash \texttt{val c}$, read as "Under the assumptions Γ, c is a constant term (not dependent on mutable fields)."

The rules for establishing val c are straightforward:

$$\frac{}{\begin{array}{l}\vdash \texttt{val null}\\ \vdash \texttt{val true}\\ \vdash \texttt{val false}\\ \vdash \texttt{val n}\\ \texttt{x} : \texttt{X} \vdash \texttt{val x}\end{array}} \quad \frac{\Gamma \vdash \texttt{X}\star \quad \Gamma \vdash \texttt{val c}}{\Gamma \vdash \texttt{val } (\exists \texttt{x} : \texttt{X})\texttt{c}} \quad \frac{\Gamma \vdash \texttt{val e}}{\Gamma \vdash \texttt{val op e}} \quad \frac{\Gamma \vdash \texttt{val e}_0 \,\&\, \texttt{val e}_1}{\Gamma \vdash \texttt{val e}_0 \texttt{ op e}_1}$$

The only inference rule for establishing Z⋆ is:

$$\frac{\Gamma, \texttt{self} : \texttt{C} \vdash \texttt{c} : \texttt{boolean} \,\&\, \texttt{val c}}{\Gamma \vdash \texttt{C}(\&\texttt{c}) \star}$$

2.3.2 Subtyping rules

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

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

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

2.3.3 Static semantics rules

First we cover the rules for literals:

$$\boxed{\frac{}{\begin{array}{l}\vdash \texttt{null} : \texttt{nullable T}\\ \vdash \texttt{n} : \texttt{int}(\texttt{n})\\ \vdash \texttt{true} : \texttt{boolean}(\texttt{true})\\ \vdash \texttt{false} : \texttt{boolan}(\texttt{false})\end{array}}\ (\texttt{T-Lit}) \quad \frac{\Gamma \vdash \texttt{e} : \texttt{T}}{\Gamma \vdash \texttt{e} : \texttt{nullable T}}\ (\texttt{T-Nullable})}$$

No rules are needed for typing field access or for establishing x:T; these are covered by the underlying constraint system.

$$\frac{\begin{array}{l}\mathtt{class}\ \mathtt{C}(\bar{\mathtt{f}}:\bar{\mathtt{X}},\ \mathtt{var}\ \bar{\mathtt{g}}:\bar{\mathtt{Y}})\ \mathtt{extends}\ \mathtt{D}\ldots\\ \bar{\mathtt{h}}:\bar{\mathtt{Z}}=\mathtt{fields}(\mathtt{D})\\ \Gamma\vdash\bar{\mathtt{x}},\bar{\mathtt{y}},\bar{\mathtt{z}}:\bar{\mathtt{X}}_1,\bar{\mathtt{Y}}_1,\bar{\mathtt{Z}}_1\\ \theta=[\bar{\mathtt{x}},\bar{\mathtt{y}},\bar{\mathtt{z}}/\bar{\mathtt{f}},\bar{\mathtt{g}},\bar{\mathtt{h}}]\\ \Gamma\vdash\bar{\mathtt{X}}_1,\bar{\mathtt{Y}}_1,\bar{\mathtt{Z}}_1\leq\bar{\mathtt{X}}\theta,\bar{\mathtt{Y}}\theta,\bar{\mathtt{Z}}\theta\end{array}}{\Gamma\vdash\mathtt{new}\ \mathtt{C}(\bar{\mathtt{x}},\bar{\mathtt{y}},\bar{\mathtt{z}}):\mathtt{C}(\&\bar{\mathtt{f}},\bar{\mathtt{g}},\bar{\mathtt{h}}==\bar{\mathtt{x}},\bar{\mathtt{y}},\bar{\mathtt{z}})}\ \texttt{(T-New)}\qquad\frac{\begin{array}{l}\Gamma\vdash\mathtt{x},\bar{\mathtt{y}}:\mathtt{X},\bar{\mathtt{Y}}\\ \mathtt{m}(\bar{\mathtt{z}}:\bar{\mathtt{Z}}\&\mathtt{c}):\mathtt{T}\in\mathtt{mType}(\mathtt{X})\\ \theta=[\mathtt{x}/\mathtt{this},\bar{\mathtt{y}}/\bar{\mathtt{z}}]\\ \Gamma\vdash\bar{\mathtt{Y}}\leq\bar{\mathtt{Z}}\theta\ \&\ \mathtt{c}\theta\end{array}}{\Gamma\vdash\mathtt{x.m}(\bar{\mathtt{y}}):\mathtt{T}\theta}\ \texttt{(T-Invoke)}$$

$$\frac{\begin{array}{l}\Gamma\vdash\mathtt{X}\star\\ \Gamma,\mathtt{x}:\mathtt{X}\ \textit{satisfiable}\\ \Gamma,\mathtt{x}:\mathtt{X}\vdash\mathtt{d}:\mathtt{Y}\ \&\ \mathtt{Y}\leq\mathtt{X}\ \&\ \mathtt{e}:\mathtt{Z}\end{array}}{\Gamma\vdash\{\mathtt{val}\ \mathtt{x}:\mathtt{X}=\mathtt{d};\mathtt{e}\}:\exists(\mathtt{x}:\mathtt{X})\,\mathtt{Z}}\ \texttt{(T-Local)}$$

$$\frac{\Gamma\vdash\mathtt{Y}\star\quad\Gamma\vdash\mathtt{e}:\mathtt{X}\ \&\ \mathtt{X}\leq\mathtt{Y}}{\Gamma\vdash(\mathtt{Y})\mathtt{e}:\mathtt{Y}}\ \texttt{(T-UCast)}\qquad\frac{\Gamma\vdash\mathtt{Y}\star\quad\Gamma\vdash\mathtt{e}:\mathtt{X}\quad\Gamma\not\vdash\mathtt{X}\leq\mathtt{Y}}{\Gamma\vdash(\mathtt{Y})\mathtt{e}:\mathtt{Y}}\ \texttt{(T-GCast)}$$

$$\frac{\Gamma\vdash\mathtt{ce}:\mathtt{boolean}\quad\Gamma,\mathtt{ce}\vdash\mathtt{r}:\mathtt{T}\quad\Gamma,!\mathtt{ce}\vdash\mathtt{s}:\mathtt{T}}{\Gamma\vdash(\mathtt{ce}\ \mathtt{=>}\ \mathtt{r}:\mathtt{s}):\mathtt{T}}\ \texttt{(T-Typecase)}$$

$$\frac{\begin{array}{l}\mathtt{M}\in\mathrm{methods}(\mathtt{C})\\ \mathtt{M}=\mathtt{def}\ \mathtt{m}(\bar{\mathtt{z}}:\bar{\mathtt{Z}}\&\mathtt{c}):\mathtt{T}=\mathtt{e}\\ \mathtt{this}:\mathtt{C}\vdash\bar{\mathtt{Z}}\star\\ \bar{\mathtt{z}}:\bar{\mathtt{Z}},\mathtt{this}:\mathtt{C}\vdash\mathtt{c}:\mathtt{boolean}\ \&\ \mathtt{val}\ \mathtt{c}\\ \bar{\mathtt{z}}:\bar{\mathtt{Z}},\mathtt{this}:\mathtt{C},\mathtt{c}\ \textit{satisfiable}\\ \bar{\mathtt{z}}:\bar{\mathtt{Z}},\mathtt{this}:\mathtt{C},\mathtt{c}\vdash\mathtt{e}:\mathtt{T}\end{array}}{\mathtt{M}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}}\ \texttt{(T-Method)}\qquad\frac{\begin{array}{l}\mathtt{this}:\mathtt{C}\vdash\bar{\mathtt{X}}\star\ \&\ \bar{\mathtt{Y}}\star\\ \bar{\mathtt{f}}:\bar{\mathtt{X}},\ \mathtt{var}\ \bar{\mathtt{g}}:\bar{\mathtt{Y}}\ \textit{satisfiable}\\ \mathtt{D}\ \mathtt{OK}\quad\bar{\mathtt{M}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\end{array}}{\mathtt{class}\ \mathtt{C}(\bar{\mathtt{f}}:\bar{\mathtt{X}},\ \mathtt{var}\,\bar{\mathtt{g}}:\bar{\mathtt{Y}})\ \mathtt{extends}\ \mathtt{D}\{\bar{\mathtt{M}}\}\ \mathtt{OK}}\ \texttt{(T-Class)}$$

Finally we cover the rules for field assignment and sequencing. These are standard.

$$\frac{\Gamma\vdash\mathtt{var}\ \mathtt{x.g}\ \&\ \mathtt{x.g}:\mathtt{U}\quad\Gamma\vdash\mathtt{e}:\mathtt{V}\quad\Gamma\vdash\mathtt{V}\leq\mathtt{U}}{\Gamma\vdash\mathtt{x.g}=\mathtt{e}:\mathtt{U}}\ \texttt{(T-Field-w)}\qquad\frac{\Gamma\vdash\mathtt{d}:\mathtt{S}\quad\Gamma\vdash\mathtt{e}:\mathtt{T}}{\Gamma\vdash\mathtt{d};\mathtt{e}:\mathtt{T}}\ \texttt{(T-Seq)}$$

2.4 Example revisited

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

Show the List example type-checks.

2.5 Dynamic semantics

We provide a dynamic semantics as a binary transition relation in the usual structural operational semantics style. We consider first the assignment-free sublanguage (the language without field-assignment, and sequencing). For such a language there is no reason to introduce heaps; the semantics can be specified by a transition relation on just expressions. The computation rules are not very different from [3]; in fact they are simpler because we permit only local constants as receivers of method calls and arguments of method calls and constructors.

First we define the notion of *canonical values*:

$$\text{(Canonical Value)}\quad \mathrm{v,w}\quad ::=\quad \mathtt{null}\ |\ \mathtt{true}\ |\ \mathtt{false}\ |\ \mathtt{n}\ |\ \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})$$

We shall see that values are *terminal* in the dynamic semantics (they cannot evolve).

$$\frac{\mathtt{fields(C)} = \bar{\mathtt{f}} : \bar{\mathtt{X}}}{(\mathtt{new\ C}(\bar{\mathtt{v}})).\mathtt{f_i} \longrightarrow \mathtt{v_i}} \text{ (R-FIELD)} \qquad \frac{\mathtt{mbody(m, C)} = (\bar{\mathtt{x}}, \mathtt{e})}{(\mathtt{new\ C}(\bar{\mathtt{v}})).\mathtt{m}(\bar{\mathtt{w}}) \longrightarrow \mathtt{e}[\bar{\mathtt{w}}/\bar{\mathtt{x}}, \mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}]} \text{ (R-INVOKE)}$$

$$\frac{}{(\mathtt{nullable\ T})\ \mathtt{null} \longrightarrow \mathtt{null}} \text{ (R-CAST-NULL)} \qquad \frac{\mathtt{C} \leq \mathtt{D} \quad \mathtt{c}[\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{self}] \longrightarrow^* \mathtt{true}}{\begin{array}{l}(\mathtt{D}(\&\mathtt{c}))(\mathtt{new\ C}(\bar{\mathtt{v}})) \longrightarrow \mathtt{new\ C}(\bar{\mathtt{v}}) \\ (\mathtt{nullable\ D}(\&\mathtt{c}))(\mathtt{new\ C}(\bar{\mathtt{v}})) \longrightarrow \mathtt{new\ C}(\bar{\mathtt{v}})\end{array}} \text{ (R-CAST)}$$

$$\frac{}{\{\mathtt{val\ x : T = v;\ e}\} \longrightarrow \mathtt{e}[\mathtt{v}/\mathtt{x}]} \text{ (R-LOCAL)}$$

$$\frac{}{\begin{array}{l}\mathtt{true} \Rightarrow \mathtt{e_0 : e_1} \longrightarrow \mathtt{e_0} \\ \mathtt{false} \Rightarrow \mathtt{e_0 : e_1} \longrightarrow \mathtt{e_1}\end{array}} \text{ (R-TYPECASE)} \qquad \frac{}{\begin{array}{l}\mathtt{v\ op\ w} \longrightarrow \mathtt{z} \quad (\mathtt{z} \text{ is result of } \mathtt{v\ op\ w}) \\ \mathtt{op\ v} \longrightarrow \mathtt{z} \quad (\mathtt{z} \text{ is result of } \mathtt{op\ v})\end{array}} \text{ (R-PRIMITIVE)}$$

Finally, we have the context rules:

$$\frac{\mathtt{r} \longrightarrow \mathtt{r}'}{\begin{array}{l}\{\mathtt{val\ x : T = r;\ e}\} \longrightarrow \{\mathtt{val\ x : T = r';\ e}\} \\ \mathtt{r} \Rightarrow \mathtt{e_0 : e_1} \longrightarrow \mathtt{r}' \Rightarrow \mathtt{e_0 : e_1} \\ (\mathtt{T})\ \mathtt{r} \longrightarrow (\mathtt{T})\ \mathtt{r}'\end{array}} \text{ (R-CONTEXT)}$$

2.6 Properties of the typing system

Below, we say that `e` is *stuck* if there is no `r` such that $e \longrightarrow r$.

Proofs of these lemmas and theorems are being worked on.

Lemma 2.1 (Well-formedness) *If* $\Gamma \vdash e : T$ *then* Γ *is satisfiable, and* $\Gamma \vdash T\star$.

Lemma 2.2 (Term substitution preserves types) *If* $\Gamma, \mathtt{x} : \mathtt{X} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma \vdash \mathtt{r} : \mathtt{Y}$ *such that* $\Gamma \vdash \mathtt{Y} \leq \mathtt{X}$ *then* $\Gamma \vdash \mathtt{e}[\mathtt{r}/\mathtt{x}] : \mathtt{T}'$ *for some* $\mathtt{T}' \leq \mathtt{T}$.

Theorem 2.3 (Type soundness) $\Gamma \vdash e' : T'$ *whenever for some* T *s.t.* $\Gamma \vdash T' \leq T$, $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$.

Say that $\Gamma \vdash_s e$ (read: "`e`" is safely well-typed in Γ) if for some T, $\Gamma \vdash_e e : T$ and the derivation does not use rule (T-GCAST). (Invocations of the cast operation "checked" by such a rule can fail dynamically.)

Theorem 2.4 (Safety) *If* $\Gamma \vdash_s e$ *and* $e \vdash e'$ *then* $\Gamma \vdash_s e'$. *If* $\Gamma \vdash_{\mathtt{s}} \mathtt{e}$ *and* `e` *is stuck then* `e` *is a value.*

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.*

References

[1] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[2] V. Saraswat et al. Report on the programming language X10. Technical report, IBM TJ Watson Research Center, 2006.

[3] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1999.

[4] M. Odersky. Report on the programming language Scala. Technical report, EPFL, 2006.

[5] V. Saraswat. The Category of Constraint Systems is Cartesian Closed. In *LICS '92*, pages 341–345, 1992.

[6] V. Saraswat. Adding Dependent Types to X10. Technical report, IBM TJ Watson Research Center, 2004.

[7] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR*, 2005.

A An extended example

```
/**
   A distributed binary tree.
   @author Satish Chandra 4/6/2006
   @author vj
 */

//                             ____P0
//                           |      |
//                           |      |
//                          _P2  __P0
//                         |  | |   |
//                         |  | |   |
//                        P3 P2 P1 P0
//                        *  *  *  *
// Right child is always on the same place as its parent;
// left child is at a different place at the top few levels of the tree,
// but at the same place as its parent at the lower levels

class Tree(localLeft: boolean,
           left: nullable Tree(& localLeft => loc=here),
           right: nullable Tree(& loc=here),
           next: nullable Tree) extends Object {
    /**
       Thread all the nodes together in a post-ordered list,
       returning the first node in the list.
    */
    def postOrder:Tree = {
        var result:Tree = this;
        if (right != null) {
            val result:Tree = right.postOrder();
            right.next = this;
            if (left != null) return left.postOrder(tt);
        } else if (left != null) return left.postOrder(tt);
        this
    }
    def postOrder(rest: Tree):Tree = {
        this.next = rest;
        postOrder()
    }

    def sum:int = size + (right==null => 0 : right.sum()) + (left==null => 0 : left.sum)
}
class TreeMaker {
    // Create a binary tree on span places.
    def build(count:int, span:int): nullable Tree(& localLeft==span/2==0) = {
        if (count == 0) return null;
        {val ll:boolean = (span/2==0);
         new Tree(ll,  eval(ll => here : place.places(here.id+span/2)){build(count/2, span/2)},
           build(count/2, span/2),count)}
    }
}
```