Initialization in X10 - Technical Report

Yoav Zibin yoav.zibin@gmail.com

1 Introduction

This technical report formalizes the hardhat initialization rules in X10 using *Featherweight X10* (FX10). Read first the paper "Object Initialization in X10" to understand the motivation behind the hardhat rules.

FX10 is similar to Featherweight Java (FJ), however it is both imperative (using a heap) and it models X10 specific constructs such as `finish` and `async`. FX10 also models field initialization including the fact that one can read a field only after it was definitely assigned, and that final (`val`) fields can be assigned exactly once (in Sec. 5).

FX10 does *not* model other aspects of X10 such as:

Places X10 can run code in multiple places, and the `at` keyword is used to execute code in a different place. Because the only raw object is `this`, and a raw `this` cannot be captured by an `at`, then only cooked objects can cross places. Therefore, there is no initialization issues with `at` and FX10 does not model it.

Inference X10 uses inference in various places: (i) it infers the type of a final field with an initializer, (ii) it infers method return types, (iii) using an inter-procedural dataflow analysis it infers 3 sets for each non-escaping method: `Read`, `SyncWrite`, `AsyncWrite`. `Read` is the set of fields that can be read by the method, `SyncWrite` are the fields that must be definitely-assigned by the method, and `AsyncWrite` are the fields that must be definitely-asynchronously-assigned by the method.

FX10 does not model inference, and instead this information is explicitly presented.

null FX10 guarantees that all fields are assigned when the object becomes cooked, and that fields are read only after written to. Therefore, `null` is no longer needed. In the formalism, an object is represented as a mapping from initialized fields to their values (so initially the mapping is empty because no field is initialized).

Miscellaneous Generics, constraints, casting, inner classes, overloading, co-variant return type, private/final, locals, field initializers, etc.

Overview Sec. 2 presents the syntax of FX10. Sec. 3 shows the typing rules (e.g., $\Gamma \vdash \mathtt{e} : \mathtt{C}$ denotes that expression `e` has type `C` in environment Γ), and defines various helper functions. Sec. 4 gives the reduction rules ($\mathtt{e}, H \rightsquigarrow \mathtt{e}', H'$) and our soundness proof. Finally, Sec. 5 extends the formalism with `val` and `var` fields.

2 Syntax

$\mathtt{P} ::= \overline{\mathtt{L}}$	Program.
$\mathtt{L} ::= \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{F}};\ \mathtt{ctor}(\overline{\mathtt{x}} : \overline{\mathtt{C}})\{\ \mathtt{super}(\overline{\mathtt{e}}); \mathtt{e};\ \}\ \overline{\mathtt{M}}\ \}$	cLass declaration.
$\mathtt{F} ::= \mathtt{f} : \mathtt{C}$	Field declaration.
$\mathtt{M} ::= \mathtt{MM\ m}(\overline{\mathtt{x}} : \overline{\mathtt{C}}) : \mathtt{C} = \mathtt{e};$	Method declaration.
$\mathtt{MM} ::= \mathtt{escaping} \mid \mathtt{Read}(\overline{\mathtt{f}})\ \mathtt{SyncWrite}(\overline{\mathtt{f}})\ \mathtt{AsyncWrite}(\overline{\mathtt{f}})$	Method Modifier.
$\mathtt{e} ::= \mathtt{l} \mid \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}}) \mid \mathtt{finish\ e} \mid \mathtt{async\ e}\ ; \mathtt{e}$	Expressions.

Figure 1: FX10 Syntax. The terminals are locations (`l`), parameters and `this` (`x`), field name (`f`), method name (`m`), class name (`B`,`C`,`D`,`Object`), and keywords (`super`, `escaping`, `Read`, `SyncWrite`, `AsyncWrite`, `new`, `finish`, `async`, `ctor`). The program source code cannot contain locations (`l`), because locations are only created during execution/reduction in R-NEW of Fig. 3.

Fig. 1 shows the syntax of FX10. (Sec. 5 will later add the `val` and `var` field modifiers.)

The syntax is similar to X10 real syntax with the following difference: instead of doing inference, we explicitly write the fields of `this` that are initialized and read in every initializing method (using 3 sets: $\texttt{Read}(\overline{\texttt{f}})$, $\texttt{SyncWrite}(\overline{\texttt{f}})$, and $\texttt{AsyncWrite}(\overline{\texttt{f}})$). Initializing methods (also called non-escaping methods) cannot leak `this` and they can only read fields of `this` that are in the `Read` set. Non-initializing methods (whose receiver is always a cooked object) are marked with `escaping` because they can escape `this` (e.g., pass `this` as an argument to another method).

Every class has a single constructor that must initialize all the fields of `this`, and it can only call non-escaping methods. The constructor first calls `super` to initialize the superclass's fields, and only then initialize the other fields.

The following is an example program in this syntax:

```
class E extends Object {
  ctor() =
    super();
    new Object();
}
class Seq extends Object {
  ctor(a1:Object, a2:Object) =
    super();
    a2;
}
class C2 extends Object {
  fVal:E;
  fVar:E;
  ctor(a1:E, a2:E) =
    super();
    finish
      async  this.fVal = a1;
      m(a2);
  Read() SyncWrite() AsyncWrite(fVar) m(a:E):E =
    async  this.fVar = a;
    new E();
  escaping m2():Seq =   new Seq(this,this.fVal);
}
class C3 extends Object {
  fVar:E;
  ctor(a:E) =
    super();
    new Seq(fVar=a,fVar);   // ok
}
class C4 extends Object {
  fVar:E;
  ctor(a:E) =
    super();
    new Seq(fVar,fVar=a);   // ERR (read before write)
}
```

3 Typing

Subtyping is exactly as in FJ: the transitive closure of the `extends` relation. That is, $\texttt{C} \leq \texttt{D}$ iff $\texttt{C} = \texttt{D}$ or `C` transitively extends `D`.

Similarly, we define: (i) $\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}}$ returns all fields of `C` (both those declared by `C` and recursively inherited from its superclass), and $\mathit{ftype}(\texttt{f}_i,\texttt{C}) = \texttt{D}_i$. (ii) $\mathit{mtype}(\texttt{m},\texttt{C}) = \overline{\texttt{B}} \mapsto \texttt{D}$ returns the type of method `m` in class `C`. (iii) $\mathit{mbody}(\texttt{m},\texttt{C}) = \overline{\texttt{x}}.\texttt{e}$ returns the method body of `m` in class `C`. Because methods have a modifier (`MM`) in FX10, we also define: (iv) $\mathit{mmodifier}(\texttt{m},\texttt{C}) = \texttt{MM}$ returns the method modifier of `m` in class `C`.

Finally, because FX10 have constructors, we define: (vi) $\mathit{ctortype}(\texttt{C}) = \overline{\texttt{B}} \mapsto \texttt{C}$ returns the type of the constructor in `C`, (vii) $\mathit{ctorbody}(\texttt{C}) = \overline{\texttt{x}}.\texttt{super}\ (\overline{\texttt{e}});\texttt{e}$; returns the body of the constructor. (Note that constructors do not have a modifier, because they implicitly must write to all fields and cannot read any field.)

Function $\mathtt{R}(\mathtt{e},F,\mathtt{C})$ returns whether `this` does not escape from `e` (i.e., `this` can be used only as a field or method receiver), and whether `e` only reads fields of `this` that are in F or that have been previously written in e. For example,

$$\begin{aligned} \mathtt{R}(\mathtt{this.f},\{\mathtt{f}\},\mathtt{C}) &= \mathtt{true} \\ \mathtt{R}(\mathtt{new\ Seq(this.f{=}new\ Object(),\ this.f)}\quad,\emptyset,\mathtt{C}) &= \mathtt{true} \end{aligned} \tag{1}$$

$$\mathtt{R}([\mathtt{e}_0,\ldots,\mathtt{e}_n],F,\mathtt{C}) = \mathtt{R}(\mathtt{e}_0,F,\mathtt{C}) \text{ and } \mathtt{R}(\mathtt{e}_1,\mathtt{SW}([\mathtt{e}_0],\mathtt{C})\cup F,\mathtt{C}) \text{ and } \ldots \text{ and } \mathtt{R}(\mathtt{e}_n,\mathtt{SW}([\mathtt{e}_0,\ldots,\mathtt{e}_{n-1}],\mathtt{C})\cup F,\mathtt{C})$$

$$\mathtt{R}(\mathtt{e}'',F,\mathtt{C}) = \begin{cases} \mathtt{true} & \mathtt{e}'' \equiv \mathtt{l} \\ \mathtt{false} & \mathtt{e}'' \equiv \mathtt{this} \\ \mathtt{true} & \mathtt{e}'' \equiv \mathtt{x} \\ \mathtt{f} \in F & \mathtt{e}'' \equiv \mathtt{this.f} \\ \mathtt{R}(\mathtt{e},F,\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e.f} \\ \mathtt{R}(\mathtt{e}',F,\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{this.f} = \mathtt{e}' \\ \mathtt{R}([\mathtt{e},\mathtt{e}'],F,\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e.f} = \mathtt{e}' \\ \mathtt{false} & \mathtt{e}'' \equiv \mathtt{this.m}(\overline{\mathtt{e}}) \quad \textit{mmodifier}(\mathtt{m},\mathtt{C}) = \mathrm{escaping} \\ R \subseteq F \text{ and } \mathtt{R}([\overline{\mathtt{e}}],F,\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{this.m}(\overline{\mathtt{e}}) \quad \textit{mmodifier}(\mathtt{m},\mathtt{C}) = \mathrm{Read}\ (R)\ldots \\ \mathtt{R}([\mathtt{e}',\overline{\mathtt{e}}],F,\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}}) \\ \mathtt{R}([\overline{\mathtt{e}}],F,\mathtt{C}) & \mathtt{e}'' \equiv \mathrm{new}\ \mathtt{C}\ (\overline{\mathtt{e}}) \\ \mathtt{R}(\mathtt{e},F,\mathtt{C}) & \mathtt{e}'' \equiv \mathrm{finish}\ \mathtt{e} \\ \mathtt{R}(\mathtt{e},F,\mathtt{C}) \text{ and } \mathtt{R}(\mathtt{e}',F,\mathtt{C}) & \mathtt{e}'' \equiv \mathrm{async}\ \mathtt{e}\ ;\mathtt{e}' \end{cases} \tag{2}$$

$\mathtt{SW}(\mathtt{e},\mathtt{C})$ is the set of fields of `this` that are definitely-(synchronously)-written in `e`. (The function is undefined if there is a call to an `escaping` method.)

$$\mathtt{SW}([\mathtt{e}_0,\ldots,\mathtt{e}_n],\mathtt{C}) = \mathtt{SW}(\mathtt{e}_0,\mathtt{C}) \cup \ldots \cup \mathtt{SW}(\mathtt{e}_n,\mathtt{C})$$

$$\mathtt{SW}(\mathtt{e}'',\mathtt{C}) = \begin{cases} \emptyset & \mathtt{e}'' \equiv \mathtt{l} \\ \emptyset & \mathtt{e}'' \equiv \mathtt{x} \\ \mathtt{SW}(\mathtt{e},\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e.f} \\ \{\mathtt{f}\} \cup \mathtt{SW}(\mathtt{e}',\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{this.f} = \mathtt{e}' \\ \mathtt{SW}([\mathtt{e},\mathtt{e}'],\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e.f} = \mathtt{e}' \\ S \cup \mathtt{SW}([\overline{\mathtt{e}}],\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{this.m}(\overline{\mathtt{e}}) \quad \textit{mmodifier}(\mathtt{m},\mathtt{C}) = \ldots \mathtt{SyncWrite}\ \ (S)\ldots \\ \mathtt{SW}([\mathtt{e}',\overline{\mathtt{e}}],\mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}}) \\ \mathtt{SW}([\overline{\mathtt{e}}],\mathtt{C}) & \mathtt{e}'' \equiv \mathrm{new}\ \mathtt{C}\ (\overline{\mathtt{e}}) \\ \mathtt{AW}(\mathtt{e},\mathtt{C}) & \mathtt{e}'' \equiv \mathrm{finish}\ \mathtt{e} \\ \mathtt{SW}(\mathtt{e}',\mathtt{C}) & \mathtt{e}'' \equiv \mathrm{async}\ \mathtt{e}\ ;\mathtt{e}' \end{cases} \tag{3}$$

$\mathtt{AW}(\mathtt{e},\mathtt{C})$ is the set of fields of `this` that are asynchronously written in `e`. Note that in $\mathtt{AW}(\mathrm{async}\ \mathtt{e}\ ;\mathtt{e}',\mathtt{C})$ we collect

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C}}{\Gamma \vdash \mathsf{finish}\ \mathtt{e}\ : \mathtt{C}}\ \text{(T-FINISH)} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \quad \Gamma \vdash \mathtt{e}' : \mathtt{C}'}{\Gamma \vdash \mathsf{async}\ \mathtt{e}\ ;\ \mathtt{e}'\ :\ \mathtt{C}'}\ \text{(T-ASYNC)}$$

$$\frac{}{\Gamma \vdash \mathtt{l} : \Gamma(\mathtt{l})}\ \text{(T-LOCATION)} \qquad \frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \text{(T-PARAMETER)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{f}} : \overline{\mathtt{C}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{C}_i}\ \text{(T-FIELD-ACCESS)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{f}} : \overline{\mathtt{C}} \quad \Gamma \vdash \mathtt{e}' : \mathtt{C}' \quad \mathtt{C}' \le \mathtt{C}_i}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i = \mathtt{e}' : \mathtt{C}'}\ \text{(T-FIELD-ASSIGN)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \mapsto \mathtt{C} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \overline{\mathtt{C}} \le \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}}\ \text{(T-INVOKE)}$$

$$\frac{\mathit{ctortype}(\mathtt{C}) = \overline{\mathtt{D}} \mapsto \mathtt{C} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \overline{\mathtt{C}} \le \overline{\mathtt{D}}}{\Gamma \vdash \mathsf{new}\ \mathtt{C}\ (\overline{\mathtt{e}}) : \mathtt{C}}\ \text{(T-NEW)}$$

Figure 2: FX10 Expression Typing Rules ($\Gamma \vdash \mathtt{e} : \mathtt{C}$). Rules (T-FINISH) and (T-ASYNC) handle the new constructs in FX10, while the other rules are identical to those in FJ.

writes to fields in both $\mathtt{e}$ and $\mathtt{e}'$, whereas in $\mathtt{SW}(\mathsf{async}\ \mathtt{e}\ ; \mathtt{e}', \mathtt{C})$ we only consider writes in $\mathtt{e}'$.

$$\mathtt{AW}([\mathtt{e}_0, \ldots, \mathtt{e}_n], \mathtt{C}) = \mathtt{AW}(\mathtt{e}_0, \mathtt{C}) \cup \ldots \cup \mathtt{AW}(\mathtt{e}_n, \mathtt{C})$$

$$\mathtt{AW}(\mathtt{e}'', \mathtt{C}) = \begin{cases} \emptyset & \mathtt{e}'' \equiv \mathtt{l} \\ \emptyset & \mathtt{e}'' \equiv \mathtt{x} \\ \mathtt{AW}(\mathtt{e}, \mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e.f} \\ \{\mathtt{f}\} \cup \mathtt{AW}((,\mathtt{C})\mathtt{e}') & \mathtt{e}'' \equiv \mathtt{this\ .f} = \mathtt{e}' \\ \mathtt{AW}([\mathtt{e}, \mathtt{e}'], \mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e.f} = \mathtt{e}' \\ A \cup \mathtt{AW}([\overline{\mathtt{e}}], \mathtt{C}) & \mathtt{e}'' \equiv \mathtt{this\ .m}(\overline{\mathtt{e}}) \quad \mathit{mmodifier}(\mathtt{m}, \mathtt{C}) = \ldots \mathtt{AsyncWrite}\ \ (A) \ldots \\ \mathtt{AW}([\mathtt{e}', \overline{\mathtt{e}}], \mathtt{C}) & \mathtt{e}'' \equiv \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}}) \\ \mathtt{AW}([\overline{\mathtt{e}}], \mathtt{C}) & \mathtt{e}'' \equiv \mathsf{new}\ \mathtt{C}\ (\overline{\mathtt{e}}) \\ \mathtt{AW}(\mathtt{e}, \mathtt{C}) & \mathtt{e}'' \equiv \mathsf{finish}\ \mathtt{e} \\ \mathtt{AW}(\mathtt{e}, \mathtt{C}) \cup \mathtt{AW}(\mathtt{e}', \mathtt{C}) & \mathtt{e}'' \equiv \mathsf{async}\ \mathtt{e}\ ; \mathtt{e}' \end{cases} \tag{4}$$

Like in FJ, we check that method declarations are ok by ensuring that the type of the method body is a subtype of the return type, and that an overriding method has the same signature. Unlike FJ, we also check that the method modifier (whether it is `escaping` or if it has the 3 sets specified).

$$\frac{\begin{array}{c} \Gamma = \{\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this}\ : \mathtt{C}\} \\ \Gamma \vdash \mathtt{e} : \mathtt{D}' \quad \mathtt{D}' \le \mathtt{D} \\ \mathtt{if\ MM}\ = \mathtt{Read}\ (R)\ \mathtt{SyncWrite}\ \ (W_s)\ \mathtt{AsyncWrite}\ \ (W_a) \\ \mathtt{then\ R}\ (\mathtt{e}, R, \mathtt{C})\ \ \mathtt{and}\ \ W_s \subseteq W_a\ \ \mathtt{and}\ \ \mathtt{SW}\ (\mathtt{e}, \mathtt{C}) \supseteq W_s\ \ \mathtt{and}\ \ \mathtt{AW}\ (\mathtt{e}, \mathtt{C}) \supseteq W_a \end{array}}{\mathtt{MM\,m}\,(\overline{\mathtt{x}} : \overline{\mathtt{D}}) : \mathtt{D} = \mathtt{e}\ \ \mathtt{OK\ IN\ C}}\ \text{(T-METHOD)}$$

$$\dfrac{\begin{array}{c}\texttt{class C extends C}'\ \{\ \ldots\ \} \\ \texttt{if}\ \ \mathit{mtype}(\texttt{m},\texttt{C}') = \overline{\texttt{D}'} \mapsto \texttt{D}' \\ \texttt{then}\ \ \overline{\texttt{D}'} \equiv \overline{\texttt{D}}\ \ \texttt{and}\ \ \texttt{D}' \equiv \texttt{D} \\ \texttt{if}\ \ \mathit{mmodifier}(\texttt{m},\texttt{C}') = \texttt{MM}' \\ \texttt{then MM} = \texttt{escaping} \Rightarrow \texttt{MM}' = \texttt{escaping and} \\ \texttt{MM} = \texttt{Read}\ (R)\ \texttt{SyncWrite}\ (W_s)\ \texttt{AsyncWrite}\ (W_a) \Rightarrow \\ (\texttt{MM}' = \texttt{escaping or} \\ (\texttt{MM}' = \texttt{Read}\ (R')\ \texttt{SyncWrite}\ (W_s')\ \texttt{AsyncWrite}\ (W_a')\ \texttt{and}\ (R \subseteq R'\ \texttt{and}\ W_s \supseteq W_s'\ \texttt{and}\ W_a \supseteq W_a'))\end{array}}{\texttt{MM m}\,(\overline{\texttt{x}} : \overline{\texttt{D}}) : \texttt{D} = \texttt{e}\ \ \texttt{Override-OK}\ \ \texttt{IN C}} \quad \text{(T-METHOD-OVERRIDE)}$$

Similarly, we need to check that constructor declarations are ok by checking the `super` call, that all fields are definitely-assigned and non are read before written. Obviously, the constructor of `Object` has no parameters.

$$\dfrac{\begin{array}{c}\Gamma = \{\overline{\texttt{x}} : \overline{\texttt{D}}, \texttt{this} : \texttt{C}\} \\ \texttt{class C extends C}'\ \{\ \ldots\ \} \\ F_s = \mathit{fields}(\texttt{C}') \quad F_a = \mathit{fields}(\texttt{C}) \quad F_d = F_a \setminus F_s \\ \Gamma \vdash \texttt{e}' : \texttt{B}' \\ \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{B}} \quad \mathit{ctortype}(\texttt{C}') = \overline{\texttt{B''}} \mapsto \texttt{C}' \quad \overline{\texttt{B}} \leq \overline{\texttt{B''}} \\ \texttt{R}([\overline{\texttt{e}}], \emptyset, \texttt{C})\ \texttt{and R}\ (\texttt{e}', F_s, \texttt{C})\ \texttt{and SW}\ (\texttt{e}', \texttt{C}) \supseteq F_d\end{array}}{\texttt{ctor}\ (\overline{\texttt{x}} : \overline{\texttt{D}})\{\ \texttt{super}\ (\overline{\texttt{e}}); \texttt{e}';\ \}\ \texttt{OK IN C}} \quad \text{(T-CTOR)}$$

As in FJ, a class is ok if its constructor (T-CTOR) and all its methods (T-METHOD) are ok.

4 Reduction

An object $\texttt{o} = \texttt{C}(\overline{\texttt{f}} \mapsto \overline{\texttt{l}'})$ is an instance of some class `C` where fields $\overline{\texttt{f}}$ has been initialized to locations $\overline{\texttt{l}'}$. A heap H is a mapping from locations `l` to objects `o`. A heap-typing Γ_H maps locations to their type, i.e., if $H[\texttt{l}] = \texttt{C}(\ldots)$ then $\Gamma_H[\texttt{l}] = \texttt{C}$. A heap H is *well-typed* iff each field location is a subtype (using Γ_H) of the declared field type, i.e., for every location in the heap $\texttt{l} \in \mathrm{dom}(H)$, where $H[\texttt{l}] = \texttt{C}(\overline{\texttt{f}} \mapsto \overline{\texttt{l}'})$ and for every field $\texttt{f}_i$, we have that $\Gamma_H(\texttt{l}_i') \leq \mathit{ftype}(\texttt{f}_i, \texttt{C})$.

An expression `e` is called *closed* if it does not contain any free variables (i.e., it does not contain method parameters `x` nor `this`).

Consider a program `P` and a closed expression `e`. If a program is well-typed, then the expression can always be reduced to a location (therefore, a field is always read after written to).

Theorem 4.1. ***(Progress and Preservation)*** *For every closed expression* $\texttt{e} \neq \texttt{l}$*, and a well-typed heap* H*, if* $\Gamma_H \vdash \texttt{e} : \texttt{C}$*, then there exists* $H', \texttt{e}', \texttt{C}'$ *such that (i)* $H, \texttt{e} \rightsquigarrow H', \texttt{e}'$*, (ii)* $\Gamma_{H'} \vdash \texttt{e}' : \texttt{C}'$*, and* $\texttt{C}' \leq \texttt{C}$*, (iii)* H' *is well-typed, (iv)* $\texttt{e}'$ *is closed.*

Proof. … □

5 Final fields

`val` and `var` fields.

5.1 Syntax

5.2 Typing

Fields also have a modifier (`FM`) so we define: $\mathit{fmodifier}(\texttt{f}, \texttt{C}) = \texttt{FM}$ returns the field modifier of `f` in class `C` (either `val` or `var`).

A method cannot assign to any `val` fields (no matter what the target is):

$$\mathit{methodVal}_\Gamma([\mathtt{e}_0,\ldots,\mathtt{e}_n]) = \mathit{methodVal}_\Gamma(\mathtt{e}_0) \ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathit{methodVal}_\Gamma(\mathtt{e}_n)$$

$$\mathit{methodVal}_\Gamma(\mathtt{e''}) = \begin{cases} \mathtt{true} & \mathtt{e''} \equiv \mathtt{l} \\ \mathtt{true} & \mathtt{e''} \equiv \mathtt{x} \\ \mathit{methodVal}_\Gamma(\mathtt{e}) & \mathtt{e''} \equiv \mathtt{e.f} \\ \mathit{methodVal}_\Gamma([\mathtt{e},\mathtt{e}']) \ \mathtt{and}\ \mathtt{FM} = \mathtt{var} & \mathtt{e''} \equiv \mathtt{e.f} = \mathtt{e}' \quad \Gamma \vdash \mathtt{e} : \mathtt{C} \quad \mathit{fmodifier}(\mathtt{f},\mathtt{C}) = \mathtt{FM} \\ \mathit{methodVal}_\Gamma([\mathtt{e}',\overline{\mathtt{e}}]) & \mathtt{e''} \equiv \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}}) \\ \mathit{methodVal}_\Gamma([\overline{\mathtt{e}}]) & \mathtt{e''} \equiv \mathtt{new\ C}\ (\overline{\mathtt{e}}) \\ \mathit{methodVal}_\Gamma(\mathtt{e}) & \mathtt{e''} \equiv \mathtt{finish\ e} \\ \mathit{methodVal}_\Gamma([\mathtt{e},\mathtt{e}']) & \mathtt{e''} \equiv \mathtt{async\ e}\ ;\mathtt{e}' \end{cases} \tag{5}$$

A constructor can assign to a `val` field at most once and only if the target is `this` :

$$\mathit{ctorVal}_\Gamma([\mathtt{e}_0,\ldots,\mathtt{e}_n],F) = \mathit{ctorVal}_\Gamma(\mathtt{e}_0,F)\ \mathtt{and}\ \mathit{ctorVal}_\Gamma(\mathtt{e}_1,\mathtt{AW}([\mathtt{e}_0],\mathtt{C}) \cup F)\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathit{ctorVal}_\Gamma(\mathtt{e}_n,\mathtt{AW}([\mathtt{e}_0,\ldots,\mathtt{e}_{n-1}],\mathtt{C}) \cup F)$$

$$\mathit{ctorVal}_\Gamma(\mathtt{e''},F) = \begin{cases} \mathtt{true} & \mathtt{e''} \equiv \mathtt{l} \\ \mathtt{true} & \mathtt{e''} \equiv \mathtt{x} \\ \mathit{ctorVal}_\Gamma(\mathtt{e},F) & \mathtt{e''} \equiv \mathtt{e.f} \\ \mathit{ctorVal}_\Gamma([\mathtt{e},\mathtt{e}'],F \cup \{\mathtt{f}\})\ \mathtt{and} & \\ (\mathtt{FM} = \mathtt{var}\ \mathtt{or}\ \ (\mathtt{e} = \mathtt{this}\ \mathtt{and}\ \mathtt{f}\ \notin F)) & \mathtt{e''} \equiv \mathtt{e.f} = \mathtt{e}' \quad \Gamma \vdash \mathtt{e} : \mathtt{C} \quad \mathit{fmodifier}(\mathtt{f},\mathtt{C}) = \mathtt{FM} \\ \mathit{ctorVal}_\Gamma([\mathtt{e}',\overline{\mathtt{e}}],F) & \mathtt{e''} \equiv \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}}) \\ \mathit{ctorVal}_\Gamma([\overline{\mathtt{e}}],F) & \mathtt{e''} \equiv \mathtt{new\ C}\ (\overline{\mathtt{e}}) \\ \mathit{ctorVal}_\Gamma(\mathtt{e},F) & \mathtt{e''} \equiv \mathtt{finish\ e} \\ \mathit{ctorVal}_\Gamma([\mathtt{e},\mathtt{e}'],F) & \mathtt{e''} \equiv \mathtt{async\ e}\ ;\mathtt{e}' \end{cases} \tag{6}$$

We need to check that we do not assign to a `val` field ($\mathit{methodVal}_\Gamma(\mathtt{e})$).

$$\frac{\mathit{methodVal}_\Gamma(\mathtt{e})}{\mathtt{MM\ m}\,(\overline{\mathtt{x}}:\overline{\mathtt{D}}):\mathtt{D} = \mathtt{e}\ \ \mathtt{OK\ IN\ C}}\ \text{(T-METHOD)}$$

Similarly, we need to check for constructors that `val` fields are treated correctly (assigned at most once).

$$\frac{\mathtt{class\ C\ extends\ C}\ \ '\ \{\ \ldots\ \} \qquad F_s = \mathit{fields}(\mathtt{C}') \qquad \mathit{ctorVal}_\Gamma([\overline{\mathtt{e}},\mathtt{e}'],F_s)}{\mathtt{ctor}\ (\overline{\mathtt{x}}:\overline{\mathtt{D}})\{\ \mathtt{super}\ (\overline{\mathtt{e}});\mathtt{e}';\}\ \ \mathtt{OK\ IN\ C}}\ \text{(T-CTOR)}$$

Moreover, *reductionVal* is preserved during the reduction, i.e., a `val` is assigned at most once.
H, `e` closed $\mathit{reductionVal}(\mathtt{e},H)$ (we have everything in H - which val fields are already assigned)

$$\frac{}{\texttt{finish l },H \rightsquigarrow \texttt{l},H}\ \text{(R-FINISH)} \qquad \frac{}{\texttt{async l };\texttt{e},H \rightsquigarrow \texttt{e},H}\ \text{(R-ASYNC)}$$

$$\frac{H(\texttt{l}) = \texttt{C}(\overline{\texttt{f}} \mapsto \overline{\texttt{l}'})}{\texttt{l.f}_i,H \rightsquigarrow \texttt{l}'_i,H}\ \text{(R-FIELD-ACCESS)} \qquad \frac{H(\texttt{l}) = \texttt{C}(F) \qquad F' = F[\texttt{f} \mapsto \texttt{l''}]}{\texttt{l.f} = \texttt{l''},H \rightsquigarrow \texttt{l''},H[\texttt{l} \mapsto \texttt{C}(F')]}\ \text{(R-FIELD-ASSIGN)}$$

$$\frac{\texttt{l''} \notin \text{dom}(H)}{\texttt{new Object }(),H \rightsquigarrow \texttt{l''},H[\texttt{l''} \mapsto \texttt{Object }()]}\ \text{(R-NEW-OBJECT)}$$

$$\frac{ctorbody(\texttt{C}) = \overline{\texttt{x}}.\texttt{super }(\overline{\texttt{e}});\texttt{e''}; \qquad \texttt{class C extends D} \qquad \texttt{new D }([\overline{\texttt{l}}/\overline{x}]\overline{\texttt{e}}),H \rightsquigarrow \texttt{l}',H' \qquad H'[\texttt{l}'] = \texttt{D}(F)}{\texttt{new C }(\overline{\texttt{l}}),H \rightsquigarrow [\overline{\texttt{l}}/\overline{x},\texttt{l}'/\texttt{this }]\texttt{e''},H'[\texttt{l}' \mapsto \texttt{C}(F)]}\ \text{(R-NEW)}$$

$$\frac{H(\texttt{l}') = \texttt{C}(\ldots) \qquad mbody(\texttt{m},\texttt{C}) = \overline{x}.\texttt{e}}{\texttt{l}'.\texttt{m}(\overline{\texttt{l}}),H \rightsquigarrow [\overline{\texttt{l}}/\overline{x},\texttt{l}'/\texttt{this }]\texttt{e},H}\ \text{(R-INVOKE)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{e.m}(\overline{\texttt{e}}),H \rightsquigarrow \texttt{e}'.\texttt{m}(\overline{\texttt{e}}),H'}\ \text{(RC-RECEIVER)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{l.m}(\overline{\texttt{l}},\texttt{e},\overline{\texttt{e}}),H \rightsquigarrow \texttt{l.m}(\overline{\texttt{l}},\texttt{e}',\overline{\texttt{e}}),H'}\ \text{(RC-ARGUMENTS)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{new C }(\overline{\texttt{l}},\texttt{e},\overline{\texttt{e}}),H \rightsquigarrow \texttt{new C }(\overline{\texttt{l}},\texttt{e}',\overline{\texttt{e}}),H'}\ \text{(RC-CTOR)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{e.f},H \rightsquigarrow \texttt{e}'.\texttt{f},H'}\ \text{(RC-FIELD-ACCESS)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{e.f} = \texttt{e''},H \rightsquigarrow \texttt{e}'.\texttt{f} = \texttt{e''},H'}\ \text{(RC-FIELD-ASSIGN1)} \qquad \frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{l.f} = \texttt{e},H \rightsquigarrow \texttt{l.f} = \texttt{e}',H'}\ \text{(RC-FIELD-ASSIGN2)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{async e };\texttt{e''},H \rightsquigarrow \texttt{async e }';\texttt{e''},H'}\ \text{(RC-ASYNC1)} \qquad \frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{async e ''};\texttt{e},H \rightsquigarrow \texttt{async e ''};\texttt{e}',H'}\ \text{(RC-ASYNC2)}$$

$$\frac{\texttt{e},H \rightsquigarrow \texttt{e}',H'}{\texttt{finish e },H \rightsquigarrow \texttt{finish e }',H'}\ \text{(RC-FINISH)}$$

$$\frac{}{(\texttt{async e };\texttt{l}).\texttt{m}(\overline{\texttt{e}}),H \rightsquigarrow \texttt{async e };\texttt{l.m}(\overline{\texttt{e}}),H}\ \text{(RA-RECEIVER)}$$

$$\frac{}{\texttt{l.m}(\overline{\texttt{l}},(\texttt{async e };\texttt{l}'),\overline{\texttt{e}}),H \rightsquigarrow \texttt{async e };\texttt{l.m}(\overline{\texttt{l}},\texttt{l}',\overline{\texttt{e}}),H}\ \text{(RA-ARGUMENTS)}$$

$$\frac{}{\texttt{new C }(\overline{\texttt{l}},(\texttt{async e };\texttt{l}'),\overline{\texttt{e}}),H \rightsquigarrow \texttt{async e };\texttt{new C }(\overline{\texttt{l}},\texttt{l}',\overline{\texttt{e}}),H}\ \text{(RA-CTOR)}$$

$$\frac{}{(\texttt{async e };\texttt{l}).\texttt{f},H \rightsquigarrow \texttt{async e };\texttt{l.f},H}\ \text{(RA-FIELD-ACCESS)}$$

$$\frac{}{(\texttt{async e };\texttt{l}).\texttt{f} = \texttt{e''},H \rightsquigarrow \texttt{async e };\texttt{l.f} = \texttt{e''},H}\ \text{(RA-FIELD-ASSIGN1)}$$

$$\frac{}{\texttt{l.f} = (\texttt{async e };\texttt{l}'),H \rightsquigarrow \texttt{async e };\texttt{l.f} = \texttt{l}',H}\ \text{(RA-FIELD-ASSIGN2)}$$

Figure 3: FX10 Reduction Rules ($H,\texttt{e} \rightsquigarrow H',\texttt{e}'$). Rules (RC-*) handle the congruence rules, and rules (RA-*) handle the concurrent nature of async (bringing the async to the top-level). Note that we do not have an (RA-FINISH) because an async cannot cross a finish.

`F` $::=$ `FM f : C`	Field declaration.
`FM` $::=$ `val` $\mid$ `var`	Field Modifier.

Figure 4: FX10 Syntax changes to support final fields (`val`). The new terminals are `val` and `var`.