Java and Effects

Matthew Parkinson

Matthew.Parkinson@cl.cam.ac.uk

University of Cambridge

(Joint work with Gavin Bierman)
Introduction

In this talk I will cover

- Middleweight Java (MJ)
- Extending MJ with effects
- Effect inference
Why not use FJ?

Featherweight Java (FJ) was developed by Igarashi, Pierce and Wadler.

- it is purely functional
- does not model object identity or state.

but we want to reason about imperative features

- object identity
- aliasing

It is not possible to study this in FJ.
 MJ Aims

Middleweight Java is an *imperative* subset of Java.

- compact
- proper subset
- imperative
- object identity (& nulls)
- block structured scope
We use $f$ for fields, $m$ for methods, $C$ for classes and $x$ for variables.

$$
\begin{align*}
Prog & ::= \text{ClassDef}_1 \ldots \text{ClassDef}_n; \bar{s} \\
\text{ClassDef} & ::= \text{class } C \text{ extends } C' \\
& \quad \{ \text{FieldDef} \; \text{ConsDef} \; \text{MethodDef} \} \\
\text{FieldDef} & ::= C\;f; \\
\text{ConsDef} & ::= C(\bar{C} \; \bar{x})\{\text{super}(\bar{e}); \; \bar{s}\} \\
\text{MethodDef} & ::= \tau \; m(\bar{C} \; \bar{x})\{\bar{s}\} \\
\tau & ::= C|\text{void}
\end{align*}
$$
Syntax

Expression

\[ e ::= x | \text{null} | e.f | (C)e | pe \]

Promotable expression

\[ pe ::= e.m(\overline{e}) | \text{new } C(\overline{e}) \]

Statement

\[ s ::= ; | pe; | \text{if } (e_1 == e_2)\{ \overline{s_1} \} \text{ else } \{ \overline{s_2} \} | e.f = e; | C \ x; | x = e; | \text{return } e; | \{ \overline{s} \} \]
Aside

It is interesting to note that in Java

\[ x++ ; \]

is a valid statement, but

\[ (x++) ; \]

is not. Hence our class of promotable expressions.
class Cell extends Object {
  Object contents;
  Cell(Object start) {
    super();
    this.contents = start;
  }
  void set(Object update) {
    this.contents = update;
  }
}
Types

Expression types

\[ C \] a valid class name, including Object

Statement types

\[ \tau ::= \text{ void} \mid C \]

Method types

\[ \mu ::= C_1, \ldots, C_n \rightarrow \tau \]

return \( e \); is a statement that does not have type void.
Typing of MJ proceeds as

1. Extract typing information from program (We will refer to this by $\Delta$)
2. Check $\Delta$ is well formed (wrt subtyping)
3. Check code is well-typed with respect to $\Delta$. We use $\Delta; \Gamma \vdash s : \tau$

Note: This allows typing of mutually recursive methods to be handled without fixed points.
Java allows variables to be introduced at any point in a sequence of commands. Hence we require two rules for typing a sequence.

\[
\Delta; \Gamma, x : C \vdash s_1 \ldots s_n : \tau \\
\Delta; \Gamma \vdash C \ x; s_1 \ldots s_n : \tau
\]

\[
\Delta; \Gamma \vdash s_1 : \text{void} \quad \Delta; \Gamma \vdash s_2 \ldots s_n : \tau \quad s_1 \neq C \ x; \\
\Delta; \Gamma \vdash s_1 \ s_2 \ldots s_n : \tau
\]
Typing : Stupid Cast

The following is perfectly legal Java.

\[(A)(\text{Object})\text{new } B()\]

but this reduces to \((A)\text{oid where oid : } B\). If \(A\) and \(B\) are unrelated this would not be well-typed. To maintain a subject reduction proof we require a “StupidCast” rule.

\[
\Delta; \Gamma \vdash e : C_2 \quad C_2 \not\prec C_1 \quad C_1 \not\prec C_2
\]

\[
\Delta; \Gamma \vdash (C_1)e : C_1
\]

where \(\prec\) is the subclass relation.
Typing : Super

Expressions in the call to the super constructor can not reference \texttt{this}.

\[
\frac{\Delta; \Gamma \vdash e_1 : C'_1 \quad \ldots \quad \Delta; \Gamma \vdash e_n : C'_n}{\Delta; \Gamma, \texttt{this} : C \vdash \text{super}(e_1, \ldots, e_n); : \texttt{void}}
\]

where \( \Delta_c(C') = C'_1, \ldots, C'_n \),
\( C'_1 \prec C_1, \ldots, C'_n \prec C_n \),
\texttt{this} \notin \text{dom}(\Gamma)
and \( C \prec_1 C'' \).
We define MJ’s operational semantics by transitions between configurations.

Why use framestacks?

- Evaluation context would require an extended language.
- Proofs are given by case analysis rather than induction.
- We like them
Configuration Syntax

A configuration is a four tuple

\[ \text{config} ::= (H, VS, CF, FS) \]

Where the components are

- **H**: Heap
- **VS**: Variable stack
- **CF**: Frame code (being evaluated)
- **FS**: Frame stack
Heap

We define the heap to be:

\[ H ::= \text{map from oids to heap objects} \]

\[ ho ::= (C, F) \]

\[ F ::= \text{map from field names to values} \]

We define values to be:

\[ v ::= \text{null} \mid o \]
We define the variable stack to be a stack of stacks.

Variable stack \( VS \) ::= \( MS \circ VS \| [] \)

Method stack \( MS \) ::= \( BS \circ MS \| [] \)

Block stack \( BS \) ::= map from variables to types and values

We use \( MS \) to represent the scope for a particular method. We require this to be a stack so we can model block structured scoping.
Example

```java
B m(A a) {
    B r;
    {
        A t;
        t = this.getVal();
        r = this.create(t, t);
    }
    return r;
}
```
Frames

\[
F ::= CF | OF
\]
\[
CF ::= s | \text{return } e; | \{ \} | e | \text{super}(e)
\]
\[
OF ::= \text{if } (\bullet == e)\{s_1\} \text{ else } \{s_2\}; | \bullet . f = e;
\]
\[
| \text{if } (v == \bullet)\{s_1\} \text{ else } \{s_2\}; | v.f = \bullet;
\]
\[
| v.m(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)
\]
\[
| \text{new } C(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)
\]
\[
| \text{super}(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)
\]
\[
| x = \bullet; | \text{return } \bullet; | \bullet . m(e) | \bullet . f | (C)\bullet
\]
Reduction Rules

[E-BlockIntro]
\[(H, MS \circ VS, \{\overline{s}\}, FS) \rightarrow (H, (\{\} \circ MS) \circ VS, \overline{s}, (\{\} \circ FS))\]

[E-VarIntro]
\[(H, (BS \circ MS) \circ VS, C x;, FS) \rightarrow (H, (BS[x \leftarrow (\text{null}, C)] \circ MS) \circ VS, ;, FS)\]
where \(x \notin \text{dom}(BS \circ MS)\)

[E-BlockElim]
\[(H, (BS \circ MS) \circ VS, \{\} , FS) \rightarrow (H, MS \circ VS, ;, FS)\]
Java and MJ have certain run time errors that are allowed. Hence we require the following reduction rules.

\[(H, VS, \text{null}.f, FS) \rightarrow \text{NPE}\]
\[(H, VS, \text{null}.f = v, FS) \rightarrow \text{NPE}\]
\[(H, VS, \text{null}.m(v_1, \ldots, v_n), FS) \rightarrow \text{NPE}\]

\[(H, VS, (C)o, FS) \rightarrow \text{CCE}\]

where \(H(o) = (C', \mathbb{F})\) and \(C' \not\in C\)
We then prove type soundness in the standard way using Progress and Type Preservation lemmas.

**Soundness**

If \((H, VS, F, FS) : \tau\) then either
\((H, VS, F, FS) \rightarrow^* (H', VS', F', FS')\) and \((H', VS', F', FS') : \tau'\) where \(\tau' \prec \tau\) or
\((H, VS, F, FS) \rightarrow^* \text{NPE} \vee \text{CCE}\).
Boyland and Greenhouse

The Boyland Greenhouse effect system [ECOOP99]:

- use regions
  - partition fields
  - track reads and writes to regions
  - delimit the scope of effects
  - provide hierarchies of regions
- give annotations to methods
- provide uniqueness analysis of pointers
What we do

We extend their work by

- providing a formal model (by extending MJ)
- proving their system correct
- provide inference algorithm for annotations

However we make two simplifications. We

- remove uniqueness
- remove hierarchies
class Point1D extends Object{
    int x /*in Position*/;
    Point1D(int x)
    /* reads nothing writes Position */
    { this.x = x;  }
    void scale(int n)
    /* reads Position writes Position */
    { this.x = this.x * n;  }
}
class Point2D extends Point1D{
    int y /* in Position*/;
    Point2D(int x, int y)
        /* reads nothing writes Position */
        { super(x); this.y = y; }
    void scale(int n)
        /* reads Position writes Position */
        { this.x = this.x * n;
          this.y = this.y * n; }
}
Extended Syntax

We must extend the syntax of MJ as follows

\[
\begin{align*}
\text{FieldDef} & ::= \quad C \ f \ /\!*\ in \ r \ */; \\
\text{MethodDef} & ::= \quad \tau \ m \ /\!*\ eff \ */(C_1 \ x_1, \ldots, C_n \ x_n) \\
& \quad \{\overline{s}\}; \\
\text{ConsDef} & ::= \quad C \ /\!*\ eff \ */(C_1 \ x_1, \ldots, C_j \ x_j) \\
& \quad \{\text{super}(\overline{e}); \overline{s}\}; \\
\text{eff} & ::= \quad \text{reads} \ \text{reglist} \ \text{writes} \ \text{reglist} \\
\text{reglist} & ::= \quad r_1, \ldots, r_n \mid \text{nothing}
\end{align*}
\]
Effects types

We define effects to be:

\[ E ::= \emptyset | W(r) | R(r) | E \cup E \]

We define subeffecting as:

\[ E_1 \leq E_2 \iff \exists E_3. E_2 = E_1 \cup E_3 \]

We extend the typing judgement to now be

\[ \Delta; \Gamma \vdash e : C ! E \]
Extended Type rules

The type system only requires extensions to the following rules

- Method overriding
- Field Access
- Field Write
- Method Call
- Object Construction

The other rules are extended to union the effects of their premises.
We must instrument the operational semantics to contain the effects of the reduction.

\[(H, VS, F, FS) \xrightarrow{E} (H', VS', F', FS')\]

Only two reduction rules generate effects

- Field Access (read, R(\(r\)))
- Field Write (write, W(\(r\)))
Correctness

We can prove correctness of the effect system by simply extending the Type soundness proof.

**Correctness** If \((H, VS, F, FS) : \tau!E\) then either

\[
(H, VS, F, FS) \xrightarrow{E'} (H', VS', F', FS') \quad \text{and} \quad (H', VS', F', FS') : \tau'!E'' \quad \text{where} \quad \tau' \prec \tau, E' \leq E
\]

and \(E'' \leq E\); or \((H, VS, F, FS) \xrightarrow{E'} \text{NPE} \lor \text{CCE}\)
Inference

Given the region information can the method annotations be inferred?

We extended effects to have variables

\[
E ::= \emptyset | W(r) | R(r) | X | E \cup E
\]

Method:
1. Give each method a fresh variable.
2. Build a set of constraints.
3. Solving constraints finds the annotations.
Example inference

Consider the following method definition.

```java
void m1() {
    this.f1 = new Object();
    this.m2();
}
```

Here we generate the constraint

\[ X_1 \geq W(r1) \cup X_2 \]

We must also give subtyping constraints.
Constraints

The constraints are all of the form:

\[ X \geq E \]

We have at least one constraint for each variable. These can be solved by a single substitution. We have the following two properties:

- There is a minimum solution [Talpin, Jouvelot 92]
- All solutions produce sound annotations
Conclusion

MJ is amenable to proof. We have shown part of the Boyland, Greenhouse effect system to be sound. Their effect system has useful applications:

- Compiler optimisations
- Improving program design

It is key that the system be shown correct if it is to be used. We have also shown that annotations can be inferred, reducing the burden on the programmer.
Future Work

Extend effect system  Adding uniqueness/ownership/alias types to MJ.

Specification Logic  We have also been developing a specification logic for MJ based on the work by Peter O’Hearn et al.

Generics  Extend MJ with generic classes.
Future Work: Generics

- FJ was designed to study generics for Java
- The core of GJ (Generic Java) was proved sound.
- [Dec 2001] Generic extension to Java (based on GJ) shown unsound by Alan Jeffrey
  - Uses mutation (cf ML reference types)
  ⇒ Use MJ to study generics extensions of Java / C#.