Effects and effect inference for a core Java calculus

G.M. Bierman M.J. Parkinson

University of Cambridge Computer Laboratory,
J.J. Thomson Avenue, Cambridge. CB3 0DF. UK.

gmb@cl.cam.ac.uk mjp41@cl.cam.ac.uk

Abstract

An effects system can be used to delimit the scope of computational effects within a program. This information is not only useful for the programmer, but also can be used in the definition of a number of optimizations. Most effects systems have been defined for functional languages with simple state. Greenhouse and Boyland have recently suggested how an effects system could be used within Java. In this paper we take a core imperative calculus for Java, and consider its extension with an effects system, following the suggestions of Greenhouse and Boyland. We define formally the effects system and an instrumented operational semantics and prove the correctness of the effects system; a question left open by Greenhouse and Boyland. We also consider the question of effect inference for our calculus, detailing an algorithm for inferring effects information and prove it correct.

1 Introduction

In order to understand the design of programming languages, and to develop better verification methods for programmers and compiler writers, a common practice is to develop a formal model. This formal model, or calculus, often takes the form of a small, yet interesting fragment of the programming language in question. Recently there has been a number of proposals for a core calculus for the Java programming language. Most notable is Featherweight Java [7], or FJ, which is a core calculus intended to facilitate the study of various aspects of the Java type system, including a proposal for extending Java with generic classes.

In contrast to the main motivation for FJ, we are as much interested in various *operational* properties of Java, as in its type system. To this extent, FJ is an oversimplification as it is simply a *functional* fragment of Java; many of the difficulties with reasoning about Java code arise from its various imperative features. Thus we propose Middleweight Java, or MJ, as a contender for a

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

minimal *imperative* core calculus for Java. MJ can be seen as an extension of FJ big enough to include the essential imperative features of Java; yet small enough that formal proofs are still feasible. In addition to FJ, we model object identity, field assignment, `null` pointers, constructor methods and block structure.

MJ is intended to be a starting point for the study of various operational features of object-oriented programming in Java. To demonstrate this utility, in the majority of this paper, we consider extending the MJ type system with *effects*. An effects system can be used to delimit the scope of computational effects within a program. Effects systems originated in work by Gifford and Lucassen [4], and were pursued by Talpin and Jouvelot [10], amongst others. Interestingly most of these systems were defined for *functional* languages with simple state. [1] Greenhouse and Boyland [5] have recently suggested how an effects system could be incorporated within Java. The key difficulty is the interaction between the effects system and the abstraction facilities (mainly the notion of a class, and also a subclass) that makes Java, and object-oriented programming in general, attractive.

Although Greenhouse and Boyland give a precise description of their effects system and a number of examples, they do not give a proof of correctness. Having formally defined our MJ effects system and instrumented operational semantics we are able to prove its correctness. In addition Greenhouse and Boyland leave the question of effect *inference* to "further work". Again we formally define an algorithm to infer effect annotations and prove it correct. Thus our work in this paper can be seen as both an extension and a formal verification of their proposal: our theory underpins their computational intuitions.

2 MJ: An imperative core Java calculus

In this section we give a brief introduction to MJ, our proposal for an imperative core calculus for Java—full details of MJ, including definitions of the type system, and operational semantics are given in our technical report [1]. We will use MJ as the host language for our effects system in the following sections. It is important to note that MJ is an entirely valid subset of Java, in that all MJ programs are literally executable Java programs; in contrast with, for example, Classic Java, which uses annotations which are not valid Java syntax to formulate the operational semantics [3].

Syntax. The syntax for MJ programs is given in Figure 1. An MJ program is thus a collection of class definitions plus a sequence of statements, $\overline{s}$, to be evaluated. This term corresponds to the body of the `main` method in Java.

[1] Another approach for lazy functional programming with state is the use of *monads* [11]. Wadler has shown that effects systems can easily be adapted to monads [12].

Program

$$p \quad ::= \quad cd_1 \ldots cd_n; \overline{s}$$

Class definition

$$cd \quad ::= \quad \texttt{class}\ C\ \texttt{extends}\ C\{\overline{fd}\ cnd\ \overline{md}\}$$

Field definition

$$fd \quad ::= \quad C\ f;$$

Constructor definition

$$cnd \quad ::= \quad C\ (C_1\ var_1, \ldots, C_j\ var_j)\ \{\texttt{super}(\overline{e});\ \overline{s}\};$$

Method definition

$$md \quad ::= \quad \tau\ m(C_1\ var_1, \ldots, C_n\ var_n)\ \{\overline{s}\};$$

Expression

e	::=	var	Variable
	|	`null`	Null
	|	$e.f$	Field access
	|	$(C)e$	Cast
	|	pe	Promotable expression

Promotable expression

pe	::=	$e.m(\overline{e})$	Method invocation
	|	`new` $C(\overline{e})$	Object creation

Statement

s	::=	;	No-op
	|	pe;	Promoted expression
	|	`if` $(e_1$ `==` $e_2)\{\overline{s_1}\}$ `else` $\{\overline{s_2}\}$	Conditional
	|	$e.f$ `=` e;	Field assignment
	|	$C\ var$;	Local variable declaration
	|	var `=` e;	Variable assignment
	|	`return` e;	Return
	|	$\{\overline{s}\}$	Block

Fig. 1. Syntax for MJ programs

A class definition contains a collection of field and method definitions, and a single constructor definition. A field is defined by a type and a name. Methods are defined as a return type, a method name, an ordered list of

arguments (where an argument is a variable name and type), and a body. A constructor is defined by an ordered list of arguments, the class name, and a body.

For example here are some typical MJ class definitions.

```
class Cell extends Object{
    Object contents;
    Cell (Object start){
        super();
        this.contents = start;
    };
    void set(Object update){
        this.contents = update;
    };
}
```

```
class Recell extends Cell{
    Object undo;
    Recell (Object start){
       super(start);
       this.undo = null;
    };
    void set(Object update){
       this.undo = this.contents;
       this.contents = update;
    };}
```

As with Featherweight Java, we insist on a certain amount of syntactic regularity, although this is really just to make the definitions compact. We insist that all class definitions (1) include a supertype (we assume a distinguished class `Object`); (2) include a constructor method (currently we only allow a single constructor method per class); (3) have a call to `super` as the first statement in a constructor method; (4) have a `return` at the end of every method definition except for `void` methods (this constraint is actually enforced by the type system); and (5) write out field accesses explicitly, even when the receiver is `this`.

In addition we assume that MJ programs are well-formed, i.e. we insist that (1) they do not have duplicate definitions for classes, fields and methods (currently we do not allow overloaded methods for simplicity—as overloading is determined statically, overloaded methods can be simulated faithfully in MJ); and (2) fields are not redefined in subclasses (we do not allow shadowing of fields).

The rest of the definition in Figure 1 defines MJ expressions and statements. We assume a number of metavariables: f ranges over field names, m over method names, and *var* over variables. We assume that the set of variables includes a distinguished variable, `this`, which is not permitted to occur as the name of an argument to a method, or on the left of an assignment. In what follows, we shall find it convenient to write $\overline{e}$ to denote the possibly empty sequence $e_1, \ldots, e_n$ (and similarly for $\overline{C}, \overline{x}$, etc.). We write $\overline{s}$ to denote the sequence $s_1 \ldots s_n$ with no commas (and similarly for $\overline{fd}$ and $\overline{md}$). We abbreviate operations on pairs of sequences in the obvious way, thus for example we write $\overline{C}\,\overline{x}$ for the sequence $C_1\,x_1, \ldots, C_n\,x_n$ where n is the length of $\overline{C}$ and $\overline{x}$.

MJ has two classes of expressions: the class of 'promotable expressions', *pe*, defines expressions that can be can be promoted to statements by postfixing a semicolon '`;`'; the other, *e*, defines the other expression forms. This slighly awkward division is imposed upon us by our desire to make MJ a valid fragment of Java.

The reader will note that the essential imperative features of Java are included in MJ. Thus we have fields, which can be both accessed and assigned to, as well as variables, which can be locally declared and assigned to. As with Java, MJ supports block structure; consider the following code.

```
if(var1 == var2) {
    ;
} else {
    Object temp; temp = var1;
    var1 = var2; var2 = temp;
}
```

This code compares two variables, `var1` and `var2`. If they are not equal then it creates a new locally scoped variable, `temp`, and uses this to swap the values of the two variables. At the end of the block `temp` will no longer be in scope and will be removed from the variable stack.

Types. As with FJ, for simplicity, MJ does not have primitive types, thus all well-typed expressions are of a class type, C. All well-typed statements are of type `void`, except for `return` e; which has the type of e (i.e. a class type). We use τ to range over valid statement types. The type of a method is a pair, written $\overline{C} \to \tau$, where $\overline{C}$ is a sequence of argument types and τ is the return type (if a method does not return anything, its return type is `void`). We use μ to range over method types.

Operational semantics. In our technical report, along with a formal definition of the MJ type system, we give an operational semantics. We then prove a type soundness property, in the style of Wright and Felleisen [13]. The details are rather similar to those in the following sections for the effects system, so we elide them in this short paper. The interested reader is referred to our technical report [1].

3 MJe: A core Java calculus with effects

In this section we define formally our extension of MJ with an effects system. We call the resulting language MJe. This extension follows closely the suggestions of Boyland and Greenhouse [5]. In the rest of this section we begin by giving a brief overview of the key features of the effects system. We then define formally MJe, giving a complete definition of its type system, and an instrumented operational semantics. We conclude by proving the correctness of our effects system, a property left open by Greenhouse and Boyland.

3.1 The Greenhouse Boyland effects system

The *effects* of Java computation includes the reading and writing of mutable state. As Greenhouse and Boyland observe, given some simple assumptions, knowing the read-write behaviour of code enables a number of useful

optimizations of code. The key problem in defining an effects system for an object-oriented language is to preserve the abstraction facilities that make this style of programming attractive.

The first problem is deciding how to describe effects. Declaring the effects of a method should not reveal hidden implementation details. In particular, private field names should not be mentioned. To solve this, Greenhouse and Boyland introduce the notion of a *region* in an object. Thus the regions of an object can provide a covering of the notional state of an object. The read and write effects of a method are then given in terms of the regions that are visible to the caller. (Greenhouse and Boyland introduce two extra notions that we do not address for simplicity: (1) Hierarchies of regions; and (2) unique references of objects.)

Greenhouse and Boyland introduce new syntax for Java to (1) define new regions; (2) to specify which region a field is in; and (3) to specify the read/write behaviour of methods. Rather than introduce new syntax we insist that these declarations are inserted in the appropriate place in the MJe code as comments. (This is similar to the use of commented annotations in Extended Static Checking [2].) For example, here are MJe class declarations for `Point1D` and `Point2D` objects.

```
class Point1D extends Object{
    int x /*in Position*/;
    Point1D(int x) /* reads Position writes Position */
    {   this.x = x;  }
    void scale(int n) /* reads Position writes Position */
    { x = this.x * n; }
}
class Point2D extends Point1D{
    int y /*in Position*/;
    Point2D(int x, int y) /* reads Position 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; }
}
```

Consider the class `Point1D`. This defines a field `x` which is in region `Position`, and a method `scale` that clearly has both read and write effects to that region. Class `Point2D` is a subclass of `Point1D`. It inherits field `x` but also defines a new field `y`, that is also defined to be in region `Position`. It overrides the method `scale`, but with the *same* effects annotation, so it is correct. (Note that this would not have been the case if we simply expressed effects at the level of fields. Then the `scale` method in the `Point2D` class would have *more* effects—it writes both fields `x` and `y`—than the method it is overriding, and so would be invalid. This demonstrates the usefulness of the regions concept.)

3.2 MJe definitions

Syntax. As we have mentioned above, we have chosen not to extend the Java syntax, but rather insist that the effects annotations are contained in comments. This ensures the rather nice property that valid MJe programs are still valid Java programs. Thus the syntax of MJe is exactly the same as for MJ, with the following exceptions, where r ranges over region names.

Field definition

$$fd \quad ::= \quad C\ f\ \texttt{/* in}\ r\ \texttt{*/};$$

Method definition

$$md \quad ::= \quad \tau\ m\ \texttt{/*}\ \mathit{eff}\ \texttt{*/}(C_1\ var_1, \ldots, C_n\ var_n)\ \{\overline{s}\};$$

Constructor definition

$$cnd \quad ::= \quad C\ \texttt{/*}\ \mathit{eff}\ \texttt{*/}(C_1\ var_1, \ldots, C_j\ var_j)\ \{\texttt{super}(\overline{e});\ \overline{s}\};$$

Effect annotation

$$\mathit{eff} \quad ::= \quad \texttt{reads}\ \mathit{reglist}\ \texttt{writes}\ \mathit{reglist}$$

$$\mathit{reglist} \quad ::= \quad r_1, \ldots, r_n \mid \texttt{nothing}$$

Effects. An effect is either empty, $\emptyset$, (written `nothing` in MJe), the union of two effects, written e.g. $E_1 \cup E_2$, or a read effect, $R(r)$, or a write effect, $W(r)$. Equality of effects is modulo the assumption that $\cup$ is commutative, associative and idempotent, and has $\emptyset$ as an unit. A subeffecting relation, $\leq$, is naturally induced on effects: $E_1 \leq E_2 \Leftrightarrow \exists E_3.E_2 = E_1 \cup E_3$. Clearly this relation is reflexive and transitive by definition.

There is a curious subtlety with effects and method overriding. Clearly when overriding a method its effect information must be the same or a subeffect of the overriden method's effect. However, consider the following example (where we have dropped the constructor methods for brevity).

```
class Cell extends Object{
    Object content /* in value */;
    void set(Object update) /*reads nothing writes value*/
    {   this.contents = update;};
}
class Recell extends Cell{
    Object undo /* in value */;
    void set(Object update) /* reads value, writes value */
    {   this.undo = this.contents;this.contents = update;};
}
```

As it stands, `Recell` is not a valid subclass of `Cell` as its `set` method has *more*

effects than in `Cell`. Greenhouse and Boyland [5] solve this by adding $R(r) \leq W(r)$ to the subeffecting relation. To keep the subeffecting relation simple (especially when considering correctness and effect inference) we instead define the effects system such that writing to a field has *both* a read and write effect.

Effects system. We now formally define the effects system. Before presenting the typing rules, we need to define how various typing assumptions are derived from the class definitions. For the most part this is pretty routine: further details can be found in our technical report [1] and the FJ paper [7].

Firstly the class definitions give subclassing information. We write $C_1 \prec_1 C_2$ to denote that C_1 is an immediate subclass of C_2. We define the subclassing relation, $\prec$, to be the reflexive, transitive closure of the immediate subclass relation. We also ensure that for all classes $C, C \prec \texttt{Object}$.

The class table, Δ, represents the remaining typing information extracted from the class definitions. It is defined to be a triple, $(\Delta_m, \Delta_c, \Delta_f)$, which provides typing information about the methods, constructors, and fields, respectively. Δ_m is a partial map from a class name to a partial map from a method name to that method's type and effect. Thus $\Delta_m(C)(m)$ is intended to denote the type and effect of method m in class C. We write a type and effect pair as $\tau!E$. Δ_c is a partial map from class name to the type and effect of that class's constructor method. Δ_f is a partial map from a class name to a map from a field name to a type and a region name. Thus $\Delta_f(C)(f)$ is intended to denote the type and region of field f in class C. The details of Δ are given below.

Method Type

$$\Delta_m(C)(m) \stackrel{\text{def}}{=} \begin{cases} \overline{C} \to \tau!\mathit{effect}(\mathit{eff}) & \text{where } md_i = \tau\ m\ \texttt{/*}\ \mathit{eff}\ \texttt{*/}(\overline{C}\ \overline{\mathit{var}})\{\ldots\} \\ \Delta_m(C')(m) & \text{where } m \notin md_1 \ldots md_n \end{cases}$$

where $\texttt{class}\ C\ \texttt{extends}\ C'\{\overline{fd}\ \mathit{cnd}\ md_1 \ldots md_n\} \in p$

Constructor Type

$$\Delta_c(C) \stackrel{\text{def}}{=} C_1, \ldots, C_j!\mathit{effect}(\mathit{eff})$$

where $\texttt{class}\ C\ \texttt{extends}\ C'\{\overline{fd}\ \mathit{cnd}\ \overline{md}\} \in p$
and $\mathit{cnd} = C\texttt{/*}\ \mathit{eff}\ \texttt{*/}(C_1\ \mathit{var}_1, \ldots, C_j\ \mathit{var}_j)\{\overline{s}\}$

Field Type

$$\Delta_f(C)(f) \stackrel{\text{def}}{=} \begin{cases} (C'', r_i) & \text{where } fd_i = C''\ f\texttt{/*}\ \texttt{in}\ r_i\ \texttt{*/;},\ \text{for some } i, 1 \leq i \leq n \\ & \text{and } \Delta_f(C)(f) \text{ is undefined} \\ \Delta_f(C)(f) & \text{otherwise} \end{cases}$$

where $\texttt{class}\ C\ \texttt{extends}\ C'\{fd_1 \ldots fd_k\ \mathit{cnd}\ \overline{md}\} \in p$

$$\mathit{effect}(\texttt{reads}\ r_1, \ldots, r_n \texttt{writes}\ r_{n+1}, \ldots, r_{n+m}) = R(r_1) \cup \ldots \cup R(r_n) \cup W(r_{n+1}) \cup R(r_{n+1}) \cup \ldots \cup W(r_{n+m}) \cup R(r_{n+m})$$

There are also a number of well-formedness conditions on class definitions, which we elide here. Again they can be found in detail in [1], and similar details can be found in [7]. However, we do give one rule here as it is of particular importance. We need to check that for every method definition, if it overrides a method in its superclass, then it must be both at the *same* type and at a *subeffect* (recall that subeffecting is reflexive). This is captured in the following rule.

$$[\text{T-MethOk1}]\ \frac{\Delta \vdash \mu\ \mathsf{ok}}{\Delta \vdash C.m\ \mathsf{ok}} \quad \begin{array}{l}\text{where } \Delta_m(C)(m) = \mu!E,\ C \prec_1 C',\\ \Delta_m(C')(m) = \mu'!E',\ \mu = \mu' \text{ and } E \leq E'\end{array}$$

We are now in a position to define formally the type and effect system of MJe. Γ is a partial map from program variables to types. We define a typing relation $\Delta; \Gamma \vdash e\colon C!E$ to mean that given a class table Δ, and typing assumptions Γ, an MJe expression e has type C and effects E. (Recall that, Java and hence MJ(e) expressions can have side-effects.) We also define a similar typing relation for MJe statements. These two typing relations are given in Figure 2. Most of these rules are self-apparent. The rules which introduce effects are [TE-FieldAccess] and [TS-FieldWrite]. Notice also that the [TE-Method] and [TE-New] rules both lookup the effect annotation from the class definition. To maintain a subject reduction theorem (that the result of a single step of a well-typed program is also a well-typed program) we require both [TE-StupidCast] and [TS-StupidIf]. The reader should note that a valid Java/MJe program is one that does not have occurrences of [TE-StupidCast] or [TS-StupidIf] in its typing derivation.

We must check that each method and constructor body is correct with respect to the annotations. Here we present the rule for checking a method body is correctly typed.

$$[\text{T-MDefn}]\ \frac{\Delta, \Gamma \vdash \overline{s} : \tau'!E'}{\Delta \vdash \mathit{mbody}(C,m)\ \mathsf{ok}} \quad \begin{array}{l}\text{where } \Gamma = \{\texttt{this} : C, \overline{\mathit{var}} : \overline{C}\},\\ \Delta_m(C)(m) = (\overline{C} \to \tau!E),\ \tau' \prec \tau,\\ \mathit{mbody}(C,m) = (\overline{\mathit{var}}, \overline{s}) \text{and } E' \leq E\end{array}$$

Instrumented operational semantics. We define the operational semantics of MJe in terms of instrumented transitions between *configurations*, in the style of Harper and Stone [6].

$$[\text{TE-Var}]\frac{var : C \in \Gamma \quad \Delta \vdash \Gamma\ \mathsf{ok} \quad \vdash \Delta\ \mathsf{ok}}{\Delta;\Gamma \vdash var : C!\emptyset}$$

$$[\text{TE-Null}]\frac{\Delta \vdash C \quad \Delta \vdash \Gamma\ \mathsf{ok} \quad \vdash \Delta\ \mathsf{ok}}{\Delta;\Gamma \vdash \texttt{null} : C!\emptyset}$$

$$[\text{TE-FieldAccess}]\frac{\Delta;\Gamma \vdash e : C_2!E \quad \Delta_f(C_2)(f) = (C_1, r)}{\Delta;\Gamma \vdash e.f : C_1!E \cup R(r)}$$

$$[\text{TE-UpCast}]\frac{\Delta;\Gamma \vdash e : C_2!E \quad C_2 \prec C_1 \quad \Delta \vdash C_1}{\Delta;\Gamma \vdash (C_1)e : C_1!E}$$

$$[\text{TE-DownCast}]\frac{\Delta;\Gamma \vdash e : C_2!E \quad C_1 \prec C_2 \quad \Delta \vdash C_1}{\Delta;\Gamma \vdash (C_1)e : C_1!E}$$

$$[\text{TE-StupidCast}]\frac{\Delta;\Gamma \vdash e : C_2!E \quad C_2 \not\prec C_1 \quad C_1 \not\prec C_2 \quad \Delta \vdash C_1}{\Delta;\Gamma \vdash (C_1)e : C_1!E}$$

$$[\text{Method}]\frac{\begin{array}{c}\Delta;\Gamma \vdash e : C'!E \quad \Delta;\Gamma \vdash e_1 : C_1!E_1 \quad \ldots \quad \Delta;\Gamma \vdash e_n : C_n!E_n \\ \Delta_m(C')(m) = C'_1, \ldots, C'_n \to C!E' \\ C_1 \prec C'_1 \quad \ldots \quad C_n \prec C'_n\end{array}}{\Delta;\Gamma \vdash e.m(e_1, \ldots, e_n) : C!E \cup E' \cup E_1 \cup \ldots \cup E_n}$$

$$[\text{New}]\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : C'_1!E_1 \quad \ldots \quad \Delta;\Gamma \vdash e_n : C'_n!E_n \\ \Delta_c(C) = C_1, \ldots, C_n!E \\ C'_1 \prec C_1 \quad \ldots \quad C'_n \prec C_n\end{array}}{\Delta;\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C!E \cup E_1 \cup \ldots \cup E_n}$$

$$[\text{TS-NoOp}]\frac{\vdash \Delta \quad \Delta \vdash \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash ; : \texttt{void}!\emptyset} \qquad [\text{TS-PE}]\frac{\Delta;\Gamma \vdash pe : C!E}{\Delta;\Gamma \vdash pe; : \texttt{void}!E}$$

$$[\text{TS-If}]\frac{\begin{array}{c}\Delta;\Gamma \vdash \overline{s_1} : \texttt{void}!E_1 \qquad \Delta;\Gamma \vdash e_1 : C'!E_3 \\ \Delta;\Gamma \vdash \overline{s_2} : \texttt{void}!E_2 \quad \Delta;\Gamma \vdash e_2 : C''!E_4 \quad C' \prec C'' \vee C' \succ C''\end{array}}{\Delta;\Gamma \vdash \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\} : \texttt{void}!E_1 \cup E_2 \cup E_3 \cup E_4}$$

$$[\text{TS-StupidIf}]\frac{\begin{array}{c}\Delta;\Gamma \vdash \overline{s_1} : \texttt{void}!E_1 \qquad \Delta;\Gamma \vdash e_1 : C'!E_3 \\ \Delta;\Gamma \vdash \overline{s_2} : \texttt{void}!E_2 \quad \Delta;\Gamma \vdash e_2 : C''!E_4 \quad C'' \not\prec C' \wedge C' \not\prec C''\end{array}}{\Delta;\Gamma \vdash \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\} : \texttt{void}!E_1 \cup E_2 \cup E_3 \cup E_4}$$

$$[\text{TS-FieldWrite}]\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : C_1!E_1 \quad \Delta_f(C_1)(f) = (C_3, r) \\ \Delta;\Gamma \vdash e_2 : C_2!E_2 \qquad C_2 \prec C_3\end{array}}{\Delta;\Gamma \vdash e_1.f = e_2; : \texttt{void}!E_1 \cup E_2 \cup W(r) \cup R(r)}$$

$$[\text{TS-VarWrite}]\frac{\Delta;\Gamma \vdash var : C!E_1 \quad \Delta;\Gamma \vdash e : C'!E_2 \quad C' \prec C}{\Delta;\Gamma \vdash var = e; : \texttt{void}!E_1 \cup E_2}$$

$$[\text{TS-Return}]\frac{\Delta;\Gamma \vdash e : C!E}{\Delta;\Gamma \vdash \texttt{return}\ e; : C!E}$$

$$[\text{TS-Block}]\frac{\Delta;\Gamma \vdash s_1 \ldots s_n : \texttt{void}!E}{\Delta;\Gamma \vdash \{s_1 \ldots s_n\} : \texttt{void}!E}$$

$$[\text{TS-Seq}]\frac{\Delta,\Gamma \vdash s_1 : \texttt{void} \quad \Delta,\Gamma \vdash s_2 \ldots s_n : \tau \quad s_1 \neq C\ var;}{\Delta,\Gamma \vdash s_1\ s_2 \ldots s_n : \tau}$$

$$[\text{TS-Intro}]\frac{\Delta,\Gamma, var : C \vdash s_1 \ldots s_n : \tau}{\Delta,\Gamma \vdash C\ var; s_1 \ldots s_n : \tau}$$

Fig. 2. Effects system for MJe

Configuration

$$\mathit{config} ::= (H, VS, CF, FS)$$

Frame Stack

$$FS ::= F \circ FS \mid [\,]$$

Frame

$$F ::= CF \mid OF$$

Closed frame

$$CF ::= \overline{s} \mid \texttt{return}\ e; \mid \{\ \} \mid e \mid \texttt{super}(\overline{e})$$

Open frame

$$\begin{aligned} OF ::= {} & \texttt{if}\ (\bullet == e)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\}; \\ \mid {} & \texttt{if}\ (v == \bullet)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\}; \\ \mid {} & \bullet.f \mid \bullet.f = e; \mid v.f = \bullet; \mid (C)\bullet \\ \mid {} & v.m(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n) \\ \mid {} & \texttt{new}\ C(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n) \\ \mid {} & \texttt{super}(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n) \\ \mid {} & x = \bullet; \mid \texttt{return}\ \bullet; \mid \bullet.m(\overline{e}) \end{aligned}$$

Values

$$v ::= \texttt{null} \mid o$$

Variable Stack

$$VS ::= MS \circ VS \mid [\,]$$

Method Scope

$$MS ::= BS \circ MS \mid [\,]$$

Block Scope

BS ::= is a finite partial function from variables to pairs of expression types and values

Heap

H ::= is a finite partial function from oids to heap objects

Heap Objects

$$ho ::= (C, \mathbb{F})$$

$\mathbb{F}$ ::= is a finite partial function from field names to values

Fig. 3. Syntax for MJ Configurations

A configuration is a four-tuple containing the following information:

(i) **Heap**: A finite partial function that maps object ids (oids) to heap objects, where a heap object is a pair of a class name, C, and a field function, $\mathbb{F}$. The field function is a partial map from field names to values. A *value* is either the `null` object or an oid.

(ii) **Variable Stack**: This essentially maps variable names to values. To handle static, block structured scoping it is implemented as a list of lists of partial functions from variables to values. We use $\circ$ to denote stack concatenation.

(iii) **Term**: The term to be evaluated

(iv) **Frame stack**: This represents the program context in which the term is evaluated.

A more formal description can be found in Figure 3.

Thus *CF* is a closed frame (i.e. with no hole) and *OF* is an open frame (i.e. requires an expression to be substituted in for the hole). *VS* is the Variable Stack and consists of a stack of Method Scopes, *MS*. These in turn consist of a stack of Block Scopes, *BS*, which are finite partial maps from variable names to(type, value) pairs.

We find it useful to define two operations on method scopes, in addition to the usual list operations. The first, $eval(MS, var)$, evaluates a variable, var in a method scope, MS. This is a partial function and is only defined if the variable name is in the scope. The second, $update(MS, var \mapsto v)$, updates a method scope MS with the value v for the variable var. Again this is a partial function and is undefined if the variable is not in the scope. The rather routine definitions of these two functions are elided here.

We shall write an instrumented transition as $(H, MS, CF, FS) \xrightarrow{E} (H', MS', CF', FS)$ to mean that there is a one-step reduction between the two configurations, which also has the effect E. As is common with these transition rule systems, there are a number of rather routine bookkeeping rules, that deal with the frame stack. These are given in Appendix A.1. In Figure 4 and 5 we give the significant transition rules. In later sections we will make use of the $\xrightarrow{E}{}^*$ relation, which is the reflexive, transitive closure of the one-step reduction relation, that 'unions up' the effects in the obvious way.

We give a few brief comments on some of these transition rules. A fuller discussion can be found in our technical report [1]. The side condition in the [E-VarWrite] rule ensures that we can only write to variables declared in the current method scope. The [E-VarIntro] rule follows Java's restriction that a variable declaration can not hide an earlier declaration within the current method scope.[2] (Note also how the rule defines the binding for the new variable in the current *block* scope.)

We have a number of rules for constructing and removing scopes. The first, [E-BlockIntro], introduces a new block scope, and leaves a token on the frame-stack. The second, [E-BlockElim], removes the token and the outermost block scope. The final rule, [E-Return], leaves the scope of a method, by removing the top scope, MS.

Rule [E-Cast] simply ensures that the cast is valid (if it is not, the program should enter an error state—these states are covered below). Rule [E-NullCast] simply ignores any cast of a `null` object.

The [E-New] rule creates a fresh oid, o, and places on the heap a heap object with class C and assigns all the fields to `null`. As we are executing a new constructor method, a new method scope is created and added on to the variable stack. This method scope initially consists of just one block scope, that consists of bindings for the method parameters, and also a binding for the `this` identifier. The method body B is then the next term to be executed, but importantly the continuation `return` o; is placed on the frame stack. This is because the result of this statement is the oid of the object, and the method scope is removed.

Finally let us consider the transition rule for method invocation. Invocation is relatively straightforward: although note that a new method scope is created, consisting of just the bindings for the method parameters and the

[2] This sort of variable hiding is, in contrast, common in functional languages such as SML.

[E-VarAccess] $(H, MS \circ VS, var, FS) \xrightarrow{\emptyset} (H, MS \circ VS, v, FS)$
where $eval(MS, var) = (v, C)$

[E-VarWrite] $(H, MS \circ VS, var = v;, FS) \xrightarrow{\emptyset} H, (update(MS, (var \mapsto v))) \circ VS, ;, FS)$
where $eval(MS, var) \downarrow$

[E-VarIntro] $(H, (BS \circ MS) \circ VS, C\ var;, FS) \xrightarrow{\emptyset} (H, (BS' \circ MS) \circ VS, ;, FS)$
where var not in $dom(BS \circ MS)$ and $BS' = BS[var \mapsto (\texttt{null}, C)]$

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

[E-BlockElim] $(H, (BS \circ MS) \circ VS, \{\ \}, FS) \xrightarrow{\emptyset} (H, MS \circ VS, ;, FS)$

[E-Return] $(H, MS \circ VS, \texttt{return}\ v;, FS) \xrightarrow{\emptyset} (H, VS, v, FS)$

[E-If1] $(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS) \xrightarrow{\emptyset} (H, VS, \{\overline{s_1}\}, FS)$
if $v_1 = v_2$

[E-If2] $(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS) \xrightarrow{\emptyset} (H, VS, \{\overline{s_2}\}, FS)$
if $v_1 \neq v_2$

[E-FieldAccess] $(H, VS, o.f, FS) \xrightarrow{R(r)} (H, VS, v, FS)$
where $o \in dom(H)$, $H(o) = (C, \mathbb{F})$, $\mathbb{F}(f) = v$ and $\Delta_f(C)(f) = (C', r)$

[E-FieldWrite] $(H, VS, o.f = v;, FS) \xrightarrow{W(r)} (H', VS, ;, FS)$
where $H(o) = (C, \mathbb{F})$, $f \in dom(\mathbb{F})$, $\Delta_f(C)(f) =) C', r)$ and $H' = H[o \mapsto (C, \mathbb{F}')]$

[E-Cast] $(H, VS, ((C_2)o), FS) \xrightarrow{\emptyset} (H, VS, o, FS)$
where $H(o) = (C_1, \mathbb{F})$ and $C_1 \prec C_2$

[E-NullCast] $(H, VS, ((C)\texttt{null}), FS) \xrightarrow{\emptyset} (H, VS, \texttt{null}, FS)$

Fig. 4. Instrumented operational semantics for MJe (Part I)

`this` identifier. We require two rules as the `void` method requires an addition to the stack to clear the new method scope once the method has completed. The last statement in [E-Method] rule must be a `return` if the method is well typed. We use a separate rule for `void` methods, [E-MethodVoid], as they do not contain a `return` at the end.

A number of transitions lead to a predictable error state. These are errors that are allowed at run-time as they are dynamically checked for by the Java Virtual Machine. Java's type system is not capable of removing these errors statically. The two errors that can be generated are NullPointerException (**NPE**) and ClassCastException (**CCE**).

[E-New] $(H, VS, \texttt{new}\ C(\overline{v}), FS) \xrightarrow{\emptyset} (H[o \mapsto (C, \mathbb{F})], (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$

where $cnbody(C) = (\overline{var}, \overline{s})$, $\Delta_c(C) = \overline{C}$, o not in $dom(H)$, $\mathbb{F} = \{f \mapsto \texttt{null}\}\ \forall f \in fields(C)$ and $BS = \{\texttt{this} \mapsto (o, C), \overline{var} \mapsto (\overline{v}, \overline{C})\}$

[E-Super] $(H, MS \circ VS, \texttt{super}(\overline{v}), FS) \xrightarrow{\emptyset} (H, (BS' \circ []) \circ (MS \circ VS), \overline{s}, FS)$

where $MS(\texttt{this}) = (o, C), C \prec_1 C'$, $BS' = \{\texttt{this} \mapsto (o, C'), \overline{var} \mapsto (\overline{v}, \overline{C})\}$, $\Delta_c(C) = \overline{C}$ and $cnbody(C') = (\overline{var}, \overline{s})$

[E-Method] $(H, VS, o.m(\overline{v}), FS) \xrightarrow{\emptyset} (H, (BS \circ []) \circ VS, \overline{s}, FS)$

where $mbody(C, m) = (\overline{var}, \overline{s})$, $H(o) = (C, \mathbb{F})$, $\Delta_m(C)(m) = \overline{C} \to C'\ !\ E$ and $BS = \{\texttt{this} \mapsto (o, C), \overline{var} \mapsto (\overline{v}, \overline{C})\}$

[E-MethodVoid] $(H, VS, o.m(\overline{v}), FS) \xrightarrow{\emptyset} (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$

where $H(o) = (C, \mathbb{F})$, $\Delta_m(C)(m) = \overline{C} \to \texttt{void}\ !\ E$, $mbody(C, m) = (\overline{var}, \overline{s})$ and $BS = \{\texttt{this} \mapsto (o, C), \overline{var} \mapsto (\overline{v}, \overline{C})\}$

[E-Skip] $(H, VS, ;, F \circ FS) \xrightarrow{\emptyset} (H, VS, F, FS)$

[E-Sub] $(H, VS, v, F \circ FS) \xrightarrow{\emptyset} (H, VS, F[v] \circ FS)$

[E-NullField] $(H, VS, \texttt{null}.f, FS) \xrightarrow{\emptyset} \mathbf{NPE}$

[E-NullWrite] $(H, VS, \texttt{null}.f = v, FS) \xrightarrow{\emptyset} \mathbf{NPE}$

[E-NullMethod] $(H, VS, \texttt{null}.m(v_1, \ldots, v_n), FS) \xrightarrow{\emptyset} \mathbf{NPE}$

[E-InvCast] $(H, VS, (C)o, FS) \xrightarrow{\emptyset} \mathbf{CCE}$

where $H(o) = (C_1, \mathbb{F})$ and $C_1 \not\prec C$

Fig. 5. Instrumented operational semantics for MJe

A configuration is said to be *terminal* if it is a valid error (**NPE** or **CCE**) or it is of the form $(H, VS, v, [])$. We can extend our type system to configurations, and write judgements of the form $\Delta \vdash (H, VS, t, FS) : \tau ! E$. Clearly this typing relation is defined by checking that all four components are well-typed and that all the effects of t and FS are contained in E. The rather routine details are given in Appendix §A.2.

Correctness. Our main technical contribution is a proof of correctness of the effects system of MJe. As we have mentioned earlier, this was not addressed by Greenhouse and Boyland. In order to prove correctness we first prove two useful propositions in the style of Wright and Felleisen [13].

The first proposition states that any well typed non-terminal configuration can both make a reduction step, and that any resulting effect is contained in the effects inferred by the effects system.

Proposition 3.1 (Progress) *If* $(H, VS, F, FS) : \tau!E$ *then* $\exists H', VS', F', FS'.(H, VS, F, FS) \xrightarrow{E'} (H', VS', F', FS')$ *and* $E' \leq E$.

Proof. By induction on the typing derivation of a configuration. Some details are given in Appendix B.1. □

Next we find it useful first to prove the following lemma, which states that subtyping on frame stacks is *covariant*.

Lemma 3.2 (Covariant subtyping of frame stack with effects) $\forall H, VS, \tau_1, \tau_2, \tau_3, E.$ *if* $\Delta, H, VS \vdash FS : \tau_1 \to \tau_2!E_1$ *and* $\tau_3 \prec \tau_1$ *then* $\exists \tau_4, E_2.$ $\Delta, H, VS \vdash FS : \tau_3 \to \tau_4!E_2$, $\tau_4 \prec \tau_2$ *and* $E_2 \leq E_1$.

Proof. *By induction on the length of FS. Note we only have to consider open frames as all closed frames ignore their argument. Appendix B.2 contains some further details.* □

We can now prove the second important proposition, which states that if a configuration can make a transition, then the resulting configuration is of the appropriate type and effect.

Proposition 3.3 (Type Preservation) *If* $(H, VS, F, FS) : \tau!E_1$ *and* $(H, VS, F, FS) \to (H', VS', F', FS')$ *then* $\exists \tau'.(H', VS', F', FS') : \tau'!E_2$ *where* $\tau' \prec \tau$ *and* $E_2 \leq E_1$.

Proof. By case analysis on the reduction step. Lemma 3.2 is needed for the reduction rules that generate subtypes. Appendix B.3 contains further details of this proof. □

We can now combine the two propositions to prove the correctness of the MJe effects system.

Theorem 3.4 (Correctness) *If* $(H, VS, F, FS) : \tau!E$ *then either* $(H, VS, F, FS) \xrightarrow{E'}{}^* (H', VS', F, FS')$ *and* $(H', VS', F', FS') : \tau'!E''$ *where* $\tau' \prec \tau$, $E' \leq E$ *and* $E'' \leq E$; *or* $(H, VS, F, FS) \xrightarrow{E'}{}^* \boldsymbol{NPE} \vee \boldsymbol{CCE}$.

4 Effect inference

In the previous section we defined an effects system, where fields are declared to be in abstract regions and methods annotated with their read/write behaviour with respect to these regions, and proved its correctness. The obvious question, not addressed by Greenhouse and Boyland, is whether the method effects can be *inferred* automatically, assuming that fields have been 'seeded' with their regions. In this section we demonstrate briefly that this is possible. We give an outline of an algorithm which is proved correct.

Our approach automatically generates the most general annotations for each method and constructor. We first extend the grammar for effects with variables, where X ranges over these effects variables. We then modify the effects system so that it generates a series of *constraints*. A constraint is written $E \leq X$ which is intended to mean that effect X is at least the effects E. A constraint set is then a set of such constraints.

A *substitution*, θ, is a function that maps effects variables to effects. It can be extended pointwise over effects. We say that a substitution θ *satisfies* a constraint $E \leq X$ if $\theta(E) \leq \theta(X)$. The key to our inference algorithm is the generation of two constraint sets. The first arises from the effects in the bodies of the methods and constructors; the second from subtyping.

The reader will recall that we generate a class table from the class definitions. Clearly we need to extend this notion, as now methods and constructors do not come with effect annotations. Instead we generate a *fresh* effect variable to represent their effects.

$$\Delta_m(C)(m) = \begin{cases} C_1, \ldots, C_j \to C''!X & \text{where } md_i = C''m(C_1 var_1, \ldots, C_j var_j)\{\ldots\} \\ \Delta_m(C')(m) & \text{where } m \notin md_1 \ldots md_n \end{cases}$$

where $\texttt{class}\ C\ \texttt{extends}\ C'\{fd_1 \ldots fd_k\, cnd\, md_1 \ldots md_n\} \in p$ and X is a fresh effect variable.

We then typecheck each method body and constructor body as before. For example assume that we have deduced the following for the body of method m in class C:

$$\Delta; \Gamma \vdash \overline{s}\ \texttt{return}\ e; : C'!E$$

Assume that in Δ we have the effect of method m as X. This means that we should generate the constraint $E \leq X$. We simply repeat this process for all method bodies and constructor bodies to generate a set of constraints. We write R_d for this constraint set.

So far our constraints do not reflect the subeffecting requirement on overridden methods. We simply adjust the well-formedness conditions on method definitions to generate these additional constraints. For example, the [T-MethOK1] rule, originally given in §3.2, is now as follows.

$$[\text{T-MethOk1}]\ \frac{\Delta \vdash \mu\ \mathsf{ok}}{\Delta \vdash_i C.m : X \leq Y} \quad \text{where} \quad \begin{array}{l} \Delta_m(C)(m) = \mu!X,\ C \prec_1 C' \\ \Delta_m(C')(m) = \mu'!Y \text{ and } \mu = \mu' \end{array}$$

Thus we extend our well-formedness rules so that they generate a set of constraints, R_s. We write this as $\vdash \Delta : R_s$, to mean that the program corresponding to the class table Δ generates constraint set R_S. We also write $\Delta \vdash p : R_d$ to mean that the program p generates the constraint set R_d. We

now have a constraint set $R_{prog} = R_d \cup R_s$ that we need to solve. We note immediately that these constraints need not have a unique solution. Consider the following excerpt from a class definition.

```
int counter; /* in r */
void count(int x) {
   this.counter=x; this.count(x-1);
}
```

Assume that the method `count` is assigned the effect variable X as its effects annotation. From the typing judgement for the method body we produce the constraint:

$$W(r) \cup R(r) \cup X \leq X$$

Clearly there are infinitely many solutions to this constraint. However the *minimium* solution is what is needed (in this case it is the substitution $\{X \mapsto W(r) \cup R(r)\}$).

Lemma 4.1 (Existence of minimum solution) *Given a constraint set $\{E_1 \leq X_1, \ldots, E_n \leq X_n\}$ there is a unique, minimal solution.*

Proof. A proof was given by Talpin and Jouvelot [10]. They also give an algorithm for finding the minimum solution of a set of constraints. □

Lemma 4.2 (Substitutions satisfy subtyping) *If $\vdash \Delta : R_s$ and θ satisfies R_s then $\vdash \theta(\Delta)$ ok.*

The next lemma states that effect substitutions preserve typing judgements.

Lemma 4.3 (Effect substitution preserves typing) *If $\Delta; \Gamma \vdash t : C!E$, $\vdash \Delta : R_s$ and θ satisfies R_s then $\theta(\Delta); \Gamma \vdash t : C!\theta(E)$ where t ranges over expressions and statements.*

Proof. By induction on the typing relation.

□

From these lemmas we can prove the correctness of the effect inference algorithm: substitutions that satisfy the constraint sets yield sound effect annotations and the algorithm generates the unique minimal such substitution.

Theorem 4.4 (Inference produces sound effect annotations) *If $\Delta \vdash p : R_d$, $\vdash \Delta : R_s$ and θ satisfies $R_d \cup R_s$ then $\theta(\Delta) \vdash p$ ok.*

Proof. This follows from the definitions and repeated use of Lemma 4.3 see Appendix §B.4 for more details. □

5 Conclusions and future work

In this paper we propose Middleweight Java, or MJ, as a contender for an *imperative* core calculus for Java. We claim that it captures most of the complicated imperative features of Java, but is compact enough to make rigorous

proofs feasible. To justify this claim we considered its extension with an effects system due to Greenhouse and Boyland [5]. We formally defined the effects system and an instrumented operational semantics, and we proved the correctness of the effects systems (a question not addressed by Greenhouse and Boyland). We then considered the question of effects *inference*, namely the inference of the effects in the method and constructor bodies. We defined an algorithm and proved its correctness.

Clearly further work remains. In terms of the effects system, we are currently investigating extending MJe with two other properties suggested by Greenhouse and Boyland; namely hierarchies of regions, and alias types. It remains to be seen if our proofs of correctness can be easily adapted to this richer setting.

In other work, we are developing a logic for reasoning about MJ programs, based on the bunched logic approach pioneered by O'Hearn, Reynolds and Yang [8,9].

Acknowledgements

We should like to thank our co-author Andrew Pitts for numerous discussions. We acknowledge funding from EPSRC (Parkinson) and from EU: PEPITO and APPSEM II (Bierman).

References

[1] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge Computer Laboratory, to appear.

[2] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.

[3] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR-97-293, Rice University, 1997. Corrected June, 1999.

[4] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *Proceedings of ACM Lisp and Functional Programming*, 1986.

[5] A. Greenhouse and J. Boyland. An object-oriented effects system. In *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229, 1999.

[6] R. Harper and C. Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, Computer Science Department, Carnegie Mellon University, 1997.

[7] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[8] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, 2001.

[9] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.

[10] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[11] P. Wadler. The essence of functional programming. In *Proceedings of Principles of Programming Languages*, 1992.

[12] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, 1998.

[13] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Definitions

A.1 Framestack booking rules

The following rules define the way in which the frame stack is manipulated. The rules in this section do not perform any evaluation they just decompose terms.

[EC-Seq]	$(H, VS, s_1\ s_2 \ldots s_n, FS) \xrightarrow{\emptyset} (H, VS, s_1, (s_2 \ldots s_n) \circ FS)$
[EC-Return]	$(H, MS \circ VS, \texttt{return}\ e;, FS)$ $\xrightarrow{\emptyset} (H, MS \circ VS, e, (\texttt{return}\ \bullet;) \circ FS)$
[EC-ExpState]	$(H, VS, e';, FS) \xrightarrow{\emptyset} (H, VS, e', FS)$
[EC-If1]	$(H, VS, \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};, FS)$ $\xrightarrow{\emptyset} (H, VS, e_1, (\texttt{if}\ (\bullet == e_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\};) \circ FS)$
[EC-If2]	$(H, VS, \texttt{if}\ (v_1 == e_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\};, FS)$ $\xrightarrow{\emptyset} (H, VS, e_2, (\texttt{if}\ (v_1 == \bullet)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\};) \circ FS)$
[EC-FieldAccess]	$(H, VS, e.f, FS) \xrightarrow{\emptyset} (H, VS, e, (\bullet.f) \circ FS)$
[EC-Cast]	$(H, VS, (C)e, FS) \xrightarrow{\emptyset} (H, VS, e, ((C)\bullet) \circ FS)$
[EC-FieldWrite1]	$(H, VS, e_1.f = e_2;, FS) \xrightarrow{\emptyset} (H, VS, e_1, (\bullet.f = e_2;) \circ FS)$
[EC-FieldWrite2]	$(H, VS, v_1.f = e_2;, FS) \xrightarrow{\emptyset} (H, VS, e_2, (v_1.f = \bullet;) \circ FS)$
[EC-VarWrite]	$(H, VS, var = e;, FS) \xrightarrow{\emptyset} (H, VS, e, (var = \bullet;) \circ FS)$
[EC-New]	$(H, VS, \texttt{new}\ C(v_1, \ldots, v_{i-1}, e_i, \ldots e_n), FS)$ $\xrightarrow{\emptyset} (H, VS, e_i, (\texttt{new}\ C(v_1, \ldots, v_{i-1}, \bullet, \ldots e_n)) \circ FS)$
[EC-Super]	$(H, VS, \texttt{super}(v_1, \ldots, v_{i-1}, e_i, \ldots e_n), FS)$ $\xrightarrow{\emptyset} (H, VS, e_i, (\texttt{super}(v_1, \ldots, v_{i-1}, \bullet, \ldots e_n)) \circ FS)$
[EC-Method1]	$(H, VS, e.m(e_1, \ldots, e_n), FS) \xrightarrow{\emptyset} (H, e, (\bullet.m(e_1, \ldots, e_n)) \circ FS)$
[EC-Method2]	$(H, VS, v.m(v_1, \ldots, v_{i-1}, e_i, \ldots e_n), FS)$ $\xrightarrow{\emptyset} (H, VS, e_i, (v.m(v_1, \ldots, v_{i-1}, \bullet, \ldots e_n)) \circ FS)$

Fig. A.1. MJ decomposition reduction rules

A.2 Configuration typing rules

In this section we give the rules for checking a configuration is well typed.

A.2.1 Expressions

For frame stacks we extend the syntax of expression to contain both object identifiers (o) and "holes" ($\bullet$). Thus, we also require Γ to be extended to additionally map holes and object identifiers to values. We require the following two additional rules to type expression in frame stacks.

$$[\text{TE-OID}]\frac{o : C \in \Gamma \quad \Delta \vdash \Gamma\ \mathsf{ok} \quad \vdash \Delta\ \mathsf{ok}}{\Delta, \Gamma \vdash o : C}$$

$$[\text{TE-Hole}]\frac{\bullet : C \in \Gamma \quad \Delta \vdash \Gamma\ \mathsf{ok} \quad \vdash \Delta\ \mathsf{ok}}{\Delta, \Gamma \vdash \bullet : C}$$

A.2.2 Context Collapsing

The following inductive definition shows how a heap and variable scope are collapsed to give a typing environment.

$$\begin{aligned}
\mathit{context}(\{\}, [\,]) &\stackrel{\text{def}}{=} \{\} \\
\mathit{context}(\{\}, (\{\} \circ MS) \circ VS) &\stackrel{\text{def}}{=} \mathit{context}(\{\}, MS \circ VS) \\
\mathit{context}(\{\}, (BS[x \mapsto v, C] \circ MS) \circ VS') &\stackrel{\text{def}}{=} \mathit{context}(\{\}, (BS \circ MS) \circ VS) \uplus \{x \mapsto C\} \\
&\quad \text{where } x \notin \mathit{dom}(BS) \text{ and } x \notin \mathit{dom}(\mathit{context}(\{\}, (BS \circ MS) \circ VS) \\
\mathit{context}(H[o \mapsto C, \mathbb{F}], VS) &\stackrel{\text{def}}{=} \mathit{context}(H, VS) \uplus \{o \mapsto C\} \\
&\quad \text{where } o \notin \mathit{dom}(H)
\end{aligned}$$

A.2.3 Heap Typing

A heap is well typed if every field points to a valid value of the required type.

$$[\textsc{ObjectTyped}]\frac{H(o) = (C, \mathbb{F}) \quad C \prec \tau \quad \Delta \vdash C}{\Delta, H \vdash o : \tau}$$

$$[\textsc{NullTyped}]\frac{\Delta \vdash C}{\Delta, H \vdash \texttt{null} : C}$$

$$[\textsc{ObjectOK}]\frac{\Delta, H \vdash \mathbb{F}(f_i) : \Delta_f(C)(f_i) \quad \forall i.1 \leq i \leq n}{\Delta, H \vdash o\ \mathsf{ok}} \quad \begin{array}{l}\text{where } H(o) = (C, \mathbb{F}), \\ \mathit{dom}(\Delta_f(C)) = f_1 \ldots f_n\end{array}$$

$$[\textsc{HeapOk}]\frac{\Delta, H \vdash o_1\ \mathsf{ok} \quad \ldots \quad H \vdash o_2\ \mathsf{ok}}{\Delta \vdash H\ \mathsf{ok}} \quad \text{where } \mathit{dom}(H) = \{o_1, \ldots, o_n\}$$

A.2.4 Local State Typing

For the local state to be valid every variable must point to a valid value of the correct type.

$$[\textsc{VarBS}]\frac{\Delta, H \vdash v_1 : C_1 \quad \ldots \quad \Delta, H \vdash v_n : C_n}{\Delta, H \vdash BS\ \mathsf{ok}}$$

where $BS = \{var_1 \mapsto (v_1, C_1), \ldots, (var_n \mapsto (v_n, C_n)\}$

$$[\textsc{VarStackEmpty}]\frac{}{\Delta, H \vdash [\,]\ \mathsf{ok}} \qquad [\textsc{VarMSEmpty}]\frac{\Delta, H \vdash VS\ \mathsf{ok}}{\Delta, H \vdash [\,] \circ VS\ \mathsf{ok}}$$

$$[\textsc{VarStack}]\frac{\Delta, H \vdash BS\ \mathsf{ok} \quad \Delta, H \vdash MS \circ VS\ \mathsf{ok}}{\Delta, H \vdash (BS \circ MS) \circ VS\ \mathsf{ok}}$$

A.2.5 Frame stack typing

For the frame stack to be well typed we require each frame is well typed with respect to the environment.

$$[\text{TF-}\textsc{StackBlock}]\frac{\Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau!E}{\Delta, H, (BS \circ MS) \circ VS \vdash (\{\}) \circ FS : \tau' \to \tau!E}$$

$$[\text{TF-}\textsc{StackMethod}]\frac{\Delta, H, VS \vdash FS : \tau \to \tau'!E}{\Delta, H, (BS \circ [\,]) \circ VS \vdash (\texttt{return}\ \bullet;) \circ FS : \tau \to \tau'!E}$$

$$[\text{TF-}\textsc{StackMethod2}]\frac{\Delta; context(H, MS \circ VS) \vdash e : \tau!E_1 \quad H, VS \vdash FS : \tau \to \tau'!E_2}{\Delta, H, MS \circ VS \vdash (\texttt{return}\ e;) \circ FS : \tau'' \to \tau'!E_1 \cup E_2}$$

$$[\text{TF-}\textsc{Sequence}]\frac{\Delta, H, VS \vdash (s_1) \circ (s_2 \ldots s_n) \circ FS : \tau \to \tau'!E}{\Delta, H, VS \vdash (s_1 s_2 \ldots s_n) \circ FS : \tau \to \tau'!E}$$

$$[\text{TF-}\textsc{StackIntro}]\frac{\Delta, H, (BS[var : C] \circ MS) \circ VS \vdash FS : \texttt{void} \to \tau!E}{\Delta, H, (BS \circ MS) \circ VS \vdash (C\ var;) \circ FS : \tau' \to \tau!E}$$

where $var \notin dom(BS \circ MS)$

$$[\text{TF-}\textsc{StackOpen}]\frac{\Delta; context(H, VS), \bullet : \tau'' \vdash F : \tau!E_1 \quad H, VS \vdash FS : \tau \to \tau'!E_2}{\Delta, H, VS \vdash OF \circ FS : \tau'' \to \tau'!E_1 \cup E_2}$$

where $OF \neq (\texttt{return}\ \bullet;)$

$$[\text{TF-}\textsc{StackClosed}]\frac{\Delta; context(H, VS) \vdash CF : \tau!E_1 \quad H, VS \vdash FS : \tau \to \tau'!E_2}{\Delta, H, VS \vdash CF \circ FS : \tau'' \to \tau'!E_1 \cup E_2}$$

where $CF \neq (\texttt{return}\ e;)$, $CF \neq (\{\})$, $CF \neq s_1 \ldots s_n \wedge n > 1$ and $CF \neq C\ var$

$$[\text{TF-}\textsc{StackEmpty}]\frac{}{\Delta, H, (BS \circ [\,]) \circ [\,] \vdash [\,] : \tau \to \tau!\emptyset}$$

A.2.6 Configuration typing
By pulling together all the previous typing rules we can give the definitition for a well typed configuration. It must have a well typed heap, variable stack, term and frame stack.

$$[\text{TF-Config}]\frac{\Delta \vdash H\ \mathtt{ok} \quad \Delta, H \vdash VS\ \mathtt{ok} \quad \Delta, H, VS \vdash CF \circ FS : \mathtt{void} \to \tau!E}{\Delta \vdash (H, VS, CF, FS) : \tau!E}$$

B Proofs

B.1 Progress Lemma

Proposition 3.1
If (H, VS, F, FS) *is not terminal and* $(H, VS, F, FS) : \tau!E$ *then*
$\exists H', VS', F', FS'.(H, VS, F, FS) \xrightarrow{E'} (H', VS', F', FS')$ *and* $E' \leq E$.

Proof. By induction on the typing of the configuration. By considering all the ways a configuration can be typed we can show all well typed configurations can reduce.

$$\frac{\dfrac{(\text{B.1})}{\Delta \vdash H\ \mathtt{ok}} \quad \dfrac{(\text{B.2})}{\Delta, H \vdash VS\ \mathtt{ok}} \quad \dfrac{(\text{B.3})}{\Delta, H, VS \vdash F \circ FS : \mathtt{void} \to \tau!E}}{\Delta \vdash (H, VS, F, FS) : \tau!E}$$

We will assume (B.1), (B.2) and (B.3). We can proceed by case analysis of F. We only need consider F to be a closed frame. For each typing rule we must provide reduction rules for each possible term. Here we just present a few example cases:

Case: $F = (\mathtt{return}\ e;)$ As it is well typed we know $VS = MS \circ VS'$. This has two possible cases of reducing.
Case: $e = v$ this can reduce by [E-Return].
Case: $e \neq v$ this can reduce by [EC-Return].

Case: $F = var$ We know that var must be in $context(H, VS)$, and as var can not be in H it must come from VS and more precisely from MS where $VS = MS \circ VS'$, hence it can reduce by [E-VarAccess].

Case: $F = e.f$ Typing this will introduce the effect $R(r)$ where the field f is in the region r.
This can be broken into two cases.
Case: $e = v$ If $v = \mathtt{null}$ then this can reduce by [E-NullField], otherwise $v = o$ as it is well typed we know that o has a field called f and hence can reduce by [E-FieldAccess]. [E-FieldAccess] has the effect $R(r)$, which we have in the typing judgement.
Case: $e \neq v$ This can reduce by [EC-FieldAccess].

Case: $F = e'.m(e_1, \ldots, e_n)$ If $e' \neq v$ then [EC-Method1] will apply. If any of $e_1, \ldots, e_n$ are not values then [EC-Method2] will be applied. Otherwise we have the case where $F = v'.m(v_1, \ldots, v_n)$ in which case [E-Method] applies if $v = o$ or [E-NullMethod] if $v = \mathtt{null}$.

The remaining cases all follow similar lines of reasoning. □

B.2 Covariant subtyping of stack

Lemma 3.2 *$\forall H, VS, \tau_1, \tau_2, \tau_3, E$. if $H, VS \vdash FS : \tau_1 \to \tau_2!E_1$ and $\tau_3 \prec \tau_1$ then $\exists \tau_4, E_2. H, VS \vdash FS : \tau_3 \to \tau_4!E_2$, $\tau_4 \prec \tau_2$ and $E_2 \leq E_1$.*

Proof. By induction on the size of FS.

Base Case:($FS = [] $)
This can only be typed by [TF-StackEmpty]. This rule is covariant, as the only constraint is that the argument and the result types are the same. The empty stack never has any effects.

Inductive Step
Show covariant subtyping holds for $F \circ FS$ by assuming it holds for FS.

If F is closed then this is trivial, because all closed frames ignore their argument. Hence their typing and effects can not depend on the argument's type.

Next we must consider the open frames. First let us consider `return` $\bullet$; as this affects the typing environment. This is covariantly typed if the remainder of the frame stack is covariantly typed. Hence this is true by inductive hypothesis. For the remainder of the cases it suffices to prove.

$$\overbrace{\Delta; \Gamma, \bullet : \tau \vdash OF : \tau_1!E_1}^{(B.4)} \wedge \overbrace{\tau_2 \prec \tau}^{(B.5)}$$
$$\Rightarrow \exists \tau_3, E_2. \underbrace{\Delta; \Gamma, \bullet : \tau_2 \vdash OF : \tau_3!E_2}_{(B.6)} \wedge \underbrace{\tau_3 \prec \tau_1}_{(B.7)} \wedge \underbrace{E_2 \prec E_1}_{(B.8)}$$

We present a few of the more interesting cases.

Case: $OF =$ `if` $(\bullet$ `==` $e)\{\overline{s_1}\}$ `else` $\{\overline{s_2}\}$; As e, $\overline{s_1}$ and $\overline{s_2}$ do not contain any occurrences of $\bullet$, they will be well typed for any valid $\bullet : \tau_2$. Hence this frame is covariant.

Case: $OF = \bullet.f$ From assumptions we know $\Delta; \Gamma, \bullet : \tau \vdash \bullet.f : C_2!R(r)$. As $\tau_2 \prec \tau$ and field types and regions can not be overidden, we know $\Delta_f(\tau_2)(f) = (C_2, r)$, which lets us prove $\Delta; \Gamma, \bullet : \tau_2 \vdash \bullet.f : C_2!R(r)$ as required.

Case: $OF = var$ `=` $\bullet$; We know from the assumptions that $\Delta; \Gamma, \bullet : \tau \vdash var$ `=` $\bullet$;$: $ `void`$!\emptyset$, which gives us $\Delta; \Gamma, \bullet : \tau \vdash var : C!\emptyset$ and $\tau \prec C$. As the sub-typing relation is transitive and $\tau_2 \prec \tau$. This is clearly well typed for $\bullet : \tau_2$. The result type and effects are the same.

Case: $OF = (C)\bullet$ The result type of this frame does not depend on the argument type. The three possible typing rules for this case combined to only require $\bullet$ is typeable. Hence this case is trivial.

□

B.3 Type preservation lemma

Proposition 3.3
If $\Delta \vdash (H, VS, F, FS) : \tau!E_1$ *and* $(H, VS, F, FS) \rightarrow (H', VS', F', FS')$ *then* $\exists \tau', E_2.\Delta \vdash (H', VS', F', FS') : \tau'!E_2$ *where* $\tau' \prec \tau$ *and* $E_2 \leq E_1$.

Proof. By induction on the $\rightarrow$ relation. By considering all possible reductions we can show that the program will always reduce to a valid configuration, and that the configuration will be a sub-type of the previous configuration. We will present a few of the more interesting cases here. The rest can be found in the technical report [1].

Case: [E-Return]

$$\begin{array}{lrl} \text{Assume} & \Delta \vdash (H, MS \circ VS, \texttt{return}\ v;, FS) : \tau!E & \text{(B.9)} \\ \text{Prove} & \Delta \vdash (H, VS, v, FS) : \tau!E & \text{(B.10)} \end{array}$$

From the definition of Local State Typing §A.2.4 we can see that $\Delta, H \vdash MS \circ VS\ \texttt{ok} \Rightarrow \Delta, H \vdash VS\ \texttt{ok}$. We know a value's typing is not affected by the variable scope, so $\Delta; context(H, MS \circ VS) \vdash v : C\ !\ \emptyset \Rightarrow \Delta; context(H, VS) \vdash v : C\ !\ \emptyset$. These two implications allow us to prove (B.10) from (B.9).

Case: [E-VarWrite]

$$\begin{array}{lrl} \text{Assume} & \Delta \vdash (H, MS \circ VS, var = v;, FS) : \tau!E & \text{(B.11)} \\ & update(MS, (var \mapsto v)) \downarrow & \text{(B.12)} \\ \text{Prove} & \Delta \vdash (H, MS', ;, FS) : \tau'!E & \text{(B.13)} \end{array}$$

where $MS' = update(MS, (var \mapsto v))$.

We know that the only change between MS' and MS is that one of the values has changed. The typing information is identical, hence $\Delta, H, MS \circ VS \vdash FS : \texttt{void} \rightarrow C \Rightarrow \Delta, H, MS' \circ VS \vdash FS : \texttt{void} \rightarrow C$. From the typing of (B.11) we know $\Delta; \Gamma \vdash var : C\ !\ \emptyset$ and $\Delta; \Gamma \vdash v : C'\ !\ \emptyset$ where $\Gamma = context(H, MS \circ VS)$ and $C' \prec C$. This allows us to deduce $\Delta, H \vdash MS' \circ VS\ \texttt{ok}$ from $\Delta, H \vdash MS \circ VS\ \texttt{ok}$. The rest of case follows trivially.

Case: [E-New]

$$\begin{array}{lrl} \text{Assume} & (H, VS, \texttt{new}\ C(v_1, \ldots, v_n), FS) : \tau!E & \text{(B.14)} \\ \text{Prove} & (H', MS \circ VS, \texttt{super}(\overline{e}); \overline{s}, (\texttt{return}\ o;) \circ FS) : \tau!E' & \text{(B.15)} \\ & E' \leq E & \text{(B.16)} \end{array}$$

where

$$o \notin dom(H) \tag{B.17}$$
$$\mathbb{F} = \{f \mapsto null | (f \in dom(\Delta_f(C))\} \tag{B.18}$$
$$\Delta_C(C) = [C_1, \ldots, C_n]!E_2 \tag{B.19}$$
$$cnbody(C) = [var_1, \ldots, var_n], \texttt{super}(\overline{e}); \overline{s} \tag{B.20}$$
$$MS = \{\texttt{this} \mapsto (o, C), var_1 \mapsto (v_1, C_1), \ldots, var_n \mapsto (v_n, C_n)\} \circ [] \tag{B.21}$$
$$H' = H[o \mapsto (C, \mathbb{F})] \tag{B.22}$$

We know from the typing of `new` that $\Delta; \Gamma \vdash v_i : C_i'$ and $C_i' \prec C_i$ for all i where $1 \leq i \leq n$. This allows us to deduce MS is a valid method scope, and hence $\Delta, H \vdash MS \circ VS$ ok. We know the extended heap H' is valid, because the object added is valid, and the rest is unchanged. We must show $\Delta, H, VS \vdash FS : \tau \to \tau' \Rightarrow \Delta, H', VS \vdash FS : \tau \to \tau'$, this is done by a trivial induction on the length of FS. Finally we must show that $\texttt{super}(\overline{e}); \overline{s}$ is well-typed when added to the stack. From the definition of MS, and the constructor being well-typed we know that $\Delta; context(H, MS \circ VS) \vdash \texttt{super}(\overline{e}); \overline{s} : \texttt{void}$. We must induct over the length of $\overline{s}$ to show it is well-typed wrt the frame stack sequencing rule.

This reduction reduces the possible effects as $E = E_1 \cup E_2$ and $E' = E_1 \cup E_2'$ where E_1 is the effects of FS, E_2 is the annotated effects of the constructor, and E_2' is the actual effects of the constructor. As the constructor is well typed we know $E_2' \leq E_2$. This proves (B.16) and completes this case. □

B.4 Inference algorithm produces sound annotations

Theorem 4.4 If $\Delta \vdash p : R_d$, $\vdash \Delta : R_s$ and θ satisfies $R_d \cup R_s$ then $\theta(\Delta) \vdash p$ ok.

Proof. If $\Delta \vdash p : R_d$, $\vdash \Delta : R_s$ and θ satisfies $R_s \cup R_d$ then $\theta(\Delta) \vdash p$ ok. This requires that we show that

$$\Delta \vdash mbody(C, m) : R_d'$$

then we have

$$\theta(\Delta) \vdash mbody(C, m) \text{ ok}$$

where $R_d' \subseteq R_d$, and a similar fact for constructors. This follows directly from applying the Lemma 4.3 to the body of the method. □