Technical Report

UCAM-CL-TR-563
ISSN 1476-2986

Number 563

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

MJ: An imperative core calculus for Java and Java with effects

G.M. Bierman, M.J. Parkinson, A.M. Pitts

April 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

MJ: An imperative core calculus for Java and Java with effects

G.M. Bierman M.J. Parkinson A.M. Pitts
University of Cambridge Computer Laboratory,
J.J. Thomson Avenue, Cambridge. CB3 0FD. UK.
{gmb,mjp41,amp12}@cl.cam.ac.uk

Abstract

In order to study rigorously object-oriented languages such as Java or C♯, a common practice is to define lightweight fragments, or calculi, which are sufficiently small to facilitate formal proofs of key properties. However many of the current proposals for calculi lack important language features. In this paper we propose Middleweight Java, MJ, as a contender for a minimal imperative core calculus for Java. Whilst compact, MJ models features such as object identity, field assignment, constructor methods and block structure. We define the syntax, type system and operational semantics of MJ, and give a proof of type safety. In order to demonstrate the usefulness of MJ to reason about operational features, we consider a recent proposal of Greenhouse and Boyland to extend Java with an effects system. This effects system is intended to delimit the scope of computational effects within a Java program. We define an extension of MJ with a similar effects system and instrument the operational semantics. We then 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 extended calculus, detail an algorithm for inferring effects information and give a proof of correctness.

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 [12], 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 in this paper we propose Middleweight Java, or MJ, as a contender for a 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 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 [8, 16], and were pursued by Talpin and Jouvelot [19, 20], amongst others. Interestingly most of these systems were defined for *functional* languages with simple forms of state.[1] Greenhouse and Boyland [10] 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 notions of class and 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 it correct. 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.

This paper is organised as follows. In §2 we give the syntax, type system and operational semantics of MJ. In §2.5 we outline a proof of type soundness; the details are given in Appendix 5. In §3 we define MJe, which is an extension of MJ with effects in the style of Greenhouse and Boyland. In §3.3 we outline a proof of correctness for the effects system of MJe; again the details are given in Appendix 5. In §3.4 we consider the problem of effects inference, define an inference algorithm and prove it correct. We survey some related work in §4, and give conclusions and indications of further work in §5.

2 MJ: An imperative core Java calculus

In this section we define Middleweight Java, MJ, our proposal for an imperative core calculus for Java. It is important to note that MJ is an entirely valid subset of Java, in that all MJ programs are literally executable Java programs. Clearly this is a attractive feature of the MJ design, although we found in a number of places that it complicated matters. An alternative would be to allow extra-language features; for example, Classic Java uses annotations and let bindings which are not valid Java syntax in the operational semantics [7].

In the rest of this section we present the syntax, type system and single-step operational semantics for MJ. We conclude the section with a proof of correctness of the type system.

2.1 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 sequence corresponds to the body of the `main` method in a Java program [9, §12.1.4].

For example, here are some typical MJ class definitions.

[1] Another approach for lazy functional programming with state is the use of *monads* [21]. Wadler has shown that effects systems can easily be adapted to monads [22].

Program

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

Class definition

cd	::=	`class` C `extends` C $\{fd_1 \ldots fd_k$ cnd $md_1 \ldots md_n\}$	

Field definition

fd	::=	$C\ f;$	

Constructor definition

cnd	::=	$C(C_1\ x_1, \ldots, C_j\ x_j)\{\texttt{super}(e_1, \ldots, e_k);\ s_1 \ldots s_n\}$	

Method definition

md	::=	$\tau\ m(C_1\ x_1, \ldots, C_n\ x_n)\{s_1 \ldots s_k\}$	

Return type

τ	::=	$C \mid$ `void`	

Expression

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

Promotable expression

pe	::=	$e.m(e_1, \ldots, e_k)$	Method invocation
	|	`new` $C(e_1, \ldots, e_k)$	Object creation

Statement

s	::=	;	No-op
	|	pe;	Promoted expression
	|	`if` $(e$ `==` $e)\{s_1 \ldots s_k\}$ `else` $\{s_{k+1} \ldots s_n\}$	Conditional
	|	$e.f$ `=` e;	Field assignment
	|	$C\ x$;	Local variable declaration
	|	x `=` e;	Variable assignment
	|	`return` e;	Return
	|	$\{s_1 \ldots s_n\}$	Block

Figure 1: Syntax for MJ programs

```
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;
    };}
```

This code defines two classes: `Cell` which is a subclass of the `Object` class, and `Recell` which is a subclass of `Cell`. The `Cell` class has one field, `contents`. The constructor method simply assigns the field to the value of the parameter. The `Cell` class defines a method `set`, which sets the field to a given value.

`Recell` objects inherit the `contents` field from their superclass, and also have another

field, `undo`. The constructor method first calls the superclass constructor method (which will assign the `contents` field) and then sets the `undo` field to the `null` object. The `Recell` class definition overrides the `set` method of its superclass.

As with Featherweight Java, we will insist on a certain amount of syntactic regularity in class definitions, 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 enforced by the type system); and (5) write out field accesses explicitly, even when the receiver is `this`.

In what follows, again for compactness, 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); (2) fields are not redefined in subclasses (we do not allow shadowing of fields); and (3) there are no cyclic class definitions (for example `A extends B` and `B extends A`). We do not formalise these straightforward conditions.

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 the class name, an ordered list of arguments and a body. There are a number of well-formedness conditions for a collection of class definitions which are formalised in the next section.

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 x 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 $e_0, \ldots, e_n$ (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 $\overline{x}$.

The reader will note 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 slightly awkward division is imposed upon us by our desire to make MJ a valid fragment of Java. Java [9, §14.8] only allows particular expression forms to be promoted to statements by postfixing a semicolon. This leads to some rather strange syntactic surprises: for example, `x++;` is a valid statement, but `(x++);` is not!

MJ includes the essential imperative features of Java. 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 valid MJ code fragment.

```
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.

2.2 Types

As with FJ, for simplicity, MJ does not have primitive types, sometimes called base types. Thus all well-typed expressions are of a class type, C. All well-typed statements are of type `void`, except for the statement form `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.

Expression types

C a valid class name, including a distinguished class `Object`

Statement types

τ ::= `void`$|C$

Method types

μ ::= $C_1, \ldots, C_n \to \tau$

Java, and MJ, class definitions contain both typing information and code. This typing information is extracted and used to typecheck the code. Thus before presenting the type-checking rules we need to specify how this typing information is induced by MJ code.

This typing information consists of two parts: The subclassing relation, and a class table (which stores the types associated with classes). First let us consider the subclassing relation. Recall that in MJ we restrict class declarations so that they must give the name of class that they are extending, even if this class is the `Object` class.

A well-formed program, p, then induces an immediate subclassing relation, which we write $\prec_1$. We define the **subclassing relation**, $\prec$, as the reflexive, transitive closure of this immediate subclassing relation. This can be defined formally as follows.

$$[\text{TR-Immediate}]\frac{\texttt{class } C_1 \texttt{ extends } C_2\{\ldots\} \in p}{C_1 \prec_1 C_2} \qquad [\text{TR-Transitive}]\frac{C_1 \prec C_2 \quad C_2 \prec C_3}{C_1 \prec C_3}$$

$$[\text{TR-Extends}]\frac{C_1 \prec_1 C_2}{C_1 \prec C_2} \qquad [\text{TR-Reflexive}]\frac{}{C \prec C}$$

A well-formed program p also induces a class table, Δ_p. As the program is fixed, we will simply write this as Δ. A class table, Δ, is actually 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. Thus

$\Delta_m(C)(m)$ is intended to denote the type of method m in class C. Δ_c is a partial map from a class name to the type 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. Thus $\Delta_f(C)(f)$ is intended to denote the type of f in class C. The details of how a well-formed program p induces a class table Δ are given below.

Method Type

$$\Delta_m(C)(m) \stackrel{\text{def}}{=} \begin{cases} \overline{C} \to C'' & \text{where } md_i = C''m(\overline{C}\,\overline{x})\{\ldots\}, \text{ for some } i, 1 \le i \le n \\ \Delta_m(C')(m) & \text{where } m \notin md_1 \ldots md_n \end{cases}$$

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

Constructor Type

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

where $\texttt{class}\ C\ \texttt{extends}\ C'\{\overline{fd}\ cnd\ \overline{md}\} \in p$
and $cnd = C(C_1\ x_1, \ldots, C_j\ x_j)\{\ldots\}$

Field Type

$$\Delta_f(C)(f) \stackrel{\text{def}}{=} \begin{cases} C'' & \text{where } fd_i = C''\ f, \text{ for some } i, 1 \le i \le k \text{ and } \Delta_f(C')(f) \uparrow \\ \Delta_f(C')(f) & \text{otherwise} \end{cases}$$

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

Up to now we have assumed that the class definitions are well-formed. Now let us define formally well-formedness of class definitions. First we will find it useful to define when a type (either a method type or statement type) is well-formed with respect to a class table. This is written as a judgement $\Delta \vdash \mu\ \mathsf{ok}$ which means that μ is a valid type given the class table Δ. We overload this judgement form for statement types. (We define $dom(\Delta)$ to be the domain of Δ_f, Δ_m or Δ_c, which are all equal.)

$$[\text{T-CType}]\ \frac{}{\Delta \vdash C\ \mathsf{ok}} \qquad \text{where } C \in dom(\Delta)$$

$$[\text{T-VType}]\ \frac{}{\Delta \vdash \texttt{void}\ \mathsf{ok}}$$

$$[\text{T-MType}]\ \frac{\Delta \vdash C_1\ \mathsf{ok} \quad \ldots \quad \Delta \vdash C_n\ \mathsf{ok} \quad \Delta \vdash \tau\ \mathsf{ok}}{\Delta \vdash C_1, \ldots, C_n \to \tau\ \mathsf{ok}}$$

Now we define formally the judgement for well-formed class tables, which is written $\vdash \Delta\ \mathsf{ok}$. This essentially checks two things: firstly that all types are valid given the classes defined in the class table, and secondly that if a method is overridden then it is so at the *same* type. The judgements are as follows.

$$[\text{T-FieldsOk}]\ \frac{\Delta \vdash \Delta_f(C)(f_1)\ \mathsf{ok} \quad \ldots \quad \Delta \vdash \Delta_f(C)(f_n)\ \mathsf{ok}}{\Delta \vdash \Delta_f(C)\ \mathsf{ok}} \quad \text{where } dom(\Delta_f(C)) = \{f_1, \ldots, f_n\}$$

$$[\text{T-ConsOK}]\ \frac{\Delta \vdash C_1\ \mathsf{ok} \quad \ldots \quad \Delta \vdash C_n\ \mathsf{ok}}{\Delta \vdash \Delta_c(C)\ \mathsf{ok}} \quad \text{where } \Delta_c(C) = C_1, \ldots, C_n$$

$$[\text{T-MethOk1}]\ \frac{\Delta \vdash \mu\ \mathsf{ok}}{\Delta \vdash C.m\ \mathsf{ok}} \quad \text{where } \Delta_m(C)(m) = \mu,\ C \prec_1 C',\ \Delta_m(C')(m) = \mu',\ \mu = \mu'$$

$$[\text{T-MethOk2}]\ \frac{\Delta \vdash \mu\ \mathsf{ok}}{\Delta \vdash C.m\ \mathsf{ok}} \quad \text{where } \Delta_m(C)(m) = \mu,\ C \prec_1 C',\ m \notin dom(\Delta_m(C'))$$

$$[\text{T-MethsOk}]\ \frac{\Delta \vdash C.m_1\ \mathsf{ok}\ \ldots\ \Delta \vdash C.m_n\ \mathsf{ok}}{\Delta \vdash \Delta_m(C)\ \mathsf{ok}} \quad \text{where } dom(\Delta_m(C)) = \{m_1, \ldots, m_n\}$$

$$[\text{T-ClassOK}]\ \frac{\Delta \vdash \Delta_f(C)\ \mathsf{ok} \quad \Delta \vdash \Delta_m(C)\ \mathsf{ok} \quad \Delta \vdash \Delta_c(C)\ \mathsf{ok}}{\vdash \Delta\ \mathsf{ok}} \quad \forall C \in dom(\Delta)$$

We can now define the typing rules for expressions and promotable expressions. Our treatment of casting is the same as in FJ—thus we include a case for 'stupid' casting, where the target class is completely unrelated to the subject class. This is needed to handle the case where an expression without stupid casts may reduce to one containing a stupid cast. Consider the following expression, taken from [12], where classes `A` and `B` are both defined to be subclasses of the `Object` class, but are otherwise unrelated.

```
(A)(Object)new B()
```

According to the Java Language Specification [9, §15.16], this is well-typed, but consider its operational behaviour: a `B` object is created and is dynamically upcast to `Object` (which has no dynamic effect). At this point we wish to downcast the `B` object to `A`—a 'stupid' cast! Thus if we wish to maintain a subject reduction theorem (that the result of a single step reduction of a well-typed program is also a well-typed program) we need to include the [TE-StupidCast] rule. For the same reason, we also require two rules for typing an `if` statement. Java [9, §15.21.2] enforces statically that when comparing objects, one object should be a subclass of the other. However this is not preserved by the dynamics. Consider again the unrelated classes `A` and `B`. The following code fragment is well-typed but at runtime will end up comparing objects of unrelated classes.

```
if ((Object)new A() == (Object)new D()) { ... } else {...};
```

The reader should note that a *valid* Java/MJ program is one that does not have occurrences of [TE-StupidCast] or [TS-StupidIf] in its typing derivation.

A typing environment, Γ, is a map from program variables to expression types. We write $\Gamma, x\colon C$ to denote the map Γ extended so that x maps to C. We write $\Delta \vdash \Gamma\ \mathsf{ok}$ to mean

that the types in the codomain of the typing environment are well-formed with respect to the class table, Δ.

A typing judgement for a given MJ expression, e, is written $\Delta; \Gamma \vdash e\colon C$ where Δ is a class table and Γ is a typing environment. These rules are given below and are fairly intuitive except perhaps [TE-Null], which allows the `null` value to have any valid type (see [9, §4.1]). One interesting fact about the evaluation of `null` is `(C)(Object)(B)`e will only reduce, without generating an exception, if e evaluates to `null`, assuming `B` is not a subclass of `C`.

$$[\text{TE-Var}]\ \frac{\Delta \vdash \Gamma\ \mathsf{ok} \qquad \vdash \Delta\ \mathsf{ok}}{\Delta; \Gamma, x\colon C \vdash x\colon C}$$

$$[\text{TE-Null}]\ \frac{\Delta \vdash C\ \ \mathsf{ok} \qquad \Delta \vdash \Gamma\ \mathsf{ok} \qquad \vdash \Delta\ \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{null}\colon C}$$

$$[\text{TE-FieldAccess}]\ \frac{\Delta; \Gamma \vdash e\colon C_2 \qquad \Delta_f(C_2)(f) = C_1}{\Delta; \Gamma \vdash e.f\colon C_1}$$

$$[\text{TE-UpCast}]\ \frac{\Delta; \Gamma \vdash e\colon C_2 \qquad C_2 \prec C_1 \qquad \Delta \vdash C_1}{\Delta; \Gamma \vdash (C_1)e\colon C_1}$$

$$[\text{TE-DownCast}]\ \frac{\Delta; \Gamma \vdash e\colon C_2 \qquad C_1 \prec C_2 \qquad \Delta \vdash C_1}{\Delta; \Gamma \vdash (C_1)e\colon C_1}$$

$$[\text{TE-StupidCast}]\ \frac{\Delta; \Gamma \vdash e\colon C_2 \qquad C_2 \not\prec C_1 \qquad C_1 \not\prec C_2 \qquad \Delta \vdash C_1}{\Delta; \Gamma \vdash (C_1)e\colon C_1}$$

A typing judgement for a given promotable expression, pe, is also written $\Delta; \Gamma \vdash pe\colon C$ where Δ is a class table and Γ is a typing environment. The rules for forming these judgements are as follows.

$$[\text{TE-Method}]\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e\colon C' \qquad \Delta; \Gamma \vdash e_1\colon C_1 \qquad \ldots \qquad \Delta; \Gamma \vdash e_n\colon C_n \\ \Delta_m(C')(m) = C'_1, \ldots, C'_n \to C \\ C_1 \prec C'_1 \ \ldots \ C_n \prec C'_n\end{array}}{\Delta; \Gamma \vdash e.m(e_1, \ldots, e_n)\colon C}$$

$$[\text{TE-New}]\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e_1\colon C'_1 \qquad \ldots \qquad \Delta; \Gamma \vdash e_n\colon C'_n \\ \Delta_c(C) = C_1, \ldots, C_n \\ C'_1 \prec C_1 \ \ldots \ C'_n \prec C_n\end{array}}{\Delta; \Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n)\colon C}$$

A typing judgement for a statement, s, is written $\Delta; \Gamma \vdash s\colon \tau$ where Δ is a class table and Γ is a typing environment. As we noted earlier, statements in Java can have a non-`void` type (see rule [TS-Return]). The rules for forming these typing judgements are as follows.

$$[\text{TS-NoOp}]\frac{\vdash \Delta\ \mathsf{ok} \quad \Delta \vdash \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash ;:\texttt{void}}$$

$$[\text{TS-PE}]\frac{\Delta;\Gamma \vdash pe:\tau}{\Delta;\Gamma \vdash pe;:\texttt{void}}$$

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

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

$$[\text{TS-FieldWrite}]\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1:C_1 \quad C_2 \prec C_3 \\ \Delta;\Gamma \vdash e_2:C_2 \quad \Delta_f(C_1)(f) = C_3\end{array}}{\Delta;\Gamma \vdash e_1.f = e_2;:\texttt{void}}$$

$$[\text{TS-VarWrite}]\frac{\Delta;\Gamma \vdash x:C \quad \Delta;\Gamma \vdash e:C' \quad C' \prec C \quad x \neq \texttt{this}}{\Delta;\Gamma \vdash x = e;:\texttt{void}}$$

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

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

Java allows variable declarations to occur at any point in a block. To handle this we introduce two typing rules for sequencing: one for the case where the first statement in the sequence is a variable declaration, and one for the other cases. An alternative approach would be to force each statement to return the new typing environment. We feel that our presentation is simpler. Typing judgements for a sequence of statements, $\overline{s}$, is written $\Delta;\Gamma \vdash \overline{s}:\tau$ where Δ is a class table and Γ is a typing environment. The rules for forming these judgements are given below.

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

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

Finally we define the typing of the `super` call in the constructor of a class. A call to the empty constructor in a class that directly extends `Object` is always valid, and otherwise it must be a valid call to the constructor of the parents class. The expressions evaluated before this call are not allowed to reference `this` [9, §8.8.5.1]. This is because it is not a valid object until the parents constructor has been called.

$$[\text{T-CObject}]\ \frac{}{\Delta;\Gamma,\texttt{this}:C\vdash \texttt{super}():\texttt{void}} \qquad C\prec_1 \texttt{Object}$$

$$[\text{T-CSuper}]\ \frac{\Delta;\Gamma'\vdash e_1:C_1' \quad \ldots \quad \Delta;\Gamma'\vdash e_n:C_n'}{\Delta;\Gamma\vdash \texttt{super}(e_1,\ldots,e_n);:\texttt{void}}$$

where $\Gamma(\texttt{this})=C$, $\Gamma=\Gamma'\uplus\{\texttt{this}:C\}$ and $\Delta_c(C')=C_1,\ldots,C_n$ and $C\prec_1 C'$ and $C_1'\prec C_1\ldots C_n'\prec C_n$

Before we give the typing rules involving methods we first define some useful auxiliary functions for accessing the bodies of constructors and methods.

Method Body

$$mbody(C,m)\stackrel{\text{def}}{=}\begin{cases}(\overline{x},\overline{s}) & \text{where } md_i=C''m(\overline{C}\,\overline{x})\{\overline{s}\}\\ mbody(C,m') & \text{where } m\notin md_1\ldots md_n\end{cases}$$

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

Constructor Body

$$cnbody(C)\stackrel{\text{def}}{=}(\overline{x},\overline{s})$$

where $\texttt{class}\ C\ \texttt{extends}\ C'\{\overline{fd}\ C(\overline{C''}\ \overline{x})\{\overline{s}\}\ \overline{md}\}\in p$

We can now formalise the notion of a program being well-typed with respect to its class table. This is denoted by the judgement $\Delta\vdash p\ \ \mathsf{ok}$. Informally this involves checking that each method body and constructor body is well-typed and that the type deduces matches that contained in Δ.

We introduce two new judgement forms: $\Delta\vdash C\ \mathsf{mok}$ denotes that the methods of class C are well-typed, and $\Delta\vdash C\ \mathsf{cok}$ denotes that the constructor method of class C is well-typed. The rules for forming these judgements are given below.

$$[\text{T-MDefn}]\ \frac{\Delta;\Gamma\vdash\overline{s}:\tau}{\Delta\vdash mbody(C,m)\ \mathsf{ok}}$$

where $\Gamma=\{\texttt{this}:C,\overline{x}:\overline{C}\}$ and $mbody(C,m)=(\overline{x},\overline{s})$ and $\Delta_m(C)(m)=(\overline{C}\to\tau)$

$$[\text{T-MBodys}]\ \frac{\Delta\vdash mbody(C,m_1)\ \mathsf{ok} \quad \ldots \quad \Delta\vdash mbody(C,m_n)\ \mathsf{ok}}{\Delta\vdash C\ \mathsf{mok}}$$

where $dom(\Delta_m(C))=\{m_1,\ldots,m_n\}$

$$[\text{T-CDefn}]\ \frac{\Delta;\Gamma\vdash\texttt{super}(\overline{e});:\texttt{void} \quad \Delta;\Gamma\vdash\overline{s}:\texttt{void}}{\Delta\vdash C\ \mathsf{cok}}$$

where $\Gamma=\texttt{this}:C,\overline{x}:\overline{C}$ and $\Delta_c(C)=\overline{C}$ and $cnbody(C)=(\overline{x},\texttt{super}(\overline{e});\ \overline{s})$

$$[\text{T-ProgDef}]\ \frac{\begin{array}{cc}\Delta\vdash C_1\ \mathsf{cok} & \Delta\vdash C_n\ \mathsf{cok}\\ \Delta\vdash C_1\ \mathsf{mok} \quad \ldots & \Delta\vdash C_n\ \mathsf{mok}\end{array}}{\Delta\vdash p\ \mathsf{ok}}$$

where $dom(\Delta)=\{C_1,\ldots,C_n\}$

2.3 Operational Semantics

We define the operational semantics of MJ in terms of transitions between *configurations*, rather than using Felleisen-style evaluation contexts. As the reader will see this style of semantics has the advantage that the transition rules are defined by case analysis rather than by induction, which simplifies some proofs.

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

1. **Heap**: A finite partial function that maps oids to heap objects, where a heap object is a pair of a class name and a field function. A field function is a partial finite map from field names to values.

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

3. **Term**: The closed frame to be evaluated.

4. **Frame stack**: This is essentially the program context in which the term is currently being evaluated.

This can be defined formally as follows.

Configuration

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

Frame Stack

$$FS ::= F \circ FS | []$$

Frame

$$F ::= CF | OF$$

Closed frame

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

Open frame

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

Values

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

Variable Stack

$$VS ::= MS \circ VS | []$$

Method Scope

$$MS ::= BS \circ MS | []$$

Block Scope

$$BS ::= \text{is a finite partial function from variables to pairs of expression types and values}$$

Heap

$$H ::= \text{is a finite partial function from oids to heap objects}$$

Heap Objects

$$\begin{aligned} ho &::= (C, \mathbb{F}) \\ \mathbb{F} &::= \text{is a finite partial function from field names to values} \end{aligned}$$

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).

The structure for representing the variable scopes may seem complicated but they are required to correctly model the block structure scoping of Java. The following example highlights this. The left hand side gives the source code and the right the contents of the variable stack.

```
1  B m(A a) {
2      B r;
3      if(this.x == arg) {
4          r = arg;
5      } else {
6          A t;
7          t = this.getVal();
8          r = this.create(t,t);
9      }
10     return r;
11 }
```

$\longleftarrow VS$
$\longleftarrow (\{a \mapsto (A, v_1)\} \circ []) \circ VS$
$\longleftarrow (\{r \mapsto (B, \texttt{null}), a \mapsto (A, v_1)\} \circ []) \circ VS$
$\longleftarrow (\{\} \circ \{r \mapsto (B, \texttt{null}), a \mapsto (A, v_1)\} \circ []) \circ VS$
$\longleftarrow (\{t \mapsto (A, \texttt{null})\} \circ \{r \mapsto (B, \texttt{null}), a \mapsto (A, v_1)\} \circ []) \circ VS$
$\longleftarrow (\{t \mapsto (A, v_3\} \circ \{r \mapsto (B, \texttt{null}), a \mapsto (A, v_2)\} \circ []) \circ VS$
$\longleftarrow (\{t \mapsto (A, v_3\} \circ \{r \mapsto (B, v_4), a \mapsto (A, v_2)\} \circ []) \circ VS$
$\longleftarrow (\{r \mapsto (B, v_4), a \mapsto (A, v_2)\} \circ []) \circ VS$
$\longleftarrow VS$

Before calling this method, let us assume we have a variable scope, VS. A method call should not affect any variables in the current scopes, so we create a new method scope, $\{a \mapsto (A, v)\} \circ []$, on entry to the method. This scope consists of a single block scope that points the argument a at the value v with a type annotation of A. On line 2 the block scope is extended to contain the new variable r. On line 5 we assumed that $\texttt{this.x} \neq \texttt{arg}$ and enter into a new block. This has the effect to adding a new empty block scope, $\{\}$, into the current method scope. On line 6 this new scope is extended to contain the variable t. Notice how it is added to the current outermost scope, as was the variable r. Updates can however occur to block scopes anywhere in the current method scope. This can be seen on line 8 where r is updated from the outer scope. On line 9 the block scope is disposed of, and hence t can not be accessed by the statement on line 10.

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

$$eval((BS \circ MS), x) \stackrel{\text{def}}{=} \begin{cases} BS(x) & \text{where } x \in dom(BS) \\ eval(MS, x) & \text{otherwise} \end{cases}$$

$$update((BS \circ MS), (x \mapsto v)) \stackrel{\text{def}}{=} \begin{cases} BS[x \mapsto (v, C)] \circ MS & \text{where } BS(x) = (v', C) \\ BS \circ update(MS, (x \mapsto v)) & \text{otherwise} \end{cases}$$

2.3.1 Reductions

This section defines the transition rules that correspond to meaningful computation steps. In spite of the computational complexity of MJ, there are only seventeen rules, one for each syntactic constructor.

We begin by giving the transition rules for accessing, assigning values to, and declaring variables. Notice that 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

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

block scope.)

[E-VarAccess]	$(H, MS \circ VS, x, FS) \rightarrow (H, MS \circ VS, v, FS)$	where $eval(MS, x) = (v, C)$
[E-VarWrite]	$(H, MS \circ VS, x = v;, FS)$ $\rightarrow (H, (update(MS, (x \mapsto v))) \circ VS, ;, FS)$	where $eval(MS, x) \downarrow$
[E-VarIntro]	$(H, (BS \circ MS) \circ VS, C\ x;, FS) \rightarrow$ $(H, (BS' \circ MS) \circ VS, ;, FS)$	where $x \notin dom(BS \circ MS)$ and $BS' = BS[x \mapsto (\texttt{null}, C)]$

Now we consider the rules for constructing and removing scopes. The first rule [E-BlockIntro] introduces a new block scope, and leaves a 'marker' 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.

[E-BlockIntro]	$(H, MS \circ VS, \{\overline{s}\}, FS) \rightarrow (H, (\{\} \circ MS) \circ VS, \overline{s}, (\{\ \}) \circ FS))$
[E-BlockElim]	$(H, (BS \circ MS) \circ VS, \{\ \}, FS) \rightarrow (H, MS \circ VS, ;, FS)$
[E-Return]	$(H, MS \circ VS, \texttt{return}\ v;, FS) \rightarrow (H,\ VS, v, FS)$

Next we give the transition rules for the conditional expression. One should note that the resulting term of the transition is a block.

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

Finally we consider the rules dealing with objects. First let us define the transition rule dealing with field access and assignment, as they are reasonably straightforward.

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

Now we consider the rules dealing with casting of objects. Rule [E-Cast] simply ensures that the cast is valid (if it is not, the program should enter an error state—this is covered in §2.3.2). Rule [E-NullCast] simply ignores any cast of a `null` object.

[E-Cast]	$(H,\ VS, ((C_2)o), FS) \rightarrow (H,\ VS, o, FS)$	where $H(o) = (C_1, \mathbb{F})$ and $C_1 \prec C_2$
[E-NullCast]	$(H,\ VS, ((C)\texttt{null}), FS) \rightarrow (H,\ VS, \texttt{null}, FS)$	

Let us consider the transition rule involving the creation of objects. The [E-New] rule creates a fresh oid, o, and places on the heap the heap object consisting of the class C and assigns all the fields to `null`. As we are executing a new 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.

[E-New]	$(H,\, VS, \texttt{new}\ C(\overline{v}), FS) \rightarrow$ $(H[o \mapsto (C, \mathbb{F})], (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$	where $cnbody(C) = (\overline{x}, \overline{s})$ $\Delta_c(C) = \overline{C},\ o \notin dom(H)$ $\mathbb{F} = \{f \mapsto \texttt{null}\}\ \forall f \in fields(C)$ $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

Next we consider the transition rule for the `super` statement, which occurs inside constructor methods.

[E-Super]	$(H, MS \circ VS, \texttt{super}(\overline{v}), FS) \rightarrow$ $(H, (BS' \circ []) \circ (MS \circ VS), \overline{s}, (\texttt{return}\ o;) \circ FS)$	where $MS(\texttt{this}) = (o, C), C \prec_1 C'$ $BS' = \{\texttt{this} \mapsto (o, C'), \overline{x} \mapsto (\overline{v}, \overline{C})\}$ $\Delta_c(C) = \overline{C},\ cnbody(C') = (\overline{x}, \overline{s})$

Next we can give the transition rule for method invocation. Invocation is relatively straightforward: note how a new method scope is created, consisting of just the bindings for the method parameters and the `this` identifier. We require two rules as a method returning a `void` type requires an addition to the stack to clear the new method scope once the method has completed. Recall that in [E-Method] rule, the last statement in the sequence $\overline{s}$ will be a `return` if the method is well typed.

[E-Method]	$(H,\, VS, o.m(\overline{v}), FS) \rightarrow$ $(H, (BS \circ []) \circ VS, \overline{s}, FS)$	where $mbody(C, m) = (\overline{x}, \overline{s})$ $H(o) = (C, \mathbb{F}),\ \Delta_m(C)(m) = \overline{C} \rightarrow C'$ $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$
[E-MethodVoid]	$(H,\, VS, o.m(\overline{v}), FS) \rightarrow$ $(H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$	where $H(o) = (C, \mathbb{F})$ $\Delta_m(C)(m) = \overline{C} \rightarrow \texttt{void}$ $mbody(C, m) = (\overline{x}, \overline{s})$ $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

Finally we have two reductions rules for fully reduced terms. The first rule deals with completed statements, and the second for evaluated expressions.

[E-Skip]	$(H,\, VS, ;, F \circ FS) \rightarrow (H,\, VS, F, FS)$
[E-Sub]	$(H,\, VS, v, F \circ FS) \rightarrow (H,\, VS, F[v] \circ FS)$

To assist the reader, all the reduction rules are repeated in full in Figure 2. There are

a number of other reduction rules, that simply decompose terms. These rules essentially embody the order of evaluation, and are given in Figure 3.

2.3.2 Error states

A number of expressions will lead us 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, in MJ, are NullPointerException, written here **NPE**, and ClassCastException, **CCE**.

[E-NullField]	$(H, VS, \texttt{null}.f, FS) \rightarrow \mathbf{NPE}$	
[E-NullWrite]	$(H, VS, \texttt{null}.f = v, FS) \rightarrow \mathbf{NPE}$	
[E-NullMethod]	$(H, VS, \texttt{null}.m(v_1, \ldots, v_n), FS) \rightarrow \mathbf{NPE}$	
[E-InvCast]	$(H, VS, (C)o, FS) \rightarrow \mathbf{CCE}$	where $H(o) = (C', \mathbb{F})$ and $C' \not< C$

Definition 2.1 (Terminal configuration) *A configuration is said to be* **terminal** *if it is a valid error (***NPE** *or* **CCE***) or it is of the form* $(H, VS, v, [])$.

2.3.3 Example Execution

To help the reader understand our operational semantics, in this section we will consider a simple code fragment and see how the operational semantics captures its dynamic behaviour.

Consider the following MJ code whose effect is to swap the contents of two variables `var1` and `var2`, using a temporary variable `temp`.

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

We consider its execution in a configuration where MS maps `var1` to a value, say $v1$, and `var2` to a value, say $v2$.

$$
\begin{array}{ll}
 & (H, MS \circ VS, \texttt{if (var1 == var2)\{;\} else \{...\}}, FS) \\
\rightarrow & (H, MS \circ VS, \texttt{var1}, \texttt{(if (}\bullet\texttt{ == var2)\{;\} else \{...\})} \circ FS) \\
\rightarrow & (H, MS \circ VS, v1, \texttt{(if (}\bullet\texttt{ == var2)\{;\} else \{...\})} \circ FS) \\
\rightarrow & (H, MS \circ VS, \texttt{(if (}v1\texttt{ == var2)\{;\} else \{...\})}, FS) \\
\rightarrow & (H, MS \circ VS, \texttt{var2}, \texttt{(if (}v1\texttt{ == }\bullet\texttt{)\{;\} else \{...\})} \circ FS) \\
\rightarrow & (H, MS \circ VS, v2, \texttt{(if (}v1\texttt{ == }\bullet\texttt{)\{;\} else \{...\})} \circ FS) \\
\rightarrow & (H, MS \circ VS, \texttt{(if (}v1\texttt{ == }v2\texttt{)\{;\} else \{...\})}, FS)
\end{array}
$$

At this point there are two possibilities: we will consider the case where $v1 \neq v2$.

[E-VarAccess]	$(H, MS \circ VS, x, FS) \rightarrow (H, MS \circ VS, v, FS)$	where $eval(MS, x) = (v, C)$
[E-VarWrite]	$(H, MS \circ VS, x = v;, FS)$ $\rightarrow (H, (update(MS, (x \mapsto v))) \circ VS, ;, FS)$	where $eval(MS, x) \downarrow$
[E-VarIntro]	$(H, (BS \circ MS) \circ VS, C\ x;, FS) \rightarrow$ $(H, (BS' \circ MS) \circ VS, ;, FS)$	where $x \notin dom(BS \circ MS)$ and $BS' = BS[x \mapsto (\texttt{null}, C)]$
[E-BlockIntro]	$(H, MS \circ VS, \{\overline{s}\}, FS)$ $\rightarrow (H, (\{\} \circ MS) \circ VS, \overline{s}, (\{\ \}) \circ FS))$	
[E-BlockElim]	$(H, (BS \circ MS) \circ VS, \{\ \}, FS) \rightarrow (H, MS \circ VS, ;, FS)$	
[E-Return]	$(H, MS \circ VS, \texttt{return}\ v;, FS) \rightarrow (H, VS, v, FS)$	
[E-If]	$(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS)$ $\rightarrow (H, VS, \{\overline{s_1}\}, FS)$	if $v_1 = v_2$
[E-If2]	$(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS)$ $\rightarrow (H, VS, \{\overline{s_2}\}, FS)$	if $v_1 \neq v_2$
[E-FieldAccess]	$(H, VS, o.f, FS) \rightarrow (H, VS, v, FS)$	where $o \in dom(H)$ and $H(o) = (C, \mathbb{F})$ and $\mathbb{F}(f) = v$ and $\Delta_f(C)(f) = C'$
[E-FieldWrite]	$(H, VS, o.f = v;, FS) \rightarrow (H', VS, ;, FS)$	where $H(o) = (C, \mathbb{F})$, $f \in dom(\mathbb{F})$, $\Delta_f(C)(f) = C'$ $H' = H[o \mapsto (C, \mathbb{F}')]$
[E-Cast]	$(H, VS, ((C_2)o), FS) \rightarrow (H, VS, o, FS)$	where $H(o) = (C_1, \mathbb{F})$ and $C_1 \prec C_2$
[E-NullCast]	$(H, VS, ((C)\texttt{null}), FS) \rightarrow (H, VS, \texttt{null}, FS)$	
[E-New]	$(H, VS, \texttt{new}\ C(\overline{v}), FS) \rightarrow$ $(H[o \mapsto (C, \mathbb{F})], (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$	where $cnbody(C) = (\overline{x}, \overline{s})$ $\Delta_c(C) = \overline{C}$, $o \notin dom(H)$ $\mathbb{F} = \{f \mapsto \texttt{null}\}\ \forall f \in fields(C)$ $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$
[E-Super]	$(H, MS \circ VS, \texttt{super}(\overline{v}), FS) \rightarrow$ $(H, (BS' \circ []) \circ (MS \circ VS), \overline{s}, (\texttt{return}\ o;) \circ FS)$	where $MS(\texttt{this}) = (o, C), C \prec_1 C'$ $BS' = \{\texttt{this} \mapsto (o, C'), \overline{x} \mapsto (\overline{v}, \overline{C})\}$ $\Delta_c(C) = \overline{C}$, $cnbody(C') = (\overline{x}, \overline{s})$
[E-Method]	$(H, VS, o.m(\overline{v}), FS) \rightarrow$ $(H, (BS \circ []) \circ VS, \overline{s}, FS)$	where $mbody(C, m) = (\overline{x}, \overline{s})$ $H(o) = (C, \mathbb{F})$, $\Delta_m(C)(m) = \overline{C} \rightarrow C'$ $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$
[E-MethodVoid]	$(H, VS, o.m(\overline{v}), FS) \rightarrow$ $(H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$	where $H(o) = (C, \mathbb{F})$ $\Delta_m(C)(m) = \overline{C} \rightarrow \texttt{void}$ $mbody(C, m) = (\overline{x}, \overline{s})$ $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$
[E-Skip]	$(H, VS, ;, F \circ FS) \rightarrow (H, VS, F, FS)$	
[E-Sub]	$(H, VS, v, F \circ FS) \rightarrow (H, VS, F[v] \circ FS)$	

Figure 2: MJ reduction rules

[EC-Seq] $(H, VS, s_1\ s_2 \ldots s_n, FS) \rightarrow (H, VS, s_1, (s_2 \ldots s_n) \circ FS)$

[EC-Return] $(H, MS \circ VS, \texttt{return}\ e;, FS)$
$\rightarrow (H, MS \circ VS, e, (\texttt{return}\ \bullet;) \circ FS)$

[EC-ExpState] $(H, VS, e';, FS) \rightarrow (H, VS, e', FS)$

[EC-If1] $(H, VS, \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};, FS)$
$\rightarrow (H, VS, e_1, (\texttt{if}\ (\bullet == e_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\};) \circ FS)$

[EC-If2] $(H, VS, \texttt{if}\ (v_1 == e_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\};, FS)$
$\rightarrow (H, VS, e_2, (\texttt{if}\ (v_1 == \bullet)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\};) \circ FS)$

[EC-FieldAccess] $(H, VS, e.f, FS) \rightarrow (H, VS, e, (\bullet.f) \circ FS)$

[EC-Cast] $(H, VS, (C)e, FS) \rightarrow (H, VS, e, ((C)\bullet) \circ FS)$

[EC-FieldWrite1] $(H, VS, e_1.f = e_2;, FS) \rightarrow (H, VS, e_1, (\bullet.f = e_2;) \circ FS)$

[EC-FieldWrite2] $(H, VS, v_1.f = e_2;, FS) \rightarrow (H, VS, e_2, (v_1.f = \bullet;) \circ FS)$

[EC-VarWrite] $(H, VS, x = e;, FS) \rightarrow (H, VS, e, (x = \bullet;) \circ FS)$

[EC-New] $(H, VS, \texttt{new}\ C(v_1, \ldots, v_{i-1}, e_i, \ldots e_n), FS)$
$\rightarrow (H, VS, e_i, (\texttt{new}\ C(v_1, \ldots, v_{i-1}, \bullet, \ldots e_n)) \circ FS)$

[EC-Super] $(H, VS, \texttt{super}(v_1, \ldots, v_{i-1}, e_i, \ldots e_n), FS)$
$\rightarrow (H, VS, e_i, (\texttt{super}(v_1, \ldots, v_{i-1}, \bullet, \ldots e_n)) \circ FS)$

[EC-Method1] $(H, VS, e.m(e_1, \ldots, e_n), FS) \rightarrow (H, e, (\bullet.m(e_1, \ldots, e_n)) \circ FS)$

[EC-Method2] $(H, VS, v.m(v_1, \ldots, v_{i-1}, e_i, \ldots e_n), FS)$
$\rightarrow (H, VS, e_i, (v.m(v_1, \ldots, v_{i-1}, \bullet, \ldots e_n)) \circ FS)$

Figure 3: MJ decomposition reduction rules

$\rightarrow \quad (H, (\{\}) \circ MS \circ VS, o\ \texttt{temp};\ldots, (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\}) \circ MS \circ VS, o\ \texttt{temp};, (\texttt{temp = var1};\ldots) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (\texttt{null}, o\}) \circ MS \circ VS, ;, (\texttt{temp = var1};\ldots) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (\texttt{null}, o)\}) \circ MS \circ VS, (\texttt{temp = var1};\ldots), (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (\texttt{null}, o)\}) \circ MS \circ VS, \texttt{temp = var1};, (\texttt{var1 = var2};\ldots) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (\texttt{null}, o)\}) \circ MS \circ VS, \texttt{var1}, (\texttt{temp} = \bullet;) \circ (\texttt{var1 = var2};\ldots) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (\texttt{null}, o)\}) \circ MS \circ VS, v1, (\texttt{temp} = \bullet;) \circ (\texttt{var1 = var2};\ldots) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (\texttt{null}, o)\}) \circ MS \circ VS, \texttt{temp} = v1;, (\texttt{var1 = var2};\ldots) \circ (\{\}) \circ FS)$

At this point we update the variable stack; note how the update does not change the type.

$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS \circ VS, ;, (\texttt{var1 = var2};\ldots) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS \circ VS, \texttt{var1 = var2};\ldots, (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS \circ VS, \texttt{var1 = var2};, (\texttt{var2 = temp};) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS \circ VS, \texttt{var2}, \texttt{var1} = \bullet; \circ (\texttt{var2 = temp};) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS \circ VS, v2, \texttt{var1} = \bullet; \circ (\texttt{var2 = temp};) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS \circ VS, \texttt{var1} = v2;, (\texttt{var2 = temp};) \circ (\{\}) \circ FS)$

Let MS' be a variable scope which is the same as MS except that `var1` is mapped to $v2$ instead of $v1$.

$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS' \circ VS, ;, (\texttt{var2 = temp};) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS' \circ VS, \texttt{var2 = temp};, (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS' \circ VS, \texttt{temp}, (\texttt{var2} = \bullet;) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS' \circ VS, v1, (\texttt{var2} = \bullet;) \circ (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS' \circ VS, \texttt{var2} = v1;, (\{\}) \circ FS)$

Let MS'' be a variable scope which is the same as MS' except that `var2` is mapped to $v1$. Also let $FS = F \circ FS'$.

$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS'' \circ VS, ;, (\{\}) \circ FS)$
$\rightarrow \quad (H, (\{\texttt{temp} \mapsto (v1, o)\}) \circ MS'' \circ VS, \{\}, FS)$
$\rightarrow \quad (H, MS'' \circ VS, F, FS')$

At this point the execution of the `if` statement has been completed and its temporary variable, `temp` has been removed from the scope. The variable stack has had the values of `var1` and `var2` correctly swapped.

2.4 Well-Typed Configuration

To prove type soundness for MJ, we need to extend our typing rules to configurations. We write $\Delta \vdash (H, VS, CF, FS) : \tau$ to mean (H, VS, CF, FS) is well-typed with respect to Δ and will result in a value of type τ (or a valid error state). We break this into three properties: $\Delta \vdash H\ \mathsf{ok}$, $\Delta, H \vdash VS\ \mathsf{ok}$ and $\Delta, H, VS \vdash CF \circ FS : \texttt{void} \rightarrow \tau$.

The first, $\Delta \vdash H\ \mathsf{ok}$, ensures that every field points to a valid object or `null`, and all the types mentioned in the heap are in Δ.

$$[\textsc{ObjectTyped}]\frac{H(o) = (C, \mathbb{F}) \qquad C \prec \tau \qquad \Delta \vdash C\ \mathsf{ok}}{\Delta, H \vdash o : \tau} \qquad [\textsc{NullTyped}]\frac{\Delta \vdash C\ \mathsf{ok}}{\Delta, H \vdash \texttt{null} : C}$$

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

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

The second, $\Delta, H \vdash VS\ \mathsf{ok}$, constrains every variable to be either a valid object identifier, or `null`.

$$[\textsc{VarBS}]\frac{\Delta, H \vdash v_1 : C_1 \quad \ldots \quad \Delta, H \vdash v_n : C_n}{\Delta, H \vdash BS\ \mathsf{ok}} \quad \text{where } BS = \{x_1 \mapsto (v_1, C_1), \ldots, (x_n \mapsto (v_n, C_n)\}$$

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

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

The final property, $\Delta, H, VS \vdash FS : \tau \rightarrow \tau'$, types each frame in the context formed from the heap and variable stack. This requires us to define a collapsing of the context to form a typing environment. We must extend the typing environment to map, in addition to variables, object identifiers, o, and holes, $\bullet$, to values. The collapsing function is defined as follows.

$$\begin{array}{rcl} context(\{\}, [\,]) & \stackrel{\text{def}}{=} & \{\} \\ context(\{\}, (\{\} \circ MS) \circ VS) & \stackrel{\text{def}}{=} & context(\{\}, MS \circ VS) \\ context(\{\}, (BS[x \mapsto v, C] \circ MS) \circ VS') & \stackrel{\text{def}}{=} & context(\{\}, (BS \circ MS) \circ VS) \uplus \{x \mapsto C\} \\ \multicolumn{3}{l}{\quad \text{where } x \notin dom(BS) \text{ and } x \notin dom(context(\{\}, (BS \circ MS) \circ VS)} \\ context(H[o \mapsto C, \mathbb{F}], VS) & \stackrel{\text{def}}{=} & context(H, VS) \uplus \{o \mapsto C\} \\ \multicolumn{3}{l}{\quad \text{where } o \notin dom(H)} \end{array}$$

The syntax of expressions in framestacks is extended to contain both object identifiers and holes. Hence we require two additional expression typing rules.

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

We can know define frame stack typing as follows. We have the obvious typing for an empty stack. We require special typing rules for the frames that alter variable scoping. We also require a rule for unrolling sequences because the sequence can contain items to alter scoping. We then require two rules for typing the rest of the frames; one for frames that require an argument and one for frames that do not.

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

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

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

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

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

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

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

$$[\text{TF-StackOpen}]\ \frac{\Delta; context(H, VS), \bullet : C \vdash OF : \tau \qquad H, VS \vdash FS : \tau \to \tau'}{\Delta, H, VS \vdash OF \circ FS : C \to \tau'}$$

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

$$[\text{TF-StackClosed}]\ \frac{\Delta; context(H, VS) \vdash CF : \tau \qquad H, VS \vdash FS : \tau \to \tau'}{\Delta, H, VS \vdash CF \circ FS : \tau'' \to \tau'}$$

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

2.5 Type Soundness

Our main technical contribution in this section is a proof of type soundness of the MJ type system. In order to prove correctness we first prove two useful propositions in the style of Wright and Felleisen[23]. The first proposition states that any well typed non-terminal configuration can make a reduction step.

Proposition 2.2 (Progress) *If* (H, VS, F, FS) *is not terminal and* $\Delta \vdash (H, VS, F, FS) : \tau$ *then*
$\exists H', VS', F', FS'.(H, VS, F, FS) \to (H', VS', F', FS')$.

Proof. By case analysis of F. Details are given in Appendix A.1. ∎

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

Lemma 2.3 (Covariant subtyping of frame stack) $\forall H, VS, \tau_1, \tau_2, \tau_3$. *if* $\Delta, H, VS \vdash FS : \tau_1 \to \tau_2$ *and* $\tau_3 \prec \tau_1$ *then* $\exists \tau_4.H, VS \vdash FS : \tau_3 \to \tau_4$ *and* $\tau_4 \prec \tau_2$.

Proof. By induction on the length of FS. Note we only have to consider open frames as all closed frames ignore their argument. Appendix A.4 contains full 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. (This is sometimes referred to as a subject reduction theorem.)

Proposition 2.4 (Type Preservation) *If* $(H, VS, F, FS) : \tau$ *and* $(H, VS, F, FS) \rightarrow (H', VS', F', FS')$ *then* $\exists \tau'.(H', VS', F', FS') : \tau'$ *where* $\tau' \prec \tau$.

Proof. By case analysis on the reduction step. Lemma 3.2 is needed for the reduction rules that generate subtypes. Appendix A.5 contains full details of this proof. ∎

We can now combine the two propositions to prove the type soundness of the MJ type system.

Theorem 2.5 (Type Soundness) *If* $(H, VS, F, FS) : \tau$ *and* $(H, VS, F, FS) \rightarrow^* (H', VS', F', FS')$ *where* (H', VS', F', FS') *is terminal then either* $(H', VS', F', FS') : \tau'$ *where* $\tau' \prec \tau$ *or the configuration is of the form* **NPE** *or* **CCE**.

3 MJe: A core Java calculus with effects

The usefulness of MJ as an imperative core calculus for Java hinges on whether it can be used as a basis for investigations into various operational properties of Java. In this section, we give the details of one such investigation: the formal development and analysis of an effects systems for Java, following closely the suggestions of Boyland and Greenhouse [10]. Thus we will describe a simple extension of MJ with an effects system, calling the resulting language MJe.

In the rest of this section we begin by giving an overview of the key features of the effects system of Boyland and Greenhouse. We then define formally MJe, giving the required extension to the MJ type system, and instrumenting the operational semantics. We conclude by proving the correctness of our effects system. This question was left open by Greenhouse and Boyland.

3.1 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 optimisations of code. Most effects systems have been defined for *functional* languages with simple state. 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 also introduce two extra notions that we do

not address in this paper 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 comments for annotations in Extended Static Checking [4].) 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 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;
    }}
```

Consider the class `Point1D`. This defines a field `x` which is in region `Position`, and a method `scale` that clearly has both a read and a write effect 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

In this section we give the extensions to the definitions of MJ to yield MJe.

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, executable 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

$$md \quad ::= \quad \tau\ m\ \texttt{/*}\ eff\ \texttt{*/}(C_1\ x_1, \ldots, C_n\ x_n)\ \{s_1 \ldots s_k\};$$

Constructor definition

$$cnd \quad ::= \quad C\ \texttt{/*}\ eff\ \texttt{*/}(C_1\ x_1, \ldots, C_j\ x_j)\ \{\texttt{super}(e_1, \ldots, e_k);\ s_1 \ldots s_n\};$$

Effect annotation

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

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

Effects. An effect is either empty, $\emptyset$, (written `nothing` in MJe source code), the union of two effects, written e.g. $E_1 \cup E_2$, a read effect, $R(r)$, or a write effect, $W(r)$.

Effect

$E \quad ::= \quad \emptyset | W(r) | R(r) | E \cup E$

Equality of effects is modulo the assumption that $\cup$ is commutative, associative and idempotent, and has $\emptyset$ as the 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 overridden 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 [10] (and subsequently [3]) solve this by adding $R(r) \leq W(r)$ for all regions r to the subeffecting relation. To keep the subeffecting relation simple, especially when considering correctness and effect inference, we 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. As with MJ, we must initially describe functions for extracting typing information from the program before we give the typing rules.

The class table, Δ, must now take account of the effect and region information contained in the MJe annotations. Δ_m and Δ_c are extended to return the effects of the methods and constructors, and Δ_f is extended to provide the region for each field. We also define a function, *effect*, that translates an MJe effect annotation into its associated effect. Because of the difficulties mentioned above, this function translates a write effect annotation, `writes r` to the effect $R(r) \cup W(r)$, for some region r.

Method Type

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

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

Constructor Type

$$\Delta_c(C) \stackrel{\text{def}}{=} \overline{C}!\mathit{effect}(\mathit{eff})$$

where $\texttt{class}\ C\ \texttt{extends}\ C'\{\overline{fd}\ cnd\ \overline{md}\} \in p$
and $cnd = C/*\mathit{eff}*/(\overline{C}\ \overline{x})\{\overline{s}\}$

Field Type

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

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

Effect annotation translation

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

When a method is overridden, to preserve subtyping the new method must not modify any additional regions. Hence we must add a subeffecting constraint to the judgement for forming well-formed methods, as follows.

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

$$[\text{T-MethOk2}]\ \frac{\Delta \vdash \mu\ \mathsf{ok}}{\Delta \vdash C.m\ \mathsf{ok}} \quad \text{where } \Delta_m(C)(m) = \mu!E,\ C \prec_1 C' \text{ and } m \notin \mathit{dom}(\Delta_m(C'))$$

The typing rules must also be extended to carry the effect information. Typing judgements are now of the form $\Delta; \Gamma \vdash e\colon \tau!E$, where Δ, Γ and τ are as before, and E is the effect. Only four rules actually introduce effects and are given below. The [TS-FieldWrite] rule handles method subtyping by introducing both read and write effects. [TE-Method] and [TE-New] are both extended to lookup the effects of the code from the annotation.

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

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

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

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

The other typing rules require trivial modifications. The axioms introduce the empty effect and the other rules simply combine the effects from their premises.

The final extension to the typing rules is to check that the effects of method and constructor bodies are valid with respect to the annotations.

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

$$[\text{T-CSuper}]\ \frac{\Delta,\Gamma \vdash e_1 : C'_1!E_1 \quad \ldots \quad \Delta,\Gamma \vdash e_n : C'_n!E_n}{\Delta,\Gamma \vdash \texttt{super}(e_1,\ldots,e_n); : \texttt{void}!E \cup E_1 \cup \ldots \cup E_n} \quad \begin{array}{l}\text{where } \Gamma(\texttt{this}) = C \text{ and } C \prec_1 C'\\ \text{and } \Delta_c(C') = C_1,\ldots,C_n!E\\ \text{and } C'_1 \prec C_1,\ldots,C'_n \prec C_n\end{array}$$

$$[\text{T-CObject}]\ \frac{C \prec_1 \texttt{Object}}{\Gamma, \texttt{this} : C \vdash \texttt{super}(); : \texttt{void}!\emptyset}$$

$$[\text{T-CDefn}]\ \frac{\Delta,\Gamma \vdash \texttt{super}(\overline{e}); : \texttt{void}!E_1 \quad \Delta,\Gamma \vdash \overline{s} : \texttt{void}!E_2}{\Delta \vdash C\ \texttt{cok}} \quad \begin{array}{l}\text{where } \Gamma = \texttt{this} : C, \overline{x : C}\\ \text{and } \Delta_c(C) = \overline{C}!E \text{ and}\\ \mathit{cnbody}(C) = (\overline{x}, \texttt{super}(\overline{e});\ \overline{s})\\ \text{and } E_1 \cup E_2 \leq E\end{array}$$

Now let us consider the dynamics of MJe. We shall instrument the MJ operational semantics to trace effectful computations. (This is used to demonstrate consistency between the semantics and effects). A single reduction step is now written $(H, VS, CF, FS) \xrightarrow{E} (H', VS', CF', FS')$, where E is a trace of the effects of the step. The only significant instrumentation is in the [E-FieldAccess] and [E-FieldWrite] rules, which are annotated with $R(r)$ and $W(r)$ respectively. The rest of the rules have the $\emptyset$ effect annotation. We define the transitive and reflexive closure of this relation, $\xrightarrow{E}{}^*$, as the obvious union of the annotations.

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

3.3 Correctness

Our main technical contribution in this section is a proof of correctness of the effects system of MJe. As we have mentioned earlier, this was not addressed by Greenhouse and Boyland. Our choice of instrumenting the MJ operational semantics means that the correctness proof is essentially an adaptation of the type soundness proof for the MJ type system. Thus in order to prove correctness we first prove two useful propositions.

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) *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 case analysis of the frame F. Further details are given in Appendix A.6. ■

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* $H, VS \vdash FS : \tau_1 \rightarrow \tau_2!E_1$ *and* $\tau_3 \prec \tau_1$ *then* $\exists \tau_4, E_2.H, VS \vdash FS : \tau_3 \rightarrow \tau_4!E_2$, $\tau_4 \prec \tau_2$ *and* $E_2 \leq E_1$.

Proof. By induction on the length of the frame stack FS. Note we only have to consider open frames as all closed frames ignore their argument. Further details are given in Appendix A.7. ■

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* $(H, VS, F, FS) \rightarrow (H', VS', F', FS')$ *then* $\exists \tau', E_2.(H', VS', F', FS') : \tau'!E_2$ *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. Full details are given in Appendix A.8. ■

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

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

3.4 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 formally 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 show that this is possible, and give an outline of an algorithm which is then 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 further extend the types 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.

Effect			**Constraint Sets**		
E	$::=$	$\emptyset \mid W(r) \mid R(r) \mid X \mid E \cup E$	R	$::=$	$\{E_1 \leq X_1, \ldots, E_n \leq X_n\}$

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.

For the inference algorithm we do not have annotations on methods to describe their effects. Instead we generate a *fresh* effect variable to represent their effects. We write $\theta(\Delta)$ to denote the substitution θ on the effects variables contained in Δ; we suppress the rather routine details.

$$\Delta_m(C)(m) = \begin{cases} \overline{C} \to C''!X & \text{where } md_i = C''m(\overline{C}\vec{x})\{\ldots\} \\ \Delta_m(C')(m) & \text{where } m \notin md_1 \ldots md_n \end{cases}$$

where `class` C `extends` $C'\{\overline{fd}\ cnd\ md_1 \ldots md_n\} \in p$
and X is a fresh effect variable.

We must extend the definitions of both a well-formed class table, $\vdash \Delta$ ok, and well-formed programs, $\Delta \vdash p$ ok to generate constraints. We will first give the new definition of a well-formed class table. Instead of checking the subtyping requirements on each methods' effects, we generate a constraint for each check. The rules then union together the constraints.

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

$$[\text{T-MethOk2}]\ \frac{\Delta \vdash \mu\ \mathsf{ok}}{\Delta \vdash_i C.m : \emptyset} \quad \text{where} \quad \begin{array}{l}\Delta_m(C)(m) = \mu!X \\ C \prec_1 C' \\ m \notin dom(\Delta_m(C'))'\end{array}$$

$$[\text{T-MethsOk}]\ \frac{\Delta \vdash_i C_1.m_{1,1} : R_{1,1} \quad \ldots \quad \Delta \vdash_i C_n.m_{n,n_n} : R_{n,n_n}}{\Delta \vdash_i \Delta_m : R_{1,1} \cup \ldots \cup R_{n,n_n}} \quad \begin{array}{l}\text{where } dom(\Delta_m) = \{C_1, \ldots, C_n\}\ , \\ dom(\Delta_m(C_1)) = \{m_{1,1}, \ldots, m_{1,n_1}\}, \\ \ldots, \\ dom(\Delta_m(C_n)) = \{m_{n,1}, \ldots, m_{n,n_n}\}\end{array}$$

$$[\text{T-Defn}_i]\ \frac{\Delta \vdash_i \Delta_m : R \quad \vdash \Delta_c\ \mathsf{ok} \quad \vdash \Delta_f\ \mathsf{ok}}{\vdash_i \Delta : R}$$

Next we will give the extension to the definition of a well-formed program. Here we must produce constraints that the effect variable associated to a method or constructor has at least the effects of the body.

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

$$[\text{T-CDefn}]\ \frac{\Delta, \Gamma \vdash \texttt{super}(\overline{e});: \texttt{void}!E_1 \quad \Delta, \Gamma \vdash \overline{s} : \texttt{void}!E_2}{\Delta \vdash_i C\ \mathsf{cok} : \{E_1 \cup E_2 \leq X\}} \quad \begin{array}{l}\text{where } \Gamma = \texttt{this} : C', \overline{x} : \overline{C} \\ \text{and } cnbody(C) = (\overline{x}, \texttt{super}(\overline{e});\ \overline{s}) \\ \text{and } \Delta_c(C') = \overline{C}!X\end{array}$$

We use the following three rules to generate the constraint set for the whole program.

$$[\text{T-Methods}]\ \frac{\Delta \vdash_i mbody(C, m_1) : R_1 \ \ldots\ \Delta \vdash_i mbody(C, m_n) : R_n}{\Delta \vdash_i C\ \mathsf{mok} : R_1 \cup \ldots \cup R_n} \quad \begin{array}{l}\text{where} \\ dom(\Delta_m(C)) = \{m_1, \ldots, m_n\}\end{array}$$

$$[\text{T-ClasDef}]\ \frac{\begin{array}{ccc}\Delta \vdash_i C\ \mathsf{mok} : R_1 & & \Delta \vdash_i C\ \mathsf{mok} : R_n \\ \Delta \vdash_i C\ \mathsf{cok} : R'_1 & \ldots & \Delta \vdash_i C\ \mathsf{cok} : R'_n\end{array}}{\Delta \vdash_i p : R_1 \cup R'_1 \cup \ldots \cup R_n \cup R'_n}$$

We have given ways of generating constraints based on the class hierarchy, $\vdash_i \Delta : R_1$, and also constraints from the implementations of methods and constructors, $\Delta \vdash p : R_2$. 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 *minimum* solution is what is needed (in this case it is $\{X \mapsto W(r) \cup R(r)\}$).

Lemma 3.5 (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 [19]. They also give an algorithm for finding the minimum solution of a set of constraints. ■

Our main result in this section is that our inference algorithm is correct, in that it generates valid effect annotations. First we prove a couple of useful lemmas about effect substitution. The first states that if a substitution satisfies the constraints generated by a class table, then applying it to the class table produces a well-formed class table; which is obvious by definition.

Lemma 3.6 (Substitutions satisfy subtyping) *If* $\vdash \Delta : R_s$ *and* θ *satisfies* R_s *then* $\vdash \theta(\Delta)$ *ok.*

Proof. Straight from definitions. ■

The next lemma states that effect substitutions preserve typing judgements.

Lemma 3.7 (Effect substitution preserves typing) *If* $\Delta; \Gamma \vdash t : C!E$, $\vdash \Delta : R_s$ *and* θ *satisfies* R_s *then* $\theta(\Delta); \Gamma \vdash t : C!\theta(E)$

Proof. By induction on the typing relation. This uses the fact that substitution is $\cup$-continuous, to prove the rules that compose effects from their premises. It requires the previous lemma to prove the axioms. ■

From these two lemmas we can prove the correctness of our effect inference algorithm.

Theorem 3.8 (Inference algorithm produces sound annotations) *If* $\Delta \vdash p : R_d$, $\vdash \Delta : R_s$ *and* θ *satisfies* $R_s \cup R_d$ *then* $\theta(\Delta) \vdash p$ *ok.*

Proof. This follows from the definitions and repeated use of Lemma 3.7. See Appendix §A.9 for more details. ■

4 Related work

There have been many works on formalising subsets of Java. Our work is closely related to, and motivated by, Featherweight Java [12]. We have upheld the same philosophy, by keeping MJ a valid subset of Java. However FJ lacks many key features we wish to model. It has no concept of state or object identity, which we feel essential for developing usable specification logics. Our work has extended FJ to contain what we feel are the important imperative features of Java.

Another related calculus is Classic Java [7], which embodies many of the language features of MJ. However, Classic Java is not a valid subset of Java, as it uses let-binding to model both sequencing and locally scoped variables. Hence several features of Java, such as its block structured state and unusual syntactic distinction on promotable expressions, are not modelled directly. However their motivation is different to ours: they are interested in extending Java with mixins, rather than reasoning about features of Java.

Eisenbach, Drossopoulou, et al. have developed type soundness proofs for various subsets of Java [5, 6]. In fact they consider a larger subset of Java than MJ as they model exceptions and arrays, however they do not model block structured scoping. Our aims were to provide an imperative core subset of Java rather than to prove soundness of a large fragment.

There has been some other related work on effects systems. The use of regions by Greenhouse and Boyland is similar to Leino's use of *data groups* [15]. Data groups are also an abstract means for encapsulating fields. The key difference is that regions own a field uniquely, where as a field can belong to many data groups. Leino uses data groups to discuss only writes/updates of fields. Clarke and Drossopoulou [3] have also defined an effects system for a Java-like language. Their system uses ownership types rather than regions to delimit the scope of computational effects. A more detailed comparison is left to future work.

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 proposed by Greenhouse and Boyland [10]. We formally defined the effects system and an instrumented operational semantics, and 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.

There are surprising problems when considering subject reduction for Java, primarily concerning the use of substitution. This was discussed in detail recently on the types forum [11]. Consider the following method declaration:

```
Object m (boolean b, Object a1, Object a2) {
    return (b ? a1 : a2);
  }
```

Imagine unfolding a method call `m(v1,v1)` using substitution. Typechecking the resulting statement `return (b ? v1 : v2);` runs into difficulty because we have lost the 'static' types of `v1` and `v2` (`Object`), and it may be the case that their 'dynamic' types are not "cast

convertible". In MJ we do not model parameter passing by substitution, and so in fact this problem does not arise (method invocations create new scopes that contain the 'static' typing information). Of course we still have two "Stupid" type rules: one for casting and one for comparison.

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.

Recent work on formalising the various generic extensions of Java, like Generic Java (GJ) [2], have been based on FJ [12]. Indeed this was part of the motivation for the design of FJ. Recently Alan Jeffrey has discovered a problem with the type inference used in GJ [13]. His counterexample exploits the manipulation of state to generate a runtime exception. Thus it appears that adopting a functional core calculus of Java to study generics is an oversimplification. We intend to use MJ as a basis for studying the proposals for generic extensions to both Java [2] and C$^\sharp$ [14]

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 [17, 18].

Acknowledgements

Portions of this work were presented at the Workshop for Object-Oriented Developments [1]. We are grateful to the referees for their comments on this work.

We are grateful for discussions with Andrew Kennedy, Alan Lawrence and Martin Odersky. This work was supported by EPSRC (Parkinson) and EU AppSem II (Bierman).

References

[1] G.M. Bierman and M.J. Parkinson. Effects and effect inference for a core Java calculus. In *Proceedings of WOOD*, volume 82 of *ENTCS*, 2003.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98*, October 1998.

[3] D. Clarke and S. Drossopolou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of OOPSLA*, November 2002.

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

[5] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999.

[6] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. URL `http://citeseer.nj.nec.com/article/drossopoulou00java.html`.

[7] 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.

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

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

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

[11] H. Hosoya, B. Pierce, and D. Turner. Subject reduction fails in Java. Note sent to the types mailing list, June 1998.
URL `http://www.cis.upenn.edu/~bcpierce/types/archives/1997-98/msg00400.html`.

[12] 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.

[13] A. Jeffrey. Generic Java type inference is unsound. Note sent to the types mailing list, December 2001.
URL `http://www.cis.upenn.edu/~bcpierce/types/archives/current/msg00849.html`.

[14] A.J. Kennedy and D. Syme. The design and implementation of generics for the .NET common language runtime. In *Proceedings of PLDI*, june 2001.

[15] K.R.M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, 1998.

[16] J.M. Lucassen. *Types and effects, towards the integration of functional and imperative programming*. PhD thesis, MIT Laboratory for Computer Science, 1987.

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

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

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

[20] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

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

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

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

A Proofs

A.1 Progress Lemma

Proposition 2.2 (Progress) *If* (H, VS, CF, FS) *is not terminal and* $\Delta \vdash (H, VS, CF, FS) : \tau$ *then* $\exists H', VS', CF', FS'.(H, VS, CF, FS) \rightarrow (H', VS', CF', FS')$.

Proof. By case analysis of CF. By considering all well typed configurations, we can show how each term can reduce.

$$\frac{\dfrac{(1)}{\Delta \vdash H\ \texttt{ok}} \quad \dfrac{(2)}{\Delta, H \vdash VS\ \texttt{ok}} \quad \dfrac{(3)}{\Delta, H, VS \vdash CF \circ FS : \texttt{void} \rightarrow \tau}}{\Delta \vdash (H, VS, CF, FS) : \tau}$$

We will assume the above typing judgement, and for each term we provide a corresponding reduction rule.

Case: $CF = (\texttt{return}\ e;)$ 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: $CF = (\{\})$ As this is well typed, we know $VS = (BS \circ MS) \circ VS'$ and hence this can reduce by [E-Block].

Case: $CF = C\ x;$ As it is well typed, we know $VS = (BS \circ MS) \circ VS'$ and also $x \notin context(VS)$. From the definition of context we can see that this gives $x \notin dom(BS \circ MS)$ and hence it can reduce by [E-VarIntro].

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

Case: $CF = o$.

- **Case:** $FS \neq []$ reduces by [E-Sub].
- **Case:** $FS = []$ This is a terminal framestack: $(H, VS, o, [])$.

Case: $CF = \texttt{null}$.

- **Case:** $FS \neq []$ reduces by [E-Sub].
- **Case:** $FS = []$ This is a terminal framestack: $(H, VS, \texttt{null}, [])$.

Case: $CF = e.f$ This can be broken into three cases.

- **Case:** $e = \texttt{null}$ reduces by [E-NullField].
- **Case:** $e = o$ reduces by [E-FieldAccess].
- **Case:** $e \neq v$ reduces by [EC-FieldAccess].

Case: $CF = e'.m(e_1, \ldots, e_n)$ 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 $CF = v'.m(v_1, \ldots, v_n)$ in which case [E-Method] applies if $v' = o$ or [E-NullMethod] if $v' = \texttt{null}$.

Case: $CF = \texttt{new}\ C(e_1, \ldots, e_n)$ reduces by [EC-New] when $e_i \neq v$ or [E-New] otherwise.

Case: $CF = (C)e$.

Case: $e \neq v$ reduces by [EC-Cast].

Case: $e = o$ reduces by either [E-Cast] or [E-InvCast] depending on the type of o.

Case: $e = \texttt{null}$ this can reduce by [E-NullCast].

Case: $CF = =;$.

Case: $FS \neq []$ reduces by [E-Skip].

Case: $FS = []$ This is a terminal state.

Case: $CF = s_1 \ldots s_n$ reduces by [EC-Seq].

Case: $CF = \texttt{if}\ (e_1\ \texttt{==}\ e_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}$.

Case: $e_1 \neq v_1$ reduces by [EC-If1]

Case: $e_2 \neq v_2 \wedge e_1 = v_1$ reduces by [EC-If2]

Case: $e_1 = v_1 \wedge e_2 = v_2$ can reduce by either depending on the test [E-If1] and [E-If2].

Case: $CF = x\ \texttt{=}\ e$.

Case: $e = v$ this can reduce using [E-VarWrite] if x is in the environment. The typing rule tells us that this must be true as $context(H, VS) \vdash x : C$

Case: $e \neq v$ this can reduce by [EC-VarWrite].

Case: $CF = e.f\ \texttt{=}\ e'$.

Case: $e \neq v$ reduces using [EC-FieldWrite1].

Case: $e = v \wedge e' \neq v'$ reduces using [EC-FieldWrite2].

Case: $e = o \wedge e' = v'$ reduces using [E-FieldWrite].

Case: $e = \texttt{null} \wedge e' = v'$ reduces using [E-NullWrite].

Case: $CF = e;$ reduces by [EC-ExpState].

Case: $CF = \texttt{super}(e_1, \ldots, e_n)$ If all the expressions are values then this can reduce by [E-Super], otherwise it can reduce by [EC-Super].

■

A.2 Frame stack sequence typing

We require two lemmas for dealing with typing sequences when they are added to the frame stack.

Lemma A.1 (Block) $\Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau \wedge \Delta; context(H, (BS \circ MS) \circ VS) \vdash \overline{s} : \texttt{void} \Rightarrow \Delta, H, (BS \circ MS) \circ VS \vdash \overline{s} \circ \{\} \circ FS : \texttt{void} \to \tau$

Lemma A.2 (Return) $\Delta, H, VS \vdash FS : \tau \to \tau' \wedge \Delta; context(H, MS \circ VS) \vdash s_1 \ldots s_n : \tau \Rightarrow \Delta, H, MS \circ VS \vdash s_1 \circ \ldots \circ s_n \circ FS : \texttt{void} \to \tau$ *where* s_n *is of the form* `return` e;

The two lemmas have very similar proofs. We will present the proof to the first lemma here.

Proof. We prove this by induction on the size of $\overline{s}$.

Base Case:

$$\Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau \wedge \Delta; context(H, (BS \circ MS) \circ VS) \vdash s : \texttt{void} \Rightarrow \Delta, H, (BS \circ MS) \circ VS \vdash s \circ \{\} \circ FS : \texttt{void} \to \tau \quad (4)$$

This is trivial as the proof of $\Delta, H, (BS \circ MS) \circ VS \vdash s \circ \{\} \circ FS : \texttt{void} \to \tau$ is given by the two assumptions.

Inductive Case: Assume:

$$\forall BS. \Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau \wedge \Delta; context(H, (BS \circ MS) \circ VS) \vdash s_2 \ldots s_n \Rightarrow H, (BS \circ MS) \circ VS \vdash s_2 \ldots s_n \circ \{\} \circ FS : \texttt{void} \to \tau \quad (5)$$

$$\Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau \quad (6)$$

$$\Delta; context(H, (BS' \circ MS) \circ VS) \vdash s_1 \ldots s_n \quad (7)$$

Prove:

$$\Delta, H, (BS' \circ MS) \circ VS \vdash s_1 \ldots s_n \circ \{\} \circ FS : \texttt{void} \to \tau \quad (8)$$

We proceed by a case analysis of s_1.

Case: $s_1 = Cx$; From (7) we can deduce $\Delta; context(H, (BS'[x \to (C, \texttt{null})] \circ MS) \circ VS) \vdash s_2 \ldots s_n : \texttt{void}$. Using this and specialising the the inductive hypothesis (5) allows us to deduce $\Delta, H, (BS[x \to (C, \texttt{null})] \circ MS) \circ VS \vdash s_2 \ldots s_n \circ \{\} \circ FS : \texttt{void} \to \tau$. This is exactly what we require to prove (8), which completes this case.

Case: $s_1 \neq Cx$; From (7) we can deduce

$$\Delta; context(H, (BS' \circ MS) \circ VS) \vdash s_2 \ldots s_n : \texttt{void} \quad (9)$$

$$\Delta; context(H, (BS' \circ MS) \circ VS) \vdash s_1 : \texttt{void} \quad (10)$$

Specialising the inductive hypothesis and using (9) gives us $\Delta, H, (BS \circ MS) \circ VS \vdash s_2 \ldots s_n \circ \{\} \circ FS : \texttt{void} \to \tau$, which combined with (10) gives (8), and completes this case.

■

A.3 Heap Extension preserves typing

Lemma A.3 *If* $\Delta, H, VS \vdash FS : \tau \to \tau'$, $o \notin dom(H)$ *and* $\Delta \vdash C$ *then* $\Delta, H[o \mapsto (C, \ldots)], VS \vdash FS : \tau \to \tau'$

Proof. by induction on the length of FS. All the rules for typing the frame stack follow by induction, except for [TF-OpenFrame], [TF-ClosedFrame] and [TF-Method2]. To prove these it suffices to prove

$$\Delta; \Gamma \vdash F : \tau \wedge \Delta \vdash C \Rightarrow \Delta; \Gamma, o : C \vdash F : \tau$$

where $o \notin dom(\Gamma)$.

We can see $\Delta \vdash \Gamma\ \mathsf{ok} \wedge \Delta \vdash C \Rightarrow \Delta \vdash \Gamma, o : C\ \mathsf{ok}$ from the definition of $\Gamma\ \mathsf{ok}$. We can use this to prove all the axioms of the typing judgement. The rules follow directly by induction except for [TS-Intro]. For this rule, we must prove that x and o are different. This is true as they are from different syntactic categories. ■

A.4 Covariant subtyping of frame stack

Lemma 2.3 (Covariant subtyping of frame stack) $\forall H, VS, \tau_1, \tau_2, \tau_3$. *if* $\Delta, H, VS \vdash FS : \tau_1 \to \tau_2$ *and* $\tau_3 \prec \tau_1$ *then* $\exists \tau_4 . \Delta, H, VS \vdash FS : \tau_3 \to \tau_4$ *and* $\tau_4 \prec \tau_2$.

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.

Inductive Step

Show that 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 their 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 the inductive hypothesis. For the remainder of the cases it suffices to prove:

$$\underbrace{\Delta; \Gamma, \bullet : \tau \vdash OF : \tau_1}_{(11)} \wedge \underbrace{\tau_2 \prec \tau}_{(12)} \Rightarrow \exists \tau_3 . \underbrace{\Delta; \Gamma, \bullet : \tau_2 \vdash OF : \tau_3}_{(13)} \wedge \underbrace{\tau_3 \prec \tau_1}_{(14)}$$

We proceed by case analysis on OF.

Case: $OF =$ `if` $(\bullet$ `==` $e)\{\overline{s_1}\}$ `else` $\{\overline{s_2}\}$; From (11) we know

$$\frac{\overline{\Delta; \Gamma, \bullet : \tau \vdash \bullet : \tau} \quad \overline{\Delta; \Gamma, \bullet : \tau \vdash e : C}^{(15)} \quad \overline{\Delta; \Gamma, \bullet : \tau \vdash \overline{s_1} : \texttt{void}}^{(16)} \quad \overline{\Delta; \Gamma, \bullet : \tau \vdash \overline{s_2} : \texttt{void}}^{(17)}}{\Delta; \Gamma, \bullet : \tau \vdash \texttt{if}\ (\bullet\ \texttt{==}\ e)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};: \texttt{void}}$$

We need to prove

$$\frac{\overline{\Delta;\Gamma,\bullet:\tau_1\vdash\bullet:\tau_1}\quad\overset{(18)}{\overline{\Delta;\Gamma,\bullet:\tau_1\vdash e:C}}\quad\overset{(19)}{\overline{\Delta;\Gamma,\bullet:\tau_1\vdash\overline{s_1}:\texttt{void}}}\quad\overset{(20)}{\overline{\Delta;\Gamma,\bullet:\tau_1\vdash\overline{s_2}:\texttt{void}}}}{\Delta;\Gamma,\bullet:\tau_1\vdash\texttt{if}\ (\bullet\ \texttt{==}\ e)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};:\texttt{void}}$$

Clearly as e, $\overline{s_1}$ and $\overline{s_2}$ do not contain $\bullet$. Therefore we have (15) $\Rightarrow$ (18), (16) $\Rightarrow$ (19) and (17) $\Rightarrow$ (20), which completes this case.

Case: $OF = \texttt{if}\ (v\ \texttt{==}\ \bullet)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}$; Similar to previous case.

Case: $OF = \bullet.f$ From assumptions we know

$$\frac{\overline{\Delta;\Gamma,\bullet:\tau\vdash\bullet:\tau}\quad\overline{\Delta_f(\tau)(f)=C_2}}{\Delta;\Gamma,\bullet:\tau\vdash\bullet.f:C_2}$$

As $\tau_2 \prec \tau$ and field types can not be overridden, we know $\Delta_f(\tau_2)(f) = C_2$, which lets us prove

$$\Delta;\Gamma,\bullet:\tau_2\vdash\bullet.f:C_2$$

as required.

Case: $OF = \bullet.f\ \texttt{=}\ e$; Similar to previous case.

Case: $OF = \bullet.m(e_1,\ldots,e_n)$ Similar to previous case, except we use the fact method types can not be overridden.

Case: $OF = x\ \texttt{=}\ \bullet$; We know from the assumptions that

$$\frac{\overline{\Delta;\Gamma,\bullet:\tau\vdash x:C}\quad\overline{\Delta;\Gamma,\bullet:\tau\vdash\bullet:\tau}\quad\overline{\tau\prec C}}{\Delta;\Gamma,\bullet:\tau\vdash x\ \texttt{=}\ \bullet;:\texttt{void}}$$

As the sub-typing relation is transitive and $\tau_2 \prec \tau$, we can see this is well typed for $\bullet : \tau_2$, and the result type is the same.

Case: $OF = v.f\ \texttt{=}\ \bullet$; Similar to previous case.

Case: $OF = v.m(v_1,\ldots,v_{i-1},\bullet,e_{i+1},\ldots,e_n)$ Similar to previous case.

Case: $OF = \texttt{new}\ C(v_1,\ldots,v_{i-1},\bullet,e_{i+1},\ldots,e_n)$ Similar to previous case.

Case: $OF = \texttt{super}(v_1,\ldots,v_{i-1},\bullet,e_{i+1},\ldots,e_n)$ Similar to previous case.

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.

■

A.5 Type preservation

Proposition 2.4 *If* $\Delta \vdash (H, VS, F, FS) : \tau$ *and* $(H, VS, F, FS) \rightarrow (H', VS', F', FS')$ *then* $\exists \tau'.\Delta \vdash (H', VS', F', FS') : \tau'$ *where* $\tau' \prec \tau$.

Proof. This done by case analysis on the $\rightarrow$ relation. By considering all possible reductions we can show that the program will always reduce to a valid state, and that the state will be a subtype of the configuration type before the reductions.

All the rules for controlling the order of evaluation, defined in §3, are proved by trivially restructuring the typing derivations. We will show the rest of the cases now.

Case: [E-Skip]

$$\begin{array}{lrl} \text{Assume} & \Delta \vdash (H, VS, ;, F \circ FS) : \tau & (21) \\ \text{Prove} & \Delta \vdash (H, VS, F, FS) : \tau & (22) \end{array}$$

$$\dfrac{\dfrac{(23)}{\Delta \vdash H\ \mathsf{ok}} \quad \dfrac{(24)}{\Delta, H \vdash VS\ \mathsf{ok}} \quad \dfrac{\dfrac{(25)}{\Delta, H, VS \vdash F \circ FS : \mathtt{void} \rightarrow \tau} \quad \Delta; context(H, VS) \vdash ; : \mathtt{void}}{\Delta, H, VS \vdash (;) \circ F \circ FS : \mathtt{void} \rightarrow \tau}}{\Delta \vdash (H, VS, ;, F \circ FS) : \tau}$$

This lets us deduce the following:

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ {\scriptstyle 23} \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}\ {\scriptstyle 24} \quad \dfrac{}{\Delta, H, VS \vdash F \circ FS : \mathtt{void} \rightarrow \tau}\ {\scriptstyle 25}}{\Delta \vdash (H, VS, F, FS) : \tau}$$

and hence prove 22.

Case: [E-Sub]

$$\begin{array}{lrl} \text{Assume} & \Delta \vdash (H, VS, v, F \circ FS) : \tau & (26) \\ \text{Prove} & \Delta \vdash (H, VS, F[v/\bullet], FS) : \tau & (27) \end{array}$$

We split the proof into two cases. Firstly where $F = CF$ (a closed frame) and secondly where $F = OF$ (frame with a hole).

The proof needs to be split into two cases. One where F is a closed term, CF, and the second where F is a term requiring an expression, OF.

Case: $F = CF$ From (26) we get

$$\dfrac{\dfrac{(28)}{\vdash H\ \mathsf{ok}} \quad \dfrac{(29)}{\Delta, H \vdash VS\ \mathsf{ok}} \quad \dfrac{\Delta; context(H, VS) \vdash v : \tau' \quad \dfrac{(30)}{\Delta, H, VS \vdash CF \circ FS : \mathtt{void} \rightarrow \tau}}{\Delta, H, VS \vdash (v) \circ CF \circ FS : \mathtt{void} \rightarrow \tau}}{\Delta \vdash (H, VS, v, CF \circ FS) : \tau}$$

We know that $CF = CF[v/\bullet]$ as CF has no holes. Hence we can deduce

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ {\scriptstyle 28} \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}\ {\scriptstyle 29} \quad \dfrac{}{\Delta, H, VS \vdash CF \circ FS : \mathtt{void} \rightarrow \tau}\ {\scriptstyle 30}}{\Delta \vdash (H, VS, CF, FS) : \tau}}{\Delta \vdash (H, VS, CF[v/\bullet], FS) : \tau}$$

Which proves (27).

Case: $F = OF$ From (26) we get

$$\dfrac{\dfrac{\dfrac{}{\Delta; context(H,\, VS), \bullet : \tau' \vdash OF : \tau''}(31) \quad \dfrac{}{\Delta, H,\, VS \vdash FS : \tau'' \to \tau}(32)}{\Delta, H,\, VS \vdash OF \circ FS : \tau' \to \tau}(36)}{\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}(33) \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}(34) \quad \dfrac{\dfrac{}{\Delta; context(H,\, VS) \vdash v : \tau'}(35) \quad (36)}{\Delta, H,\, VS \vdash (v) \circ OF \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H,\, VS, v,\, OF \circ FS) : \tau}}$$

From this we can deduce

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\,33 \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}\,34 \quad \dfrac{\dfrac{}{\Delta; context(H,\, VS) \vdash OF[v/\bullet] : \tau''}(37) \quad \dfrac{}{\Delta, H,\, VS \vdash FS : \tau'' \to \tau}\,32}{\Delta, H,\, VS \vdash OF[v/\bullet] \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H,\, VS, OF[v/\bullet], FS) : \tau}$$

We need to show that (35)$\wedge$(31) $\Rightarrow$ (37). We can break this into two cases $v = \texttt{null}$ and $v = o$. The first is trivially true as `null` can have any type that is required, and hence plugging `null` in for any identifier will still leave the term typeable. The second case looks harder but this actually just alpha conversion, as both $\bullet$ and o are identifiers.

Hence we have proved (27).

Case: [E-Return]

Assume	$\Delta \vdash (H, MS \circ VS, \texttt{return}\ v;, FS) : \tau$	(38)
Prove	$\Delta \vdash (H,\, VS, v, FS) : \tau$	(39)

From (38) we have the following proof tree

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}(40) \quad \dfrac{}{\Delta, H \vdash MS \circ VS\ \mathsf{ok}}(41) \quad \dfrac{\dfrac{}{\Delta; context(H, MS \circ VS) \vdash v : \tau'}(42) \quad \dfrac{}{\Delta, H,\, VS \vdash FS : \tau' \to \tau}(43)}{\Delta, H, MS \circ VS \vdash (\texttt{return}\ v;) \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H, MS \circ VS, \texttt{return}\ v;, FS) : \tau}$$

For (39) we need the following tree.

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\,40 \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}(44) \quad \dfrac{\dfrac{}{\Delta; context(H,\, VS) \vdash v : \tau'}(45) \quad \dfrac{}{\Delta, H,\, VS \vdash FS : \tau' \to \tau}\,43}{\Delta, H,\, VS \vdash (v) \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H,\, VS, v,\, FS) : \tau}$$

From the definition of $\Delta, H \vdash VS\ \mathsf{ok}$ we can see that (41) $\Rightarrow$ (44).

We know a value's typing is not affected by the variable scope. Hence (42) $\Rightarrow$ (45), which proves (39).

Case: [E-VarAccess]

Assume	$\Delta \vdash (H, MS \circ VS, x, FS) : \tau$	(46)
	$eval(MS, x) = (v, C_2)$	(47)
Prove	$\Delta \vdash (H, MS \circ VS, v, FS) : \tau_1$	(48)
	$\tau_1 \prec \tau$	(49)

From (46) we can deduce the following tree.

$$\dfrac{\dfrac{(50)}{\Delta \vdash H\ \mathsf{ok}} \quad \dfrac{(51)}{\Delta, H \vdash MS \circ VS\ \mathsf{ok}} \quad \dfrac{\dfrac{(52)}{\Delta; context(H, MS \circ VS) \vdash x : C_2} \quad \dfrac{(53)}{\Delta, H, MS \circ VS \vdash FS : C_2 \to \tau}}{\Delta, H, MS \circ VS \vdash (x) \circ FS : \mathtt{void} \to \tau}}{\Delta \vdash (H, MS \circ VS, x, FS) : \tau}$$

To prove (48)

$$\dfrac{\dfrac{\dfrac{(54)}{\Delta; context(H, MS \circ VS) \vdash v : C_3} \quad \dfrac{(55)}{\Delta, H, MS \circ VS \vdash FS : C_3 \to \tau_1}}{\Delta, H, MS \circ VS \vdash (v) \circ FS : \mathtt{void} \to \tau_1}}{(56)}$$

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ {\scriptstyle 50} \quad \dfrac{}{\Delta, H \vdash MS \circ VS\ \mathsf{ok}}\ {\scriptstyle 51} \quad (56)}{\Delta \vdash (H, MS \circ VS, v, FS) : \tau_1}$$

Combining (47) and (51), we can deduce that v is well typed and that $C_3 \prec C_2$, which proves (54)

We use the Covariant Subtyping of the stack lemma

$$C_3 \prec C_2 \wedge (53) \Rightarrow (55) \wedge (49)$$

This completes the case.

Case: [E-VarWrite]

Assume	$\Delta \vdash (H, MS \circ VS, x = v;, FS) : \tau$	(57)
	$update(MS, (x \mapsto v)) \downarrow$	(58)
Prove	$\Delta \vdash (H, MS', ;, FS) : \tau$	(59)

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

From (57) we can deduce the following tree.

$$\dfrac{\dfrac{\dfrac{(61)}{\Delta; context(H, MS \circ VS) \vdash x : C} \quad \dfrac{(62)}{\Delta; context(H, MS \circ VS) \vdash v : C'} \quad \dfrac{(63)}{C' \prec C}}{\Delta; context(H, MS \circ VS) \vdash x = v; : \mathtt{void}}}{(60)}$$

$$\dfrac{\dfrac{}{\Delta \vdash H\ \texttt{ok}}(64) \quad \dfrac{}{\Delta, H \vdash MS \circ VS\ \texttt{ok}}(65) \quad \dfrac{\dfrac{}{\Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau}(66)}{\Delta, H, MS \circ VS \vdash (x = v;) \circ FS : \texttt{void} \to \tau}(60)}{\Delta \vdash (H, MS \circ VS, x = v;, FS) : \tau}$$

To deduce (59) we need the following proof.

$$\dfrac{\dfrac{\dfrac{}{\Delta; context(H, MS' \circ VS) \vdash; : \texttt{void}}\ \text{TS-Skip} \quad \dfrac{}{\Delta, H, MS' \circ VS \vdash FS : \texttt{void} \to \tau}(67)}{\Delta, H, MS' \circ VS \vdash; \circ FS : \texttt{void} \to \tau}}{(69)}$$

$$\dfrac{\dfrac{}{\vdash H\ \texttt{ok}}\ 64 \quad \dfrac{}{\Delta, H \vdash MS' \circ VS\ \texttt{ok}}(68) \quad (69)}{\Delta \vdash (H, MS' \circ VS, ;, FS) : \tau}$$

We know the only difference between MS and MS' is that one of the variable blocks contains the new value $x \mapsto (v, C)$. For this to be well-typed we require that

$$\Delta, H \vdash v : C$$

This is given by (62) and (63), which with (65), gives (68).

The typing information contained in MS and MS' is identical. Therefore (66) $\Rightarrow$ (67).

Case: [E-VarIntro]

Assume	$\Delta \vdash (H, (BS \circ MS) \circ VS, C\ x;, FS) : \tau$	(70)
Prove	$\Delta \vdash (H, (BS' \circ MS) \circ VS, ;, FS) : \tau'$	(71)

where $BS' = BS[x \mapsto (\texttt{null}, C)]$

From (70) we can deduce the following tree.

$$\dfrac{\dfrac{}{x \notin dom(BS \circ MS)}(72) \quad \dfrac{}{\Delta, H, (BS' \circ MS) \circ VS \vdash FS : \texttt{void} \to \tau}(73)}{(76)}$$

$$\dfrac{\dfrac{}{\Delta \vdash H\ \texttt{ok}}(74) \quad \dfrac{}{\Delta, H \vdash (BS \circ MS) \circ VS\ \texttt{ok}}(75) \quad \dfrac{}{\Delta, H, (BS \circ MS) \circ VS \vdash (C\ x;) \circ FS : \texttt{void} \to \tau}(76)}{\Delta \vdash (H, (BS \circ MS) \circ VS, x, FS) : \tau}$$

To prove (71)

$$\dfrac{\dfrac{\dfrac{}{\Delta; context(H, (BS' \circ MS) \circ VS) \vdash; : \texttt{void}}\ \text{TS-Skip} \quad \dfrac{}{\Delta, H, (BS' \circ MS) \circ VS \vdash FS : \texttt{void} \to \tau}\ 73}{\Delta, H, (BS' \circ MS) \circ VS \vdash (;) \circ FS : \texttt{void} \to \tau}}{(77)}$$

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ {\scriptstyle 74} \quad (77) \quad \dfrac{(78)}{\Delta, H \vdash (BS' \circ MS) \circ VS\ \mathsf{ok}}}{\Delta \vdash (H, (BS' \circ MS) \circ VS, ;, FS) : \tau}$$

By the definition of $\Delta, H \vdash VS\ \mathsf{ok}$ we know that (75) $\Rightarrow$ (78), as the additional variable is valid. Hence proving (71)

Case: [E-If1]

$$\begin{array}{lrl} \text{Assume} & \Delta \vdash (H,\, VS, \texttt{if}\ (v_1 \texttt{ == } v_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}, FS) : \tau & (79) \\ \text{Prove} & \Delta \vdash (H,\, VS, \{\overline{s}_1\}, FS) : \tau & (80) \end{array}$$

From (79) we can deduce

$$\dfrac{\Delta; \Gamma \vdash v_1 : \tau_1 \quad \Delta; \Gamma \vdash v_2 : \tau_2 \quad \dfrac{(81)}{\Delta; \Gamma \vdash \{\overline{s}_1\} : \texttt{void}} \quad \Delta; \Gamma \vdash \{\overline{s}_2\} : \texttt{void}}{84}$$

$$\dfrac{\dfrac{(82)}{\Delta \vdash H\ \mathsf{ok}} \quad \dfrac{(83)}{\Delta, H \vdash VS\ \mathsf{ok}} \quad \dfrac{\dfrac{(84)}{\Delta; \Gamma \vdash \texttt{if}\ (v_1 \texttt{ == } v_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}) : \texttt{void}} \quad \dfrac{(85)}{\Delta, H, VS \vdash FS : \texttt{void} \to \tau}}{\Delta, H, VS \vdash (\texttt{if}\ (v_1 \texttt{ == } v_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}) \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H, VS, \texttt{if}\ (v_1 \texttt{ == } v_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}, FS) : \tau}$$

where $\Gamma = context(H, VS)$.

From this we can deduce the following tree, and hence prove (80).

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ {\scriptstyle 82} \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}\ {\scriptstyle 83} \quad \dfrac{\dfrac{}{\Delta; \Gamma \vdash \{\overline{s_1}\} : \texttt{void}}\ {\scriptstyle 81} \quad \dfrac{}{\Delta, H, VS \vdash FS : \texttt{void} \to \tau}\ {\scriptstyle 85}}{\Delta, H, VS \vdash \{\overline{s_1}\} \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H, VS, \{\overline{s_1}\}, FS) : \tau}$$

Case: [E-If2] Identical to previous.

Case: [E-BlockIntro]

$$\begin{array}{lrl} \text{Assume} & \Delta \vdash (H, MS \circ VS, \{\overline{s}\}, FS) : \tau & (86) \\ \text{Prove} & \Delta \vdash (H, (\{\} \circ MS) \circ VS, \overline{s}, \{\} \circ FS) : \tau ! E & (87) \end{array}$$

We know

$$\dfrac{\dfrac{(88)}{\Delta, \vdash H\ \mathsf{ok}} \quad \dfrac{(89)}{\Delta, H \vdash MS \circ VS\ \mathsf{ok}} \quad \dfrac{\dfrac{\dfrac{(90)}{\Delta; context(H, MS \circ VS) \vdash \overline{s} : \texttt{void}}}{\Delta; context(H, MS \circ VS) \vdash \{\overline{s}\} : \texttt{void}} \quad \dfrac{(91)}{\Delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau}}{\Delta, H, MS \circ VS \vdash \{\overline{s}\} \circ FS : \texttt{void} \to \tau}}{\Delta \vdash (H, MS \circ VS, \{\overline{s}\}, FS) : \tau}$$

We need to prove

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ 88 \quad \dfrac{}{\Delta, H \vdash (\{\} \circ MS) \circ VS\ \mathsf{ok}}\ 89 \quad \dfrac{(92)}{\Delta, H, (\{\} \circ MS) \circ VS \vdash (\overline{s}) \circ \{\} \circ FS : \mathtt{void} \to \tau}}{\Delta \vdash (H, (\{\} \circ MS) \circ VS, \overline{s}, \{\} \circ FS) : \tau}$$

Using (90) and the definition of *context*, we can deduce $\Delta; context(Heap, (\{\} \circ MS) \circ VS) \vdash \overline{s} : \mathtt{void}$. Using this, (91) and Lemma A.1 gives us (92), hence completing this case.

Case: [E-Cast]

Assume	$\Delta \vdash (H,\, VS, (C_1)o, FS) : \tau$	(93)
	$H(o) = (C_2, \mathbb{F})$	(94)
	$C_2 \prec C_1$	(95)
Prove	$\Delta \vdash (H,\, VS, o, FS) : \tau'$	(96)
	$\tau' \prec \tau$	(97)

From (93) we can deduce

$$\dfrac{\dfrac{(98)}{\Delta \vdash H\ \mathsf{ok}} \quad \dfrac{(99)}{\Delta, H \vdash VS\ \mathsf{ok}} \quad \dfrac{\Delta; context(H, VS) \vdash (C_1)o : C_1 \quad \dfrac{(100)}{\Delta, H, VS \vdash FS : C_1 \to \tau}}{\Delta, H, VS \vdash ((C_1)o) \circ FS : \mathtt{void} \to \tau}}{\Delta \vdash (H, VS, (C_1)o, FS) : \tau}$$

We need to prove

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ 98 \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}\ 99 \quad \dfrac{\dfrac{}{\Delta; context(H, VS) \vdash o : C_2}\ 94 \quad \dfrac{(101)}{\Delta, H, VS \vdash FS : C_2 \to \tau'}}{\Delta, H, VS \vdash (o) \circ FS : \mathtt{void} \to \tau'}}{\Delta \vdash (H, VS, o, FS) : \tau')}$$

We use the Covariant subtyping lemma this to gives us:

$$(95) \wedge (100) \Rightarrow (97) \wedge (101)$$

Hence proving (96) and (97).

Case: [E-FieldWrite]

Assume	$\Delta \vdash (H,\, VS, o.f = v;, FS) : \tau$	(102)
Prove	$\Delta \vdash (H',\, VS, ;, FS) : \tau'$	(103)

where $H(o) = (C, \mathbb{F})$ and $H' = H[o \mapsto (C, \mathbb{F}[f \mapsto v])]$

From 102 we can deduce

$$\dfrac{\dfrac{(105)}{\Delta; \Gamma \vdash o : C} \quad \dfrac{(106)}{\Delta; \Gamma \vdash v : C_2} \quad \dfrac{(107)}{\Delta_f(C)(f) = (C_3)} \quad \dfrac{(108)}{C_2 \prec C_3}}{(104)}$$

$$\dfrac{\dfrac{(109)}{\Delta \vdash H\ \mathsf{ok}} \quad \dfrac{(110)}{\Delta, H \vdash VS\ \mathsf{ok}} \quad \dfrac{\dfrac{(104)}{\Delta; \Gamma \vdash o.f = v; : \mathtt{void}} \quad \dfrac{(111)}{\Delta, H, VS \vdash FS : \mathtt{void} \to \tau}}{\Delta, H, VS \vdash (o.f = v;) \circ FS : \mathtt{void} \to \tau}}{\Delta \vdash (H, VS, o.f = v;, FS) : \tau}$$

where $\Gamma = context(H, VS)$.

We need to prove the following tree.

$$\dfrac{\dfrac{(112)}{\Delta \vdash H'\ \mathsf{ok}} \quad \dfrac{}{\Delta, H' \vdash VS\ \mathsf{ok}}\ 110 \quad \dfrac{\dfrac{}{\Delta; context(H', VS) \vdash ; : \mathtt{void}}\ \text{TS-Skip} \quad \dfrac{(113)}{\Delta, H', VS \vdash FS : \mathtt{void} \to \tau}}{\Delta, H', VS \vdash (;) \circ FS : \mathtt{void} \to \tau}}{\Delta \vdash (H', VS, ;, FS) : \tau}$$

Both H and H' contain the same typing information, wrt the function *context*, hence (111) $\Rightarrow$ (113).

As we know (109) we only need to prove $H' \vdash o\ \mathsf{ok}$ for (112). To prove this we are required to show $\Delta, H' \vdash v : \Delta_f(C)(f)$, which is given by (105), (106), (107) and (108).

Case: [E-FieldAccess]

Assume	$(H, VS, o.f, FS) : \tau$	(114)
	$o \in dom(H)$	(115)
	$H(o) = (C, \mathbb{F})$	(116)
	$\mathbb{F}(f) = v$	(117)
Prove	$(H, VS, v, FS) : \tau_1$	(118)
	$\tau_1 \prec \tau$	(119)

From (114) we can deduce

$$\dfrac{\dfrac{(120)}{\Delta \vdash H\ \mathsf{ok}} \quad \dfrac{(121)}{\Delta, H \vdash VS\ \mathsf{ok}} \quad \dfrac{\dfrac{\dfrac{(122)}{\Delta; context(H, VS) \vdash o : C} \quad \dfrac{(123)}{\Delta_f(C)(f) = C_2}}{\Delta; context(H, VS) \vdash o.f : C_2} \quad \dfrac{(124)}{\Delta, H, VS \vdash FS : C_2 \to \tau}}{\Delta, H, VS \vdash (o.f) \circ FS : \mathtt{void} \to \tau}}{\Delta \vdash (H, VS, o.f, FS) : \tau}$$

We need to prove

$$\dfrac{\dfrac{}{\Delta \vdash H\ \mathsf{ok}}\ 120 \quad \dfrac{}{\Delta, H \vdash VS\ \mathsf{ok}}\ 121 \quad \dfrac{\dfrac{(125)}{\Delta; context(H, VS) \vdash v : C_3} \quad \dfrac{(126)}{\Delta, H, VS \vdash FS : C_3 \to \tau_1}}{\Delta, H, VS \vdash (v) \circ FS : \mathtt{void} \to \tau_1}}{\Delta \vdash (H, VS, v, FS) : \tau_1}$$

By the definition of $\Delta \vdash H\ \mathsf{ok}$, we can see that (120) $\wedge$ (122) $\wedge$ (123) $\Rightarrow$ (125) $\wedge\, C_3 \prec C_2$ and using covariant subtyping of the stack gives us $C_3 \prec C_2 \wedge$ (124) $\Rightarrow$ (126) $\wedge$ (119). Hence proving (118) and (119).

Case: [E-New]

$$\text{Assume} \qquad \Delta \vdash (H,\, VS, \texttt{new}\ C(v_1, \ldots, v_n), FS) : \tau \tag{127}$$

$$\text{Prove} \qquad \Delta \vdash (H',\, MS \circ VS, \texttt{super}(\overline{e}); \overline{s}, (\texttt{return}\ o;) \circ FS) : \tau \tag{128}$$

where

$$o \notin dom(H) \tag{129}$$

$$\mathbb{F} = \{f \mapsto null | (f \in dom(\Delta_f(C))\} \tag{130}$$

$$\Delta_C(C) = [C_1, \ldots, C_n] \tag{131}$$

$$cnbody(C) = [x_1, \ldots, x_n], \texttt{super}(\overline{e}); \overline{s} \tag{132}$$

$$MS = \{\texttt{this} \mapsto (o, C), x_1 \mapsto (v_1, C_1), \ldots, x_n \mapsto (v_n, C_n)\} \circ [] \tag{133}$$

$$H' = H[o \mapsto (C, \mathbb{F})] \tag{134}$$

From (127) we can deduce

$$\frac{\overline{\Gamma \vdash v_1 : C_1'} \quad \overset{(136)}{\cdots} \quad \overline{\Gamma \vdash v_n : C_n'} \quad \overline{C_1' \prec C_1} \quad \overset{(137)}{\cdots} \quad \overline{C_n' \prec C_n}}{(135)}$$

$$\frac{\overset{(138)}{\vdash H\ \texttt{ok}} \quad \overset{(139)}{\overline{H \vdash VS\ \texttt{ok}}} \quad \frac{\overset{(135)}{\overline{\Gamma \vdash \texttt{new}\ C(v_1, \ldots, v_n)}} \quad \overset{(140)}{\overline{H, VS \vdash FS : C \to \tau}}}{H, VS \vdash (\texttt{new}\ C(v_1, \ldots, v_n)) \circ FS : \texttt{void} \to \tau}}{(H, VS, \texttt{new}\ C(v_1, \ldots, v_n), FS) : \tau}$$

where $\Gamma = context(H, VS)$.

To prove (128)

$$\frac{\overset{(141)}{\vdash H'\ \texttt{ok}} \quad \overset{(142)}{\overline{H' \vdash MS \circ VS\ \texttt{ok}}} \quad \overset{(143)}{\overline{H', MS \circ VS \vdash (\texttt{super}(\overline{e}); \overline{s}) \circ (\texttt{return}\ o;) \circ FS : \texttt{void} \to \tau}}}{(H', MS \circ VS, \texttt{super}(\overline{e}); \overline{s}, (\texttt{return}\ o;) \circ FS) : \tau}$$

We can prove (141) as the only change from H to H' is adding a new object. As all its fields are set to `null` we know this is object is valid, and all the other objects are valid by (138).

We need to prove that MS is a valid variable scope wrt to H'. We know that `this` is of type C and so is o so this part is okay, as the subtype relation is reflexive. The other variables are all bound to values that are known to be valid from (136) and (137). Hence using (139) we know (142).

Using heap extension preserves typing we know (140) $\Rightarrow \Delta, H', VS \vdash FS : C \to \tau$.

From [T-ConsOK] we know that $\Delta; \Gamma \vdash \texttt{super}(\overline{e}); \overline{s} : \texttt{void}$ where $\Gamma = \{\texttt{this} : C, x_1 : C_1, \ldots, x_n : C_n\}$. We can see from the definition of context that $context(H', MS \circ VS) = \Gamma \uplus \Gamma'$ where Γ' contains only type assignments for object ids. By the extension property we know $\Delta; context(H', MS \circ VS) \vdash \texttt{super}(\overline{e}); \overline{s} : \texttt{void}$. We can extend this to give $\Delta; context(H', MS \circ VS) \vdash \texttt{super}(\overline{e}); \overline{s}\ \texttt{return}\ o; : C$, which allows us to use lemma A.2 to complete the case.

Case: [E-Method]

$$\begin{array}{lll} \text{Assume} & (H,\, VS, o.m(v_1,\ldots,v_n), FS) : \tau & (144)\\ \text{Prove} & (H, MS \circ VS, \overline{s}\ \texttt{return}\ e;, FS) : \tau & (145)\end{array}$$

where

$$\begin{array}{ll} H(o) = (C, \mathbb{F}) & (146)\\ \Delta_m(C)(m) = C_1, \ldots, C_n \to C' & (147)\\ mbody(C, m) = [x_1, \ldots, x_n], \overline{s}\ \texttt{return}\ e; & (148)\\ MS = \{\texttt{this} \mapsto (o, C), x_1 \mapsto (v_1, C_1), \ldots, x_n \mapsto (v_n, C_n)\} \circ [] & (149)\end{array}$$

From (144) we can deduce

$$\dfrac{\dfrac{(151)}{\Gamma \vdash o : C} \quad \overline{\Gamma \vdash v_1 : C_1'} \quad \overset{(152)}{\cdots} \quad \overline{\Gamma \vdash v_n : C_n'} \quad \overline{C_1' \prec C_1} \quad \overset{(153)}{\cdots} \quad \overline{C_n' \prec C_n}}{\dfrac{\Gamma \vdash o.m(v_1,\ldots,v_n) : C'}{(150)}}$$

$$\dfrac{\dfrac{(154)}{\vdash H\ \texttt{ok}} \quad \dfrac{(155)}{H \vdash VS\ \texttt{ok}} \quad \dfrac{(150) \quad \dfrac{(156)}{H, VS \vdash FS : C' \to \tau}}{H, VS \vdash (o.m(v_1,\ldots,v_n)) \circ FS : \texttt{void} \to \tau}}{H, VS, o.m(v_1,\ldots,v_n), FS) : \tau}$$

where $\Gamma = context(H, VS)$.
To prove (145) we need the following tree

$$\dfrac{\dfrac{}{\vdash H\ \texttt{ok}}\ {\scriptstyle 154} \quad \dfrac{(157)}{H \vdash MS \circ VS\ \texttt{ok}} \quad \dfrac{(158)}{H, MS \circ VS \vdash (\overline{s}\ \texttt{return}\ e;) \circ FS : \texttt{void} \to \tau}}{H, MS \circ VS, (\overline{s}\ \texttt{return}\ e;), FS) : \tau}$$

We need to show that MS is a valid variable scope. This can be seen to be true in the same way as the previous case, except for the typing of object which comes from (151). So we know (155) $\Rightarrow$ (157)

We know from methods ok that $\Delta; \Gamma \vdash \overline{s}\ \texttt{return}\ e; : C$ where $\Gamma = \{o : C, x_1 : C_1, \ldots, x_n : C_n\}$, by similar reasoning to previous case and again using lemma A.2 we can complete the case. TODO: `Requires Covariant subtyping lemma as well.`

TODO

Case: [E-MethodVoid] Same as previous case, with a trivial alteration for the `return` .

Case: [E-Super]

$$\begin{array}{lll} \text{Assume} & (H, MS \circ VS, \texttt{super}(v_1,\ldots,v_n), FS) : \tau & (159)\\ \text{Prove} & (H, MS' \circ MS \circ VS, \overline{s}, \texttt{return}\ o;, FS) : \tau' & (160)\\ & & (161)\end{array}$$

where

$$MS(\texttt{this}) = (o, C) \tag{162}$$
$$\Delta_c(C') = \{C_1, \ldots, C_n\} \tag{163}$$
$$cnbody(C') = [x_1, \ldots, x_n], \overline{s} \tag{164}$$
$$MS' = \{\texttt{this} \mapsto (o, C'), x_1 \mapsto (v_1, C_1), \ldots, x_n \mapsto (v_n, C_n)\} \circ [] \tag{165}$$
$$C \prec_1 C' \tag{166}$$

From (159) we can deduce

$$\dfrac{\dfrac{\overline{\Gamma \vdash v_1 : C'_1} \quad \overset{(168)}{\cdots} \quad \overline{\Gamma \vdash v_n : C'_n} \quad \overline{C'_1 \prec C_1} \quad \overset{(169)}{\cdots} \quad \overline{C'_n \prec C_n} \quad \overline{C \prec 1C}}{\Gamma \vdash \texttt{super}(v_1, \ldots, v_n) : \texttt{void}}}{(167)}$$

$$\dfrac{\dfrac{\overset{(170)}{\vdash H\ \texttt{ok}} \quad \overset{(171)}{\overline{H \vdash MS \circ VS\ \texttt{ok}}} \quad \overset{(167)}{\overline{H, MS \circ VS \vdash (\texttt{super}(v_1, \ldots, v_n)) \circ FS : \texttt{void} \to \tau}}}{H, MS \circ VS, \texttt{super}(v_1, \ldots, v_n), FS) : \tau} \quad \overset{(172)}{\overline{H, VS \vdash FS : \texttt{void} \to \tau}}}{}$$

where $\Gamma = context(H, VS)$.

To prove (160) we need the following proof

$$\dfrac{\overset{170}{\overline{\vdash H\ \texttt{ok}}} \quad \overset{(173)}{\overline{H \vdash MS' \circ MS \circ VS\ \texttt{ok}}} \quad \overset{(174)}{\overline{H, MS' \circ MS \circ VS \vdash (\overline{s}) \circ (\texttt{return}\ o;) \circ FS : \texttt{void} \to \tau}}}{H, MS' \circ MS \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS) : \tau}$$

We need to show that MS' is a valid variable scope. This is true in the same way as the previous case, except for the typing of `this`, which comes from MS, because `super` requires Γ to contain `this`. We know this to be valid from (171) and (167), so we know (171) $\Rightarrow$ (173)

We can prove (174) in the same way as for [E-New]. Hence we have proved the final case.

■

A.6 Progress with effects.

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. The proof of this lemma is identical to the proof in §A.1 except for the following two cases.

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 three cases.

Case: $e = \texttt{null}$ reduces by [E-NullField].

Case: $e = o$ reduces by [E-FieldAccess]. This reduction has effect $R(r)$, which we have in the typing judgement.

Case: $e \neq v$ reduces by [EC-FieldAccess].

Case: $F = e.f = e'$. The typing of this introduces the effect $W(r)$ where f is in the region r.

Case: $e \neq v$ reduces using [EC-FieldWrite1].

Case: $e = v \wedge e' \neq v'$ reduces using [EC-FieldWrite2].

Case: $e = o \wedge e' = v'$ reduces using [E-FieldWrite]. This reduction rule has the effect of $W(r)$, which is in the typing derivation.

Case: $e = \texttt{null} \wedge e' = v'$ reduces using [E-NullWrite].

■

A.7 Covariant subtyping of frame stack with effects

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. This is proved in the same way as the lemma without effects. The following cases need slight extension.

Case: $OF = \bullet.f$ From assumptions we know

$$\frac{\overline{\Delta; \Gamma, \bullet : \tau \vdash \bullet : \tau!\emptyset} \quad \overline{\Delta_f(\tau)(f) = (C_2, r)}}{\Delta; \Gamma, \bullet : \tau \vdash \bullet.f : C_2!R(r)}$$

As $\tau_1 \prec \tau$ and field types and regions can not be overridden, we know $\Delta_f(\tau_1)(f) = (C_2, r)$, which lets us prove

$$\Delta; \Gamma, \bullet : \tau_1 \vdash \bullet.f : C_2!R(r)$$

as required.

Case: $OF = \bullet.f = e$; Similar to previous case.

Case: $OF = \bullet.m(e_1, \ldots, e_n)$ Similar to previous case, except we use the fact method types and effects can not be overridden.

■

A.8 Type and effect 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* $\exists \tau'.\Delta \vdash (H', VS', F', FS') : \tau'!E_2$ *where* $\tau' \prec \tau$ *and* $E_2 \leq E_1$.

Proof. The proof proceeds in exactly the same way as for §A.5. The majority of cases either do not require any additional proof or follow directly from the extended covariant lemma. We will now present the additional information required to prove the extended type preservation proof.

The following cases follow using the same proof, but using the rules of the effect system: [E-Sub], [E-Skip], [E-Return], [E-VarWrite], [E-VarIntro]

The following cases follow from extending the proof trees and using the extended covariant lemma: [E-VarAccess], [E-Cast]

The remaining cases all alter the effect type. We must show that the new effect type is a subeffect of the old.

Case: [E-If1]

Assume	$\Delta \vdash (H, VS, \texttt{if}\ (v_1\ \texttt{==}\ v_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}, FS) : \tau!E$	(175)
Prove	$\Delta \vdash (H, VS, \{\overline{s}_1\}, FS) : \tau!E'$	(176)
	$E' \leq E$	(177)

If we consider $\{\overline{s}_1\}$ to have the effects E_1, $\{\overline{s}_2\}$ to have the effects E_2 and FS to have the effects E_3. Then we can see, before the reduction the effects are $E_1 \cup E_2 \cup E_3$ and after they are $E_1 \cup E_3$, which is clearly a subeffect.

Case: [E-If2] Identical to previous.

Case: [E-FieldWrite] Before the reduction this has the effects $E \cup W(r)$ and after E. Again the reduction only reduces the effects.

Case: [E-FieldAccess] Same as previous case but with a read effect.

Case: [E-New] Before the reduction the effects are the effect annotation given in Δ, after the reduction they are the effects of the actual implementation, but from [T-ConsOK] we know the implementation must has less effects. Hence the reduction reduces the effects.

Case: [E-Super] Same as previous case.

Case: [E-Method] Same as previous case, except we need to use the same property of methods.

Case: [E-MethodVoid] Same as previous case.

■

A.9 Inference algorithm produce sound annotations.

Theorem 3.8 *If $\Delta \vdash p : R_1$, $\vdash \Delta : R_2$ and θ satisfies $R_1 \cup R_2$ then $\theta(\Delta) \vdash p : \emptyset$.*

Proof. Assume

$$\Delta \vdash_i p : R \tag{178}$$
$$\vdash_i \Delta : R' \tag{179}$$
$$\theta \text{ satisfies } \mathrm{R} \cup \mathrm{R}' \tag{180}$$

Prove

$$\theta(\Delta) \vdash p \tag{181}$$

From (178) we know
$\forall j$

$$\Delta \vdash_i C_j \ \mathsf{ok} : R_j \tag{182}$$
$$R_j \subseteq R \tag{183}$$

Which gives

$$\Delta \vdash_i C_j \ \mathsf{mok} : R'_j \tag{184}$$
$$\Delta \vdash_i C_j \ \mathsf{cok} : R''_j \tag{185}$$
$$R_j = R'_j \cup R''_j \tag{186}$$

From (184) we know
$\forall k$

$$\Delta \vdash_i C_j.m_k : R'_{j,k} \tag{187}$$
$$R'_{j,k} \subseteq R'_j \tag{188}$$

From (187) we know

$$\Delta, \Gamma \vdash \mathit{body} : C'!E' \tag{189}$$
$$R'_{j,k} = E' \leq X \tag{190}$$
$$\Delta_m(C_j)(m_k) = \mu!X \tag{191}$$

By Lemma 3.7 with (189), (180) and (179) we can deduce

$$\theta(\Delta), \Gamma \vdash \mathit{body} : C'!\theta(E') \tag{192}$$

As $R'_{j,k} \subseteq R$ (by (183),(186) and (188)), (180) and (190) we have

$$\theta(E') \leq \theta(X) \tag{193}$$

By definition of substitution on Δ and (191)

$$\theta(\Delta_m)(C_j)(m_k) = \mu!\theta(X) \tag{194}$$

Using (192), (193) and (194) we can deduce

$$\theta(\Delta) \vdash C_j.m_k \ \mathsf{ok} \tag{195}$$

As we have shown this for an arbitrary k without any assumptions about it, so we know

$$\theta(\Delta) \vdash C_j \ \mathsf{mok} \tag{196}$$

A similar proof allows us to prove

$$\theta(\Delta) \vdash C_j \ \mathsf{cok} \tag{197}$$

which allows us to prove

$$\theta(\Delta) \vdash C_j \ \mathsf{ok} \tag{198}$$

for arbitrary j, so we can assume it for all j. Hence

$$\theta(\Delta) \vdash p$$

■