A TYPE-AND-EFFECT SYSTEM
FOR ENCAPSULATING MEMORY IN JAVA

by

BENNETT NORTON YATES

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

August 1999

"A Type-And-Effect System For Encapsulating Memory In Java," a thesis prepared by Bennett Norton Yates in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:

Dr. Amr Sabry

Date

Accepted by:

Dean of the Graduate School

An Abstract of the Thesis of

Bennett Norton Yates for the degree of Master of Science

in the Department of Computer and Information Science

to be taken August 1999

Title: A TYPE-AND-EFFECT SYSTEM FOR ENCAPSULATING MEMORY IN JAVA

Approved: ______________________________

Dr. Amr Sabry

This paper proposes a simple type-and-effect system for encapsulating memory in Java. Encapsulation is particularly relevant to Java because programs are frequently assembled from diverse sources. The ability to determine that an arbitrary expression will not affect other parts of the system is an important security issue. We start with an already developed core language and add an expression for encapsulation to the syntax. Then we extend the type system to include effect information. Our effects are simply region names and every type includes region information about the location of the instance, its fields, and the work that was done to produce it. No attempt is made to infer types or effects. Our system is rather limited but illustrates some of the issues that will have to be addressed in adapting any such system to Java.

CURRICULUM VITA

NAME OF AUTHOR: Bennett Norton Yates

PLACE OF BIRTH: Tempe, Arizona

DATE OF BIRTH: October 27, 1966

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon

DEGREES AWARDED:

Bachelor of Science, 1994, University of Oregon

AREAS OF SPECIAL INTEREST:

Programming Languages
Semantics

PROFESSIONAL EXPERIENCE:

Software Engineer

ACKNOWLEDGEMENTS

I learned a great deal more on this project than I ever expected. This is the direct result of the generous and clear guidance of my advisor, Dr. Amr Sabry, to whom I am immensely grateful.

TABLE OF CONTENTS

Chapter — Page

I. INTRODUCTION 1

II. RELATED WORK 3

- State In Haskell 3
- Type-And-Effect Systems 4
- Monadic ML 4
- ClassicJava 5

III. SYNTAX 6

- Java Syntax 6
- ClassicJava Syntax 7
- RegionJava Syntax 8
- Operators and Functions 10

IV. TYPES 17

V. SEMANTICS 24

Page

VI. TYPE SOUNDNESS 29

VII. CONCLUSION AND FURTHER WORK 37

APPENDIX

SUBJECT REDUCTION 38

BIBLIOGRAPHY 58

LIST OF FIGURES

Figure Page

1. Syntax 10
2. Operators 11
3. Class Member Operators 12
4. Interface Member Operators 13
5. Functions 14
6. Method Overriding Functions 15
7. Abstract Method Functions 16
8. Judgements 18
9. Type Rules For Defintions 20
10. Type Rules For Fields and Methods 21
11. Additional Type Rules For Expressions 22

Page

12. Extended Syntax 25

13. Additional Type Rules 25

14. Reduction Rules 27

15. Evaluation Contexts 28

16. Reductions to Errors 28

CHAPTER I

INTRODUCTION

The use of shared memory in a programming language has a limiting effect on our ability to understand programs written in that language. Remote sections of code can interact in complex ways through shared data and it may be difficult to distinguish between data which is relevant and that which is useless. Recent research has attempted to address these issues with varying forms of encapsulation. Encapsulation attempts to determine if an expression needs any information from or provides any information to the rest of the program. Interaction with the rest of the program is only allowed through explicit arguments or return values. In essence, an encapsulatable expression should produce the same result from the same arguments in any context. It should be able to run as a different process on a separate machine at any point during the overall evaluation of the program. Determining this statically would give compilers a great deal of power to optimize the order of evaluation and parallelize programs. In addition, from a security standpoint it would be highly desirable to know if a portion of a program will or will not have any effects on the rest of the program before it starts to run.

We investigate a system of encapsulating memory in a Java-like language. Encapsulation is especially relevant to Java because Java programs are frequently assembled from various sources and issues of securely determining the effects of any given expression before it is run are paramount. We propose a simple type-and-effect system where memory is partitioned into regions and an effect is simply the name of the region being read from or written to. We do not attempt any inference and different regions are never combined to create new larger regions. In this way the resulting system is fairly restricted. However, our research does reveal some of the interesting questions that must be addressed when adapting such an approach to Java or any object oriented language.

CHAPTER II

RELATED WORK

State In Haskell

One approach to encapsulating memory makes use of the concept of monads [6]. The monadic approach separates the type system of a language like Haskell into two parts: values, and computations. Functions like *read*, *write*, and *new*, that interact with the store have return types that belong to the set of computation types. Other parts of the language do not accept one of these types as an argument and thus expressions that use the store and those that do not are are prevented by the type system from interacting with each other directly. However, computations can be strung together with special connectors, $ThenST$ and $LetST$, which allow them to share the same store. Then, in order to return a value back to the rest of the program, a special transformer function, $RunST$, is provided that can translate computations into values. $RunST$ has special typing rules that assure that the store used by the string of computations is no longer needed. This requires a small change to the existing type system. In this manner, $RunST$ encapsulates the string of computations, preventing them from sharing their memory with the rest of the program.

Type-And-Effect Systems

Another approach is to extend the type system to explicitly identify and track the maximum possible effect of each statement [9, 7]. These "type-and-effect" systems use the type system to not only track the minimal type of an expression but also the maximal effect. These effects are of the form $get(r)$, $set(r)$, or $alloc(r)$, where r was the region of memory being affected. These effects are accumulated during the process of static analysis. Function types include a latent effect that represents the effect of executing the function body that does not occur until the function is called. A means of masking the effects of an expression is provided that enforces that the effects are not relevant to the rest of the program and local regions can be discarded. These systems often use powerful systems of inference to assign regions.

Monadic ML

A recent look at encapsulation in ML defines a simplified form of effect inference and masking and then gives a translation to monadic form that preserves the type, thus showing a relationship between the above two approaches [8]. Then it shows the soundness of the type system using a semantics that allows dangling pointers. Our research relies heavily on the techniques used in this semantics.

ClassicJava

ClassicJava was developed as a jumping off point for an analysis of “mixins” and polymorphic types [4]. It represents many of the interesting aspects of Java in a simple core language. We too use it as a jumping off point, though our direction is unrelated. It is discussed in detail later in this paper.

CHAPTER III

SYNTAX

Java Syntax

Java is an object oriented language in which data and methods are structured as part of the type definitions. Classes can extend other classes to form a tree-like hierarchy that has one unique root, named "Object", which is built into the language. By extending a class, the subclass inherits all of its fields and methods. This means that every field and method in every class on the path from the current class to the root of the tree is defined in the current class. A class cannot declare two fields of the same name. However, through inheritance, it is possible for multiple occurrences of a single field name to occur. Thus, fields are only uniquely identified by the combination of the field name and the source class name. When field names are encountered in the code the source class is determined by searching up the tree for the closest occurrence of that field name. The same is true for methods except that when searching for the closest occurrence of a method name in the hierarchy the argument types must be matched as well.

There is also a more subtle difference between the inheritance of fields and methods. The search for the source class of a field starts in the hierarchy at the static type of the instance while the search for the source class of a method does not take place until runtime and starts at the runtime type rather than the declared static type.

A method is run in the context of the instance which was used to invoke it and has access to all the data and methods of that instance. The method also has access to the keywords **this**, to refer to the instance itself, and **super**, to facilitate accessing fields or methods further up the hierarchy. New instances are created using the keyword **new** and fields and methods are accessed or invoked using the standard operator, ".".

In addition to classes, Java has an interface construct. This allows a type to be defined that just describes a set of method types. The method bodies are not defined in the interface definition but rather they are defined in any class definition that "implements" that interface. This results in a more flexible polymorphism by allowing diverse classes to all implement methods of the same signature, in possibly dramatically different ways, and be considered of the same type.

ClassicJava Syntax

The syntax of RegionJava is based on that of ClassicJava [4]. ClassicJava was developed as a core language from which to expand into a discussion of "mix-ins" and polymorphic types. It contains the basic features of Java-like classes and interfaces with inheritance and polymorphism. It also uses standard object oriented syntax like the "." operator to access an instance's fields and methods and includes Java keywords **new**, **this**, and **super** which have their usual meanings.

ClassicJava does differ from Java in some significant ways, however. Features like primitive types and static data that add considerable complexity without being particularly relevant to our purposes have been excluded. As a result, operators like Java's "+" operator would have to be simulated with addition methods in class representations of the relevant types like an "Integer" class. Thus, in ClassicJava everything is a reference to an instance of a class and the defined types are the defined classes.

In addition, method bodies are reduced from a series of statements to a single expression. The return value of the method is the result of the expression. A new **let** expression was added to introduce variables and scope. A program is a series of definitions followed by an expression. The expression roughly corresponds to the main method mechanism used by Sun's JDK except it is not considered within the context of an instance of a class.

RegionJava Syntax

The surface syntax of our RegionJava differs from ClassicJava only by the addition of an **encap** expression. The **encap** expression is used to mask the effects of a subexpression. Much of the type system is directed at ensuring that the **encap** expression will only type check if its subexpression neither affects nor is affected by the rest of the program. No data that is not explicitly provided or returned should enter or escape. There is, however, a significant difference in the elaborated syntax of method definitions. The elaborated syntax is not entered by the programmer but added later during static analysis to pass information gleaned from that analysis on to the runtime system. The elaborated syntax appears underlined for clarity. The process of elaboration will be discussed in the next chapter.

The effect of calling a method is different from the effect of defining the method. Yet we only want to analyze the method once. Thus this information is stored as part of the program where it can be looked up with each call. This information includes assigning a region variable to each argument including the implicit **this** argument that we have made explicit. Then the body is analyzed in the context of this region information to determine information about the result and the effect of evaluating the method when a call is made. All of this information is visible in the elaborated syntax of the method definition.

The annotation $\underline{(}\, t\ \underline{\mathtt{at}\ (\rho, \mathcal{R}))}$ for a reference adds to the declared type, t, the region in which that instance of t lies, ρ, and a description of that instance's components, $\mathcal{R}$. Each of the arguments to a method are annotated with this information as well as the instance itself in the form of an explicit **this** argument. The body is analyzed in the context of these annotations and an overall "latent" effect is determined as well as the return type. The latent effect is the effect that will occur each time the method is called. The latent and return effects annotate the method definition as well.

$$
\begin{array}{rcl}
P & = & \mathit{defn}^{*}\ e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \textbf{implements}\ i^{*}\ \{\ \mathit{field}^{*}\ \mathit{method}^{*}\ \} \\
 & | & \textbf{interface}\ i\ \textbf{extends}\ i^{*}\ \{\ \mathit{method}^{*}\ \} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{meth} & = & \underline{(}\ t\ \underline{\texttt{at}\ (\rho, \mathcal{R}))}\ \ \mathit{md}\ (\underline{(t\ \texttt{at}\ (\rho, \mathcal{R}))\ \textbf{this},}\ \mathit{arg}^{*}\ )\ \{\ \mathit{body}\ \}\underline{:\ \rho} \\
\mathit{arg} & = & \underline{(}\ t\ \underline{\texttt{at}\ (\rho, \mathcal{R}))}\ \ \mathit{var} \\
\mathit{body} & = & e\ |\ \textbf{abstract} \\
e & = & \textbf{new}\ c\ |\ \mathit{var}\ |\ \mathsf{null}\ |\ e\underline{:\ c}.\mathit{fd}\ |\ e\underline{:\ c}.\mathit{fd} = e \\
 & | & e.\mathit{md}\ (\ e^{*}\ )\ |\ \textbf{super}\underline{\equiv\ \textbf{this}:\ c}.\mathit{md}\ (\ e^{*}\ ) \\
 & | & \textbf{view}\ t\ e\ |\ \textbf{let}\ \mathit{var} = e\ \textbf{in}\ e\ |\ \textbf{encap}\ e \\
\mathit{var} & = & \text{a variable name}\ |\ \textbf{this} \\
c & = & \text{a class name}\ |\ \mathsf{Object} \\
i & = & \text{an interface name}\ |\ \mathsf{Empty} \\
\mathit{fd} & = & \text{a field name} \\
\mathit{md} & = & \text{a method name} \\
t & = & c\ |\ i \\
\rho & = & \text{a region variable} \\
\mathcal{R} & = & \text{a table of region variables}
\end{array}
$$

FIGURE 1. Syntax

The full syntax is given in figure 1.

Operators and Functions

We will use altered versions of the operators and functions defined for ClassicJava. The operators allow us to conveniently express relationships between classes or between a class and its members. Operators that relate classes to each other organize the definitions in a program into a tree hierarchy. Classes can extend at most one other class and cycles are not allowed. If a class does not explicitly extend another class it implicitly extends the built in Object class which forms the root of the tree. Other operators describe relationships between a class and its members and make use of this hierarchy. For example, a field, *fd*, of a given type, t, can be tested for membership in a given class, c, concisely and precisely with

$\prec^c_{P'}$

Class is declared as an immediate subclass
$c \prec^c_{P'} c' \Leftrightarrow$ **class** c **extends** $c' \cdots \{\cdots\}$ is in P$'$

$\prec^i_{P'}$

Interface is declared as an immediate subinterface
$i \ \prec^i_{P'} \ i' \Leftrightarrow$ **interface** i **extends** $\cdots i' \cdots \{\cdots\}$ is in P$'$

$\prec\!\!\!\prec^c_{P'}$

Class declares implementation of an interface
$c \prec\!\!\!\prec^c_{P'} i \Leftrightarrow$ **class** c **implements** $\cdots i \cdots \{\cdots\}$ is in P$'$

$\leq^c_{P'}$

Class is a subclass
$\leq^c_{P'} \equiv$ the transitive, reflexive closure of $\prec^c_{P'}$

$\leq^i_{P'}$

Interface is a subinterface
$\leq^i_{P'} \equiv$ the transitive, reflexive closure of $\prec^i_{P'}$

$\ll^c_{P'}$

Class implements an interface
$c \ll^c_{P'} i \Leftrightarrow \exists c', i' \ s.t. \ c \leq^c_{P'} c'$ and $i' \leq^i_{P'} i$ and $c' \prec\!\!\!\prec^c_{P'} i'$

$\leq_{P'}$

Type is a subtype
$\leq_{P'} \equiv \leq^c_{P'} \cup \leq^i_{P'} \cup \ll^c_{P'}$

$\in_{P'}$

Field or method is in a type
$\in_{P'} \equiv \in^c_{P'} \cup \in^i_{P'}$

FIGURE 2. Operators

$\langle c'.fd, \ t\rangle \in^c_{P'} c$. The distinction between c' and c is necessary because the class c may not explicitly declare the field fd but may inherit it from another class, in this case c'. This makes use of the hierarchical information by requiring c' to be the closest class on the path between c and Object that defines fd. This is defined as the minimum. The operator definitions are given in figures 2, 3, and 4.

The functions establish more general properties about a program. For example, establishing that all of the methods in all of the declared interfaces in a program, P, are abstract can be tested with InterfacesAbstract (P). The level of com-

$\Subset^c_{P'}$

Field is declared in a class

$\langle\ c.fd,\ t\ \rangle \Subset^c_{P'}\ c \Leftrightarrow$ **class** $c\ \cdots\{\cdots t\ fd\cdots\}$ is in P$'$

$\in^c_{P'}$

Field is contained in class

$$\begin{aligned}\langle\ c'.fd,\ t\ \rangle \in^c_{P'}\ c \Leftrightarrow\ & \langle\ c'.fd,\ t\ \rangle \Subset^c_{P'}\ c' \text{ and}\\ & c' = \texttt{min}\ \{\ c''\ \mid c \leq^c_{P'} c'' \texttt{ and } \exists t'\ s.t.\ \langle\ c''.fd,\ t'\ \rangle \Subset^c_{P'}\ \ c''\ \}\end{aligned}$$

$\Subset^c_{P'}$

Method is declared in a class

$$\begin{array}{l}\langle\ \ md,\\ \quad ((c\ \texttt{at}\ (\rho_0,\mathcal{R}_0)),\ldots,(t_n\ \texttt{at}\ (\rho_n,\mathcal{R}_n)) \xrightarrow{\rho_b} (t_r\ \texttt{at}\ (\rho_r,\mathcal{R}_r))),\\ \quad (var_1,\ldots,var_n),\\ \quad e_b\ \rangle \Subset^c_{P'}\ c \Leftrightarrow\\ \textbf{class}\ \ c\ \ \cdots\ \{\\ \qquad \cdots\\ \qquad \underline{(}\ t_r\ \ \underline{\texttt{at}\ (\rho_r,\mathcal{R}_r))}\ \ md\ (\ \ \underline{(\ c\ \texttt{at}\ (\rho_0,\mathcal{R}_0))\ \ \textbf{this}},\\ \qquad\qquad\qquad\qquad\qquad\quad \underline{(}\ t_1\ \ \underline{\texttt{at}\ (\rho_1,\mathcal{R}_1))}\ \ var_1,\\ \qquad\qquad\qquad\qquad\qquad\quad \ldots,\\ \qquad\qquad\qquad\qquad\qquad\quad \underline{(}\ t_n\ \ \underline{\texttt{at}\ (\rho_n,\mathcal{R}_n))}\ \ var_n\ \ )\ )\ \{\ \ e_b\ \ \}\underline{:\rho_b}\\ \qquad \cdots\\ \}\ \text{is in P}'\end{array}$$

$\in^c_{P'}$

Method is contained in class

$$\begin{array}{l}\langle\ \ md,\\ \quad ((c\ \texttt{at}\ (\rho_0,\mathcal{R}_0)),\ldots,(t_n\ \texttt{at}\ (\rho_n,\mathcal{R}_n)) \xrightarrow{\rho} (t_r\ \texttt{at}\ (\rho_r,\mathcal{R}_r))),\\ \quad (var_1,\ldots,var_n),\\ \quad e\ \rangle \in^c_{P'}\ c \Leftrightarrow\\ (\langle\ \ md,\\ \quad ((c'\ \texttt{at}\ (\rho_0,\mathcal{R}_0)),\ldots,(t_n\ \texttt{at}\ (\rho_n,\mathcal{R}_n)) \xrightarrow{\rho} (t_r\ \texttt{at}\ (\rho_r,\mathcal{R}_r))),\\ \quad (var_1,\ldots,var_n),\\ \quad e\ \rangle \Subset^c_{P'}\ c'\ \texttt{and}\\ c' = \texttt{min}\ \{\ c''\ \mid\ \ c \leq^c_{P'} c''\ \texttt{and}\\ \qquad\qquad\qquad \exists e', var'_1,\ldots,var'_n,\rho',\rho'_0,\ldots,\rho'_n,\rho'_r,\mathcal{R}'_0,\ldots,\mathcal{R}'_n,\mathcal{R}'_r\ s.t.\\ \qquad\qquad\qquad \langle\ \ md,\\ \qquad\qquad\qquad\quad ((c''\ \texttt{at}\ (\rho'_0,\mathcal{R}'_0)),\ldots,(t_n\ \texttt{at}\ (\rho'_n,\mathcal{R}'_n)) \xrightarrow{\rho'} (t_r\ \texttt{at}\ (\rho'_r,\mathcal{R}'_r))),\\ \qquad\qquad\qquad\quad (var'_1,\ldots,var'_n),\\ \qquad\qquad\qquad\quad e'\ \rangle \Subset^c_{P'}\ c''\ \})\end{array}$$

FIGURE 3. Class Member Operators

$\Subset^i_{P'}$

Method is declared in interface

$\langle\ md,$
$\quad ((i\ \mathtt{at}\ (\rho_0, \mathcal{R}_0)), \ldots, (t_n\ \mathtt{at}\ (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho_b} (t_r\ \mathtt{at}\ (\rho_r, \mathcal{R}_r))),$
$\quad (var_1, \ldots, var_n),$
$\quad \mathbf{abstract}\ \rangle \Subset^i_{P'} i \Leftrightarrow$
$\mathbf{interface}\ \ i\ \ \cdots\ \{$
$\quad \ldots$
$\quad \underline{(}\ t_r\ \underline{\mathtt{at}\ (\rho_r, \mathcal{R}_r))}\ \ md\ (\ \ \underline{(\ i\ \mathtt{at}\ (\rho_0, \mathcal{R}_0))\ \ \mathbf{this},}$
$\qquad\qquad\qquad\qquad \underline{(}\ t_1\ \underline{\mathtt{at}\ (\rho_1, \mathcal{R}_1))}\ \ var_1,$
$\qquad\qquad\qquad\qquad \ldots,$
$\qquad\qquad\qquad\qquad \underline{(}\ t_n\ \underline{\mathtt{at}\ (\rho_n, \mathcal{R}_n))}\ \ var_n\ )\ )\ \ \{\ \ \mathbf{abstract}\ \ \}\underline{:\rho_b}$
$\quad \ldots$
$\}$ is in P$'$

$\in^i_{P'}$

Method is contained in interface

$\langle\ md,$
$\quad ((i\ \mathtt{at}\ (\rho_0, \mathcal{R}_0)) \ldots (t_n\ \mathtt{at}\ (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho} (t_r\ \mathtt{at}\ (\rho_r, \mathcal{R}_r))),$
$\quad (var_1, \ldots, var_n),$
$\quad \mathbf{abstract}\ \rangle \in^i_{P'} i \Leftrightarrow$
$\exists i', \rho', \rho'_0 \ldots \rho'_n, \mathcal{R}'_0 \ldots \mathcal{R}'_n, var'_1, \ldots, var'_n\ \ s.t.$
$i \leq^i_{P'} i'\ \mathtt{and}$
$\langle\ md,$
$\quad ((i'\ \mathtt{at}\ (\rho'_0, \mathcal{R}'_0)) \ldots (t_n\ \mathtt{at}\ (\rho'_n, \mathcal{R}'_n)) \xrightarrow{\rho'} (t_r\ \mathtt{at}\ (\rho'_r, \mathcal{R}'_r))),$
$\quad (var'_1, \ldots, var'_n),$
$\quad \mathbf{abstract}\ \rangle \Subset^i_{P'} i'$

FIGURE 4. Interface Member Operators

$\textsc{ClassesOnce}(\mathrm{P}')$
Each class name is declared only once
$\forall c, c'$ **class** $c \cdots$ **class** c' $\cdots$ is in $\mathrm{P}' \Longrightarrow c \neq c'$

$\textsc{FieldOncePerClass}(\mathrm{P}')$
Field names in each class declaration are unique
$\forall fd, fd'$ **class** $\cdots \{\cdots fd \cdots fd' \cdots\}$ is in $\mathrm{P}' \Longrightarrow fd \neq fd'$

$\textsc{MethodOncePerClass}(\mathrm{P}')$
Method names in each class declaration are unique
$\forall md, md'$ **class** $\cdots \{\cdots md(\cdots)\ \{\cdots\} \cdots md'(\cdots)\ \{\cdots\} \cdots\}$ is in $\mathrm{P}' \Longrightarrow md \neq md'$

$\textsc{InterfacesOnce}(\mathrm{P}')$
Each interface name is declared only once
$\forall i, i'$ **interface** $i \cdots$ **interface** i' $\cdots$ is in $\mathrm{P}' \Longrightarrow i \neq i'$

$\textsc{InterfacesAbstract}(\mathrm{P}')$
Method declarations in each interface are abstract
$\forall md, e$ **interface** $\cdots \{\cdots\ md(\cdots)\ \{e\} \cdots\}$ is in $\mathrm{P}' \Longrightarrow e$ is **abstract**

$\textsc{CompleteClasses}(\mathrm{P}')$
Classes that are extended are defined
$\mathtt{rng}(\prec^c_{P'}) \subseteq \mathtt{dom}(\prec^c_{P'}) \cup \{\mathtt{Object}\}$

$\textsc{WellFoundedClasses}(\mathrm{P}')$
Class hierarchy is an order
$\leq^c_{P'}$ is antisymmetric

$\textsc{CompleteInterfaces}(\mathrm{P}')$
Extended/implemented interfaces are defined
$\mathtt{rng}(\prec^i_{P'}) \cup \mathtt{rng}(\prec\!\!\!\prec^c_{P'}) \subseteq \mathtt{dom}(\prec^i_{P'}) \cup \{\mathtt{Empty}\}$

$\textsc{WellFoundedInterfaces}(\mathrm{P}')$
interface hierarchy is an order
$\leq^i_{P'}$ is antisymmetric

FIGURE 5. Functions

plexity in the function definitions is high. We have proposed a set of functions that appear to capture the essential aspects of our system. However, certain subtleties need further examination and it is likely further refinement will be necessary. The function definitions are given in figures 5, 6, and 7.

$\textsc{ClassMethodsOK}(\mathrm{P}')$
Method overriding preserves the type
$\forall c, c', e, e', md, t_0, \ldots, t_n, t_r, \rho_0, \ldots, \rho_n, \rho_r, \mathcal{R}_0, \ldots, \mathcal{R}_n, \mathcal{R}_r,$
$t'_0, \ldots, t'_n, t'_r, \rho'_0, \ldots, \rho'_n, \rho'_r, \mathcal{R}'_0, \ldots, \mathcal{R}'_n, \mathcal{R}'_r, var_1, \ldots, var_n.$
$(\langle\ md,$
$((t_0 \texttt{ at } (\rho_0, \mathcal{R}_0)), \ldots, (t_n \texttt{ at } (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho_b} (t_r \texttt{ at } (\rho_r, \mathcal{R}_r))),$
$(var_1, \ldots, var_n),$
$e_b\ \rangle \in^c_{P'} t_0$ $\texttt{and}$
$\langle\ md,$
$((t'_0 \texttt{ at } (\rho'_0, \mathcal{R}'_0)), \ldots, (t'_n \texttt{ at } (\rho'_n, \mathcal{R}'_n)) \xrightarrow{\rho'_b} (t'_r \texttt{ at } (\rho'_r, \mathcal{R}'_r))),$
$(var'_1, \ldots, var'_n),$
$e'_b\ \rangle \in^c_{P'} t'_0\ ) \Rightarrow$
$(((t_i = t'_j \texttt{ for } j \in [1,n], \texttt{ and } t_r = t'_r),$
$(\mathcal{R}_j = \mathcal{R}'_j \texttt{ for } j \in [1,n], \texttt{ and } \mathcal{R}_r = \mathcal{R}'_r, \texttt{ and } \mathcal{R}'_0 \subseteq \mathcal{R}_0), \texttt{ and}$
$(\rho_j = \rho'_j \texttt{ for } j \in [0,n],\ \rho_r = \rho'_r, \texttt{ and } \rho_b = \rho'_b)) \texttt{ or}$
$t_0 \not\leq^c_{P'} t'_0)$

$\textsc{InterfaceMethodsOK}(\mathrm{P}')$
Redeclarations of methods are consistent
$\forall i, i', md, t_0, \ldots, t_n, t_r, \rho_0, \ldots, \rho_n, \rho_r, \mathcal{R}_0, \ldots, \mathcal{R}_n, \mathcal{R}_r,$
$t'_0, \ldots, t'_n, t'_r, \rho'_0, \ldots, \rho'_n, \rho'_r, \mathcal{R}'_0, \ldots, \mathcal{R}'_n, \mathcal{R}'_r, var_1, \ldots, var_n.$
$(\langle\ md,$
$((t_0 \texttt{ at } (\rho_0, \mathcal{R}_0)) \ldots (t_n \texttt{ at } (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho_b} (t_r \texttt{ at } (\rho_r, \mathcal{R}_r))),$
$(var_1, \ldots, var_n),$
$\mathbf{abstract}\ \rangle \in^i_{P'} t_0$ $\texttt{and}$
$\langle\ md,$
$((t'_0 \texttt{ at } (\rho'_0, \mathcal{R}'_0)), \ldots (t'_n \texttt{ at } (\rho'_n, \mathcal{R}'_n)) \xrightarrow{\rho'_b} (t'_r \texttt{ at } (\rho'_r, \mathcal{R}'_r))),$
$(var'_1, \ldots, var'_n),$
$\mathbf{abstract}\ \rangle \in^i_{P'} t'_0\ ) \Rightarrow$
$(((t_j = t'_j \texttt{ for } j \in [1,n], \texttt{ and } t_r = t'_r),$
$(\mathcal{R}_j = \mathcal{R}'_j \texttt{ for } j \in [1,n], \texttt{ and } \mathcal{R}_r = \mathcal{R}'_r, \texttt{ and } \mathcal{R}'_0 \subseteq \mathcal{R}_0), \texttt{ and}$
$(\rho_j = \rho'_j \texttt{ for } j \in [0,n],\ \rho_r = \rho'_r, \texttt{ and } \rho_b = \rho'_b)) \texttt{ or}$
$\forall t''(t'' \not\leq^i_{P'} t_0 \texttt{ or } t'' \not\leq^i_{P'} t'_0))$

FIGURE 6. Method Overriding Functions

$\textsc{ClassesImplementAll}(\mathrm{P}')$

Classes supply methods to implement interfaces

$\forall i, c\ c \ll^{c}_{P'} i \Longrightarrow$

$(\langle\ md,$

$((i \texttt{ at } (\rho_0, \mathcal{R}_0)), \ldots, (t_n \texttt{ at } (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho_b} (t_r \texttt{ at } (\rho_r, \mathcal{R}_r))),$

$(var_1, \ldots, var_n),$

$\textbf{abstract}\ \rangle \in^{c}_{P'} i \ \Longrightarrow$

$\langle\ md,$

$((c \texttt{ at } (\rho'_0, \mathcal{R}'_0)), \ldots, (t'_n \texttt{ at } (\rho'_n, \mathcal{R}'_n)) \xrightarrow{\rho'_b} (t'_r \texttt{ at } (\rho'_r, \mathcal{R}'_r))),$

$(var'_1, \ldots, var'_n),$

$e'_b\ \rangle \in^{c}_{P'} c$ $\texttt{and}$

$(t_i = t'_i \texttt{ for } i \in [1, n],\ \texttt{and } t_r = t'_r),$

$(\mathcal{R}_i = \mathcal{R}'_i \texttt{ for } i \in [1, n],\ \texttt{and } \mathcal{R}_r = \mathcal{R}'_r,\ \texttt{and } \mathcal{R}'_0 \subseteq \mathcal{R}_0),$ $\texttt{and}$

$(\rho_i = \rho'_i \texttt{ for } i \in [0, n],\ \rho_r = \rho'_r,\ \rho_b = \rho'_b)$

$\textsc{NoAbstractMethods}(\mathrm{P}',\ c)$

Class has no **abstract** methods (can be instantiated)

$\forall md, t_0, \ldots, t_n, t_r, \rho_0, \ldots, \rho_n, \rho_r, \rho_b, \mathcal{R}_0, \ldots, \mathcal{R}_n, \mathcal{R}_r, var_1, \ldots, var_n, e_b$

$\langle\ md,$

$((c \texttt{ at } (\rho_0, \mathcal{R}_0)) \ldots (t_n \texttt{ at } (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho_b} (t_r \texttt{ at } (\rho_r, \mathcal{R}_r))),$

$(var_1, \ldots, var_n),$

$e_b\ \rangle \in^{c}_{P'} c \ \Rightarrow$

$e_b \neq \textbf{abstract}$

FIGURE 7. Abstract Method Functions

CHAPTER IV

TYPES

Static analysis in RegionJava has greatly extended the types of ClassicJava to include information about regions. Memory is partitioned into regions to facilitate independent "workspaces" for different encapsulated expressions. Effects in our system are simply names for these different regions. We make no distinction between the reading from and writing to a region and only observe the fact that some interaction with that region occurred. References to instances are annotated with the region in which the instance lies. In addition, since instances are records of fields and fields are themselves references to instances that lie in regions, references are also annotated with a description of their components. This annotation takes the form of a lookup table from field identifiers to region information. Thus each reference has a type description of the form (t $\texttt{at}$ $(\rho, \mathcal{R})$). This includes the type, t, the region in which that instance of t lies, ρ, and a description of that instance's components, $\mathcal{R}$.

An expression's type-and-effect description is of the form ((t $\texttt{at}$ $(\rho', \mathcal{R})$) ! ρ). The first part of this description, up to the "!", includes all the information of the previous paragraph needed to describe the resulting reference. After the "!" is a additional region which indicates where the work was done that produced that

	$\vdash_P$	$\mathrm{P} \Rightarrow \mathrm{P}' \ : \ ((t \texttt{ at } (\rho_1, \mathcal{R})) \ ! \ \rho_2)$
		P elaborates to P′ with type – and – effect
P'	$\vdash_d$	$\mathit{defn} \Rightarrow \mathit{defn}'$
		defn elaborates to *defn′*
$\mathrm{P}' \ t$	$\vdash_m$	$\mathit{meth} \Rightarrow \mathit{meth}'$
		meth in t elaborates to *meth′*
$\mathrm{P}' \ \Gamma$	$\vdash_e$	$e \Rightarrow e' \ : \ ((t \texttt{ at } (\rho_1, \mathcal{R})) \ ! \ \rho_2)$
		e elaborates to e' and has given type – and – effect
$\mathrm{P}' \ \Gamma$	$\vdash_s$	$e \Rightarrow e' \ : \ ((t \texttt{ at } (\rho_1, \mathcal{R})) \ ! \ \rho_2)$
		uses subsumption
P'	$\vdash_t$	t
		t is a defined class or interface name

FIGURE 8. Judgements

result.

The basic structure of the type rules is similar to that of ClassicJava. Judgments are of various types as described in Figure 8. For example, the judgment for expressions uses the elaborated program, P′, and a type environment, Γ, to elaborate an expression and determine its type. Judgments for methods use the elaborated program and the class in which the method occurs to elaborate the method definition. It is important to note that many of these judgments use P′. This means that an elaborated program is necessary for the elaboration process. Thus the process is not currently one of discovery but rather one that checks information that has already been provided up front. This is a big barrier that will have to be overcome before any attempt to implement the system is made.

Elaboration of expressions is written $e \Rightarrow e'$. Other forms of elaboration are written similarly. Elaboration is the process of recording static analysis information to be used later by the runtime system. This occurs in three instances and is underlined in the syntax for clarity. First, since in Java the field to be accessed is determined based on the static type of the expression, and the static type of an expression may differ from the runtime type, the static type is inserted into all the field access expressions. Similarly, in Java the type of the keyword **super** is determined during static analysis, and is therefore also inserted into expressions using **super**. In addition, unlike ClassicJava, method definitions in RegionJava contain elaborated information. This allows the region of the return type to be based on the effects of the method body rather than be forced to match the location of the instance in which the method is invoked. This allows work done by the method to take place in a remote region as long as it does not access the **this** pointer.

The type rules are very similar to ClassicJava except that in each rule, RegionJava, in addition to checking type information, also checks region information. For example, in ClassicJava, the rule for setting a field is:

$$\frac{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e e \Rightarrow e' \ : \ t_2 \\ \langle\ c.fd,\ t_1\ \rangle \in_{P'} t_2 \\ \mathrm{P}'\Gamma \vdash_s e_v \Rightarrow e'_v \ : \ t_1 \end{array}}{\mathrm{P}'\Gamma \vdash_e e.fd = e_v \Rightarrow e' \underline{:\ c}.fd = e'_v \ : \ t_1}$$

In RegionJava this becomes:

$$\frac{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e e \Rightarrow e' \ : \ ((t_2 \ \mathtt{at}\ (\rho, \mathcal{R}))\ !\ \rho) \\ \langle\ c.fd,\ t_1\ \rangle \in_{P'} t_2 \\ \mathrm{P}'\Gamma \vdash_s e_v \Rightarrow e'_v \ : \ ((t_1 \ \mathtt{at}\ \mathcal{R}(c.fd))\ !\ \rho) \end{array}}{\mathrm{P}'\Gamma \vdash_e e.fd = e_v \Rightarrow e' \underline{:\ c}.fd = e'_v \ : \ ((t_1 \ \mathtt{at}\ \mathcal{R}(c.fd))\ !\ \rho)}$$

$$\frac{\begin{array}{ll}
\textsc{ClassesOnce}(\mathrm{P}') & \textsc{InterfacesOnce}(\mathrm{P}') \\
\textsc{MethodOncePerClass}(\mathrm{P}') & \textsc{FieldOncePerClass}(\mathrm{P}') \\
\textsc{CompleteClasses}(\mathrm{P}') & \textsc{WellFoundedClasses}(\mathrm{P}') \\
\textsc{CompleteInterfaces}(\mathrm{P}') & \textsc{WellFoundedInterfaces}(\mathrm{P}') \\
\textsc{ClassFieldsOK}(\mathrm{P}') & \textsc{ClassMethodsOK}(\mathrm{P}') \\
\textsc{InterfaceMethodsOK}(\mathrm{P}') & \textsc{InterfacesAbstract}(\mathrm{P}') \\
\textsc{ClassesImplementAll}(\mathrm{P}') & \\
\mathrm{P} = \mathit{defn}_1\ \ldots\ \mathit{defn}_n\ e & \\
\mathrm{P}' = \mathit{defn}'_1\ \ldots\ \mathit{defn}'_n\ e' & \\
\mathrm{P}' \vdash_d \mathit{defn}_j \Rightarrow \mathit{defn}'_j \text{ for } j \in [1,n] & \\
\mathrm{P}'\ []\ \vdash_e e \Rightarrow e'\ :\ ((t\ \texttt{at}\ (\rho', \mathcal{R}))\ !\ \rho) &
\end{array}}{\mathrm{P}' \vdash_P \mathit{defn}_1 \ldots \mathit{defn}_n\ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e'\ :\ ((t\ \texttt{at}\ (\rho', \mathcal{R}))\ !\ \rho)}\ [\text{prog}]$$

$$\frac{\begin{array}{l}
\mathrm{P}' \vdash_t t_j \text{ for each } j \in [1,n] \\
\mathrm{P}' c \vdash_m \mathit{meth}_k \Rightarrow \mathit{meth}'_k \text{ for each } k \in [1,p]
\end{array}}{\begin{array}{l}
\mathrm{P}' \vdash_d \textbf{class}\ c\ \ldots\ \{\ t_1\ \mathit{fd}_1\ \ldots\ t_n\ \mathit{fd}_n\ \mathit{meth}_1\ \ldots\ \mathit{meth}_p\ \} \Rightarrow \\
\qquad \textbf{class}\ c\ \ldots\ \{\ t_1\ \mathit{fd}_1\ \ldots\ t_n\ \mathit{fd}_n\ \mathit{meth}'_1\ \ldots\ \mathit{meth}'_p\ \}
\end{array}}\ [\text{cdefn}]$$

$$\frac{\mathrm{P}' i \vdash_m \mathit{meth}_j \Rightarrow \mathit{meth}'_j \text{ for each } j \in [1,p]}{\begin{array}{l}
\mathrm{P}' \vdash_d \textbf{interface}\ i\ \ldots\ \{\ \mathit{meth}_1\ \ldots\ \mathit{meth}_p\ \}\ \Rightarrow \\
\qquad \textbf{interface}\ i\ \ldots\ \{\ \mathit{meth}'_1\ \ldots\ \mathit{meth}'_p\ \}
\end{array}}\ [\text{idefn}]$$

$$\frac{\begin{array}{l}
\mathrm{P}' \vdash_t t_r \\
\mathrm{P}' \vdash_t t_j \text{ for } j \in [1,n] \\
\mathrm{P}'\ [\ \ \textbf{this} \mapsto (t_0\ \texttt{at}\ (\rho_0, \mathcal{R}_0)), \\
\qquad \mathit{var}_1 \mapsto (t_1\ \texttt{at}\ (\rho_1, \mathcal{R}_1)), \\
\qquad \ldots, \\
\qquad \mathit{var}_n \mapsto (t_n\ \texttt{at}\ (\rho_n, \mathcal{R}_n))\ ] \vdash_s e_b \Rightarrow e'_b\ :\ ((t_r\ \texttt{at}\ (\rho_r, \mathcal{R}_r))\ !\ \rho_b)
\end{array}}{\begin{array}{l}
\mathrm{P}'\ t_0 \vdash_m\ \ t_r\ \mathit{md}(t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\ \{\ e_b\ \} \Rightarrow \\
\qquad \underline{(}\ t_r\ \underline{\texttt{at}\ (\rho_r, \mathcal{R}_r))}\ \ \mathit{md}\ (\ \ \underline{(\ t_0\ \texttt{at}\ (\rho_0, \mathcal{R}_0))\ \ \textbf{this},} \\
\qquad\qquad\qquad\qquad \underline{(}\ t_1\ \underline{\texttt{at}\ (\rho_1, \mathcal{R}_1))}\ \ \mathit{var}_1, \\
\qquad\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad\qquad \underline{(}\ t_n\ \underline{\texttt{at}\ (\rho_n, \mathcal{R}_n))}\ \ \mathit{var}_n\ )\ \ \ \{\ \ e'_b\ \ \}\underline{:\ \rho_b}
\end{array}}\ [\text{meth}]$$

$$\frac{t \in \mathrm{dom}(\prec^c_{P'}) \cup \mathrm{dom}(\prec^i_{P'}) \cup \{\textsf{Object}, \textsf{Empty}\}}{\mathrm{P}' \vdash_t t}\ [\text{type}]$$

FIGURE 9. Type Rules For Defintions

$$\frac{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e e \Rightarrow e' \;:\; ((t_2 \;\texttt{at}\; (\rho, \mathcal{R}))\;!\;\rho) \\ \langle\; c.fd,\; t_1 \;\rangle \in_{P'} t_2 \end{array}}{\mathrm{P}'\Gamma \vdash_e e.fd \Rightarrow e'\underline{:c}.fd \;:\; ((t_1 \;\texttt{at}\; \mathcal{R}(c.fd))\;!\;\rho)} \;[\text{get}]$$

$$\frac{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e e \Rightarrow e' \;:\; ((t_2 \;\texttt{at}\; (\rho, \mathcal{R}))\;!\;\rho) \\ \langle\; c.fd,\; t_1 \;\rangle \in_{P'} t_2 \\ \mathrm{P}'\Gamma \vdash_s e_v \Rightarrow e'_v \;:\; ((t_1 \;\texttt{at}\; \mathcal{R}(c.fd))\;!\;\rho) \end{array}}{\mathrm{P}'\Gamma \vdash_e e.fd = e_v \Rightarrow e'\underline{:c}.fd = e'_v \;:\; ((t_1 \;\texttt{at}\; \mathcal{R}(c.fd))\;!\;\rho)} \;[\text{set}]$$

$$\frac{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e e \Rightarrow e' \;:\; ((t_0 \;\texttt{at}\; (\rho_0, \mathcal{R}_0))\;!\;\rho) \\ \mathrm{P}'\Gamma \vdash_s e_j \Rightarrow e'_j \;:\; ((t_j \;\texttt{at}\; (\rho_j, \mathcal{R}_j))\;!\;\rho) \text{ for } j \in [1, n] \\ \langle\;\; md, \\ \quad ((t_0 \;\texttt{at}\; (\rho_0, \mathcal{R}_0)) \ldots (t_n \;\texttt{at}\; (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho} (t_r \;\texttt{at}\; (\rho_r, \mathcal{R}_r)), \\ \quad (var_1 \ldots var_n), \\ \quad e_b \;\rangle \in_{P'} t_0 \end{array}}{\mathrm{P}'\Gamma \vdash_e e.md(e_1, \ldots, e_n) \Rightarrow e'.md(e'_1, \ldots, e'_n) \;:\; ((t_r \;\texttt{at}\; (\rho_r, \mathcal{R}_r))\;!\;\rho)} \;[\text{call}]$$

$$\frac{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e \mathbf{this} \Rightarrow \mathbf{this} \;:\; ((c_2 \;\texttt{at}\; (\rho_0, \mathcal{R}_0))\;!\;\rho) \\ c_2 \;\prec^c_{P'}\; c_1 \\ \mathrm{P}'\Gamma \vdash_s e_j \Rightarrow e'_j \;:\; ((t_j \;\texttt{at}\; (\rho_j, \mathcal{R}_j))\;!\;\rho) \text{ for } j \in [1, n] \\ e_b \neq \mathbf{abstract} \\ \langle\;\; md, \\ \quad ((c_1 \;\texttt{at}\; (\rho_0, \mathcal{R}_0)) \ldots (t_n \;\texttt{at}\; (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho} (t_r \;\texttt{at}\; (\rho_r, \mathcal{R}_r)), \\ \quad (var_1 \ldots var_n), \\ \quad e_b \;\rangle \in_{P'} c_1 \end{array}}{\begin{array}{l} \mathrm{P}'\Gamma \vdash_e \mathbf{super}.md(e_1, \ldots, e_n) \Rightarrow \\ \qquad \mathbf{super}\underline{\equiv \mathbf{this} : c_1}.md(e'_1, \ldots, e'_n) \;:\; ((t_r \;\texttt{at}\; (\rho_r, \mathcal{R}_r))\;!\;\rho) \end{array}} \;[\text{super}]$$

FIGURE 10. Type Rules For Fields and Methods

$$\frac{\begin{array}{l}\mathrm{P}' \vdash_t c \\ \textsc{NoAbstractMethods}(\mathrm{P}',\ c)\end{array}}{\mathrm{P}'\Gamma \vdash_e \textbf{new}\ c \Rightarrow \textbf{new}\ c\ :\ ((c\ \texttt{at}\ (\rho, \mathcal{R}))\ !\ \rho)}\ [\text{new}]$$

$$\frac{\text{where } var \in \mathrm{dom}(\Gamma)}{\mathrm{P}'\Gamma \vdash_e var \Rightarrow var\ :\ (\ \Gamma(var)\ !\ \rho\ )}\ [\text{var}]$$

$$\frac{\mathrm{P}' \vdash_t t}{\mathrm{P}'\Gamma \vdash_e \mathsf{null} \Rightarrow \mathsf{null}\ :\ ((t\ \texttt{at}\ (\rho', \mathcal{R}))\ !\ \rho)}\ [\text{null}]$$

$$\frac{\mathrm{P}'\Gamma \vdash_s e \Rightarrow e'\ :\ ((t\ \texttt{at}\ (\rho', \mathcal{R}))\ !\ \rho)}{\mathrm{P}'\Gamma \vdash_e \textbf{view}\ t\ e \Rightarrow e'\ :\ ((t\ \texttt{at}\ (\rho', \mathcal{R}))\ !\ \rho)}\ [\text{wcast}]$$

$$\frac{\begin{array}{l}\mathrm{P}'\Gamma \vdash_e e \Rightarrow e'\ :\ ((t_2\ \texttt{at}\ (\rho', \mathcal{R}_2))\ !\ \rho) \\ t_1 \leq_{P'} t_2 \text{ or } t_1 \in \mathrm{dom}(\leq^i_{P'}) \text{ or } t_2 \in \mathrm{dom}(\leq^i_{P'})\end{array}}{\mathrm{P}'\Gamma \vdash_e \textbf{view}\ t_1\ e \Rightarrow \textbf{view}\ t_1\ e'\ :\ ((t_1\ \texttt{at}\ (\rho', \mathcal{R}_2))\ !\ \rho)}\ [\text{ncast}]$$

$$\frac{\begin{array}{l}\mathrm{P}'\Gamma \vdash_e e_1 \Rightarrow e_1'\ :\ ((t_1\ \texttt{at}\ (\rho_1, \mathcal{R}_1))\ !\ \rho) \\ \mathrm{P}'\Gamma[var \mapsto (t_1\ \texttt{at}\ (\rho_1, \mathcal{R}_1))] \vdash_e e_2 \Rightarrow e_2'\ :\ ((t_2\ \texttt{at}\ (\rho_2, \mathcal{R}_2))\ !\ \rho)\end{array}}{\mathrm{P}'\Gamma \vdash_e \textbf{let}\ var = e_1\ \textbf{in}\ e_2 \Rightarrow \textbf{let}\ var = e_1'\ \textbf{in}\ e_2'\ :\ ((t_2\ \texttt{at}\ (\rho_2, \mathcal{R}_2))\ !\ \rho)}\ [\text{let}]$$

$$\frac{\begin{array}{l}\mathrm{P}'\Gamma \vdash_e e \Rightarrow e'\ :\ ((t_2\ \texttt{at}\ (\rho', \mathcal{R}_2))\ !\ \rho) \\ t_2 \leq_{P'} t_1\end{array}}{\mathrm{P}'\Gamma \vdash_s e \Rightarrow e'\ :\ ((t_1\ \texttt{at}\ (\rho', \mathcal{R}_2))\ !\ \rho)}\ [\text{sub}]$$

$$\frac{\mathrm{P}' \vdash_t t}{\mathrm{P}'\Gamma \vdash_e \textbf{abstract} \Rightarrow \textbf{abstract}\ :\ ((t\ \texttt{at}\ (\rho', \mathcal{R}))\ !\ \rho)}\ [\text{abs}]$$

$$\frac{\begin{array}{l}\mathrm{P}'\Gamma \vdash_e e \Rightarrow e'\ :\ ((t\ \texttt{at}\ (\rho_1, \mathcal{R}))\ !\ \rho_2) \\ \rho_2 \neq \rho_1 \text{ and } \rho_2 \notin FV(\Gamma)\end{array}}{\mathrm{P}'\Gamma \vdash_e \textbf{encap}\ e \Rightarrow \textbf{encap}\ e'\ :\ ((t\ \texttt{at}\ (\rho_1, \mathcal{R}))\ !\ \rho)}\ [\text{encap}]$$

FIGURE 11. Additional Type Rules For Expressions

This requires the class of the value to match the declared class of the field in the same manner as ClassicJava. However, this new rule also requires three additional things. First, by using the same region variable for both the instance location and the current working region in the first judgement, this new rule requires that the instance whose field is being accessed lie in the current working region. Second, by using the region obtained from the accessed fields entry in $\mathcal{R}$ in the third judgement, this new rule also requires that the new value come from the same region as the existing value for that field. Third, by using the same working region for each judgement, this new rule requires that all of the work takes place in the same region.

It should be noted that none of the rules in the system tries to combine distinct regions into a new region that includes them both. We do not infer any of the region information but rather assume a valid assignment of region names to region variables is provided up front and only proceed to check its correctness.

CHAPTER V

SEMANTICS

Our semantics shares some similarities with ClassicJava as well. Both are specified operationally using reductions that transform program configurations. In both systems a configuration is a pair in which one term represents the expression and the other represents the store. ClassicJava configurations are written $\langle e,\ \mathsf{S}\rangle$ and in RegionJava they are written **sto** $\Delta\ e$. The representation of objects in the store also comes from ClassicJava. Objects are represented as a pair consisting of the class name and a mapping of field names to references, $\langle c, \mathcal{F}\rangle$. The store is a mapping of references to these objects.

However, at this point our semantics diverges significantly from ClassicJava and instead models the semantics of Monadic ML [8]. Like Monadic ML, we add **sto** $\Delta\ e$ to the syntax as a valid expression so that they can be nested. Nesting expressions allows us to cleanly localize store usage even when an encapsulated expression encapsulates one of its subexpressions. The Δ above, unlike S, actually stands for a series of partitions, $\theta_1 \ldots \theta_n$. The leftmost partition is considered the current working region. Partitions to the right of the current working region are not accessable but are maintained to allow dangling pointers. The complete extended syntax is in Figure 12 .

$$
\begin{array}{rcl}
e & = & \ldots \mid \mathsf{ref} \mid \mathbf{sto}\ \Delta\ e \\
\mathsf{object} & = & \langle\ c,\ \mathcal{F}\ \rangle \\
p & = & \mathsf{ref} \mid \mathsf{null} \\
\mathcal{F} & = & \{(c_1.fd_1, p_1), \ldots, (c_n.fd_n, p_n)\} \\
\theta & = & \{(\mathsf{ref}_1, \mathsf{object}_1), \ldots, (\mathsf{ref}_n, \mathsf{object}_n)\} \\
\Delta & = & \theta \mid \theta\Delta
\end{array}
$$

FIGURE 12. Extended Syntax

$$
\frac{\Gamma(\mathsf{ref}) = (t\ \mathtt{at}\ (\rho', \mathcal{R}))}{\mathrm{P}'\Gamma \vdash_e \mathsf{ref} \Rightarrow \mathsf{ref}\ :\ ((t\ \mathtt{at}\ (\rho', \mathcal{R}))\ !\ \rho)}\,[loc]
$$

$$
\frac{\begin{array}{l}
\mathtt{dom}(\mathcal{F}) = \mathtt{dom}(\mathcal{R}) = \{c'.fd \mid \exists t\ \langle\ c'.fd,\ t\ \rangle \in_{P'} c\} \\
\mathcal{R}(c'.fd) = \rho' \text{ and } \mathcal{F}(c'.fd) = p' \text{ implies} \\
\qquad \exists\ t_p, \mathcal{R}_p, \rho_p \text{ s.t.} \\
\qquad \mathrm{P}\Gamma \vdash_{\underline{e}} p'\ :\ ((t_p\ \mathtt{at}\ (\rho', \mathcal{R}_p))\ !\ \rho_p)
\end{array}}{\mathrm{P}'\Gamma \vdash_o \langle\ c,\ \mathcal{F}\ \rangle \bowtie (c\ \mathtt{at}\ (\rho, \mathcal{R}))}\,[obj]
$$

$$
\frac{\begin{array}{l}
\mathrm{P}'\Gamma\ [[\mathsf{ref}_{1j} \mapsto (t_{1j}\ \mathtt{at}\ (\rho_1, \mathcal{R}_{1j}))]_{j=1}^{s_1} \\
\quad \vdots \\
\quad [\mathsf{ref}_{kj} \mapsto (t_{kj}\ \mathtt{at}\ (\rho_k, \mathcal{R}_{kj}))]_{j=1}^{s_k}] \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij}\ \mathtt{at}\ (\rho_i, \mathcal{R}_{ij})) \\
\mathrm{P}'\Gamma\ [[\mathsf{ref}_{1j} \mapsto (t_{1j}\ \mathtt{at}\ (\rho_1, \mathcal{R}_{1j}))]_{j=1}^{s_1} \\
\quad \vdots \\
\quad [\mathsf{ref}_{kj} \mapsto (t_{kj}\ \mathtt{at}\ (\rho_k, \mathcal{R}_{kj}))]_{j=1}^{s_k}] \vdash_e e \Rightarrow e'\ :\ ((t''\ \mathtt{at}\ (\rho'', \mathcal{R}''))\ !\ \rho_1) \\
\rho_i, \rho'' \text{ distinct} \qquad \rho_i \notin FV(\Gamma)
\end{array}}{\begin{array}{ll}
\mathrm{P}'\Gamma \vdash_e & \mathbf{sto}\ \{(\mathsf{ref}_{1j}, \mathsf{object}_{1j})_{j=1}^{s_1}\} \ldots \{(\mathsf{ref}_{kj}, \mathsf{object}_{kj})_{j=1}^{s_k}\}\ e \Rightarrow \\
& \mathbf{sto}\ \{(\mathsf{ref}_{1j}, \mathsf{object}_{1j})_{j=1}^{s_1}\} \ldots \{(\mathsf{ref}_{kj}, \mathsf{object}_{kj})_{j=1}^{s_k}\}\ e'\ :\ ((t''\ \mathtt{at}\ (\rho'', \mathcal{R}''))\ !\ \rho''')
\end{array}}\,[sto]
$$

$$
\text{where } 1 \leq i \leq k,\ \ 1 \leq j \leq s_i,\ \ \text{and } 1 \leq m_{ij} \leq n_{ij}
$$

FIGURE 13. Additional Type Rules

Since the **sto** expressions are added to the syntax, additional type rules are added as well. These are listed in Figure 13 . The checking of references is described with [loc]. In addition, a new judgment type was added to describe the consistency between an object in the store and a type-and-effect description. This is the [obj] rule. This is then used by the [sto] rule to require that objects in the store be consistent with the type of the references to them. The [sto] rule defines how pairs from each partition are used to add relevant pairs to Γ. Each pair added to Γ

has region information that identifies the partition of the referenced object, and then the expression is checked in the new environment and required to do all of its work using only the leftmost partition. There may be many partitions yet there is always only a single partition that is considered active at any one time.

Evaluation proceeds syntactically as a series of term rewriting. The rewrite rules are of the form $\mathrm{P}'\Gamma \vdash e \hookrightarrow e'$, representing one step in the evaluation. Multiple steps are written $e \hookrightarrow\!\!\!\!\rightarrow e'$. The rules are given in Figure 14. Rules are expressed using the evaluation contexts of Figure 15. These determine the order of term rewriting. Since **sto** expressions may be nested, it is critical that an expression use the innermost set of partitions. This is ensured by two separate contexts which are identical except that the A context does not contain the **sto** expression. Thus, the expression **sto** Δ $\mathsf{A}[\mathsf{ref}]$ will allow ref to be contained within a complicated context but will require that, should ref be used to access the instance, ref will be looked up in the given Δ. The other characteristics of the contexts are the standard left to right evaluation and call by value.

The rules (new), (get), and (set) describe how to continue execution with a new expression and a possibly modified store. The (new) rule uses the term $\mathcal{F}^{c}_{init}$ to describe an $\mathcal{F}$ whose domain is the range of $\in^{c}_{P'}$ for the given class, and where all of the values are initialized to null. The rules (call) and (super) describe how to execute method bodies using substitution for the arguments and the **this** pointer. The (let) rule uses substitution as well to handle variables. The (mask) rule allows nested stores to be lifted out of the expression while (encap) creates a new working partition for the given expression that can later be lifted out with (mask). Note

$$
\begin{array}{llll}
(\mathit{new}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{new}\ c]] \hookrightarrow \\
 & & & \mathsf{E}[\mathbf{sto}\ \theta'\Delta\ \mathsf{A}[\mathsf{ref}]] \\
 & & & \text{where}\quad \mathsf{ref} \notin \mathrm{dom}(\theta\Delta),\ \text{and} \\
 & & & \qquad\quad \theta' = \theta \cup \{(\mathsf{ref},\ \langle\ c,\ \mathcal{F}^{c}_{init}\ \rangle)\}
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{get}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd]] \hookrightarrow \\
 & & & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[p]] \\
 & & & \text{where}\quad \theta(\mathsf{ref}) = \langle\ c',\ \mathcal{F}\ \rangle,\ \text{and} \\
 & & & \qquad\quad \mathcal{F}(c.fd) = p
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{set}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd = p_1]] \hookrightarrow \\
 & & & \mathsf{E}[\mathbf{sto}\ \theta'\Delta\ \mathsf{A}[p_1]] \\
 & & & \text{where}\quad \theta(\mathsf{ref}) = \langle\ c',\ \mathcal{F}\ \rangle, \\
 & & & \qquad\quad \mathcal{F}(c.fd) = p_2, \\
 & & & \qquad\quad \theta'(\mathsf{ref}) = \langle\ c',\ \mathcal{F}'\ \rangle,\ \text{and} \\
 & & & \qquad\quad \mathcal{F}'(c.fd) = p_1
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{call}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}.md(p_1,\ldots,p_n)]] \hookrightarrow \\
 & & & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[e_b[\mathsf{ref}/\mathbf{this}, p_1/var_1,\ldots,p_n/var_n]]] \\
 & & & \text{where}\quad \theta(\mathsf{ref}) = \langle\ c,\ \mathcal{F}\ \rangle,\ \text{and} \\
 & & & \qquad\quad \langle\ md, \\
 & & & \qquad\qquad ((c\ \mathtt{at}\ (\rho_0,\mathcal{R}_0))\ldots,(t_n\ \mathtt{at}\ (\rho_n,\mathcal{R}_n)) \xrightarrow{\rho} (t_r\ \mathtt{at}\ (\rho_r,\mathcal{R}_r))), \\
 & & & \qquad\qquad (var_1,\ldots,var_n), \\
 & & & \qquad\qquad e_b\ \rangle \in_{P'} c
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{super}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{super}\underline{\equiv\ \mathsf{ref}\ :\ c'}.md(p_1,\ldots,p_n)]] \hookrightarrow \\
 & & & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[e_b[\mathsf{ref}/\mathbf{this}, p_1/var_1,\ldots,p_n/var_n]]] \\
 & & & \text{where}\quad \theta(\mathsf{ref}) = \langle\ c,\ \mathcal{F}\ \rangle,\ \text{and} \\
 & & & \qquad\quad \langle\ md, \\
 & & & \qquad\qquad ((c'\ \mathtt{at}\ (\rho_0,\mathcal{R}_0))\ldots,(t_n\ \mathtt{at}\ (\rho_n,\mathcal{R}_n)) \xrightarrow{\rho} (t_r\ \mathtt{at}\ (\rho_r,\mathcal{R}_r))), \\
 & & & \qquad\qquad (var_1,\ldots,var_n), \\
 & & & \qquad\qquad e_b\ \rangle \in_{P'} c'
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{view}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{view}\ t'\ \mathsf{ref}]] \hookrightarrow \\
 & & & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}]] \\
 & & & \text{where}\quad c \leq_{P'} t',\ \text{and} \\
 & & & \qquad\quad \theta(\mathsf{ref}) = \langle\ c,\ \mathcal{F}\ \rangle
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{view}^n) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{view}\ t'\ \mathsf{null}] \hookrightarrow \mathsf{E}[\mathsf{null}] \\
(\mathit{let}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{let}\ var = p\ \mathbf{in}\ e] \hookrightarrow \mathsf{E}[e[p/var]] \\
(\mathit{mask}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{sto}\ \Delta\ A[\mathbf{sto}\ \Delta'\ p]] \hookrightarrow \mathsf{E}[\mathbf{sto}\ \Delta\Delta'\ A[p]] \\
(\mathit{encap}) & \mathrm{P}' & \vdash & \mathsf{E}[\mathbf{encap}\ e] \hookrightarrow \mathsf{E}[\mathbf{sto}\ \{\}\ e]
\end{array}
$$

FIGURE 14. Reduction Rules

$$
\begin{array}{lcl}
\mathsf{A} & = & [\,] \mid \mathsf{A}\underline{:c}.fd \mid \mathsf{A}\underline{:c}.fd = e \mid p\underline{:c}.fd = \mathsf{A} \\
& \mid & \mathsf{A}.md(e\ldots) \mid p.md(p\ldots\mathsf{A}\ e\ldots) \\
& \mid & \textbf{super}\underline{\equiv \mathsf{ref} : c}.md(p\ldots\mathsf{A}\ e\ldots) \\
& \mid & \textbf{view}\ t\ \mathsf{A} \mid \textbf{let}\ var = \mathsf{A}\ \textbf{in}\ e
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{E} & = & [\,] \mid \mathsf{E}\underline{:c}.fd \mid \mathsf{E}\underline{:c}.fd = e \mid p\underline{:c}.fd = \mathsf{E} \\
& \mid & \mathsf{E}.md(e\ldots) \mid p.md(p\ldots\mathsf{E}\ e\ldots) \\
& \mid & \textbf{super}\underline{\equiv \mathsf{ref} : c}.md(p\ldots\mathsf{E}\ e\ldots) \\
& \mid & \textbf{view}\ t\ \mathsf{E} \mid \textbf{let}\ var = \mathsf{E}\ \textbf{in}\ e \\
& \mid & \textbf{sto}\ \Delta\ \mathsf{E}
\end{array}
$$

FIGURE 15. Evaluation Contexts

$$
\begin{array}{llll}
(view^x) & \mathrm{P}' & \vdash & \mathsf{E}[\textbf{sto}\ \theta\Delta\ \mathsf{A}[\textbf{view}\ t'\ \mathsf{ref}]] \hookrightarrow \text{error: bad cast} \\
& & & \text{where } \theta(\mathsf{ref}) = \langle\ c,\ \mathcal{F}\ \rangle \text{ and } c \not\leq_P t' \\
(get^x) & \mathrm{P}' & \vdash & \mathsf{E}[\textbf{sto}\ \Delta\ \mathsf{null}\underline{:c}.fd] \hookrightarrow \text{error: null pointer} \\
(set^x) & \mathrm{P}' & \vdash & \mathsf{E}[\textbf{sto}\ \Delta\ \mathsf{null}\underline{:c}.fd = p] \hookrightarrow \text{error: null pointer} \\
(call^x) & \mathrm{P}' & \vdash & \mathsf{E}[\textbf{sto}\ \Delta\ \mathsf{null}.md(p_1, \ldots, p_n)] \hookrightarrow \text{error: null pointer}
\end{array}
$$

FIGURE 16. Reductions to Errors

that these lifted partitions are added to the right of the list of partitions and are thus not accessible to work with, but are retained to accommodate dangling pointers. We have also added a (view^n) rule which was not present in ClassicJava to allow null references to be cast to any type.

Some expressions lead to errors but are not caught during static analysis. These runtime errors are the result of specific data values that cannot be determined without running the program. For example, the type of an expression may be a class that is appropriate for a specific field or method access. But if its value at run time is null then there is no instance to access and an error has occurred. This situation results in one of the error configurations listed in Figure 16.

CHAPTER VI

TYPE SOUNDNESS

Our examination of the soundness of the system follows the standard syntactic approach [12]. Namely it depends on three steps. First, we must show that every program is an answer, faulty, or corresponds to a rewrite rule. Second, we must show that the process of rewriting preserves the type. Third, we must show that faulty programs are untypable. Once each of these has been determined then typable programs must either diverge, written $\mathrm{P}'\Gamma \vdash e\Uparrow$, produce an answer of the right type, or produce and *error configuration.*

Like the soundness proof of ClassicJava[3] we create judgements that each correspond to an existing judgement, performing all the same checks, except they operate on already annotated terms and do no further elaboration. For example,

$$\mathrm{P}'\Gamma \vdash_e e \Rightarrow e' \ : \ ((t \ \mathtt{at} \ (\rho_1, \mathcal{R})) \ ! \ \rho_2)$$

will correspond to

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} e \ : \ ((t \ \mathtt{at} \ (\rho_1, \mathcal{R})) \ ! \ \rho_2)$$

The only deviation from a complete correspondence is that $\vdash_{\underline{e}}$ judgements ignore [wcast] rules for simplicity since it is only an optimization.

We start with some basic definitions.

Definition 1 (Programs)

A program is a series of class and interface definitions and a closed expression.

The initial expression, e, given by the programmer is written **sto** $\{\}$ e to provide an initial working region for the program.

Definition 2 (Error Configuration)

An *error configuration* is any one of $(view^x)$, (get^x), (set^x), and $(call^x)$.

The *error configurations* are runtime errors and are not targeted by the type system. They are therefore considered distinct from the faulty expressions.

Definition 3 (Answers)

An answer is a term of the form **sto** Δ p.

The faulty expressions correspond to stuck states where the semantics is undefined.

Definition 4 (Faulty Expressions)

The following expression are faulty:

$\mathbf{sto}\ \theta\Delta\ \mathsf{ref}$ where $\mathsf{ref} \in \theta$

$\mathbf{sto}\ \theta\Delta\ \mathsf{ref}\underline{:\ c'}.fd$

where $\theta(\mathsf{ref}) = \langle\ c,\ \mathcal{F}\ \rangle$, and

$\langle\ c'.fd,\ t\ \rangle \notin_{P'} c$, or where

$\theta(\mathsf{ref})$ is undefined

$\mathbf{sto}\ \theta\Delta\ \mathsf{ref}\underline{:\ c'}.fd = p$

where $\theta(\mathsf{ref}) = \langle\ c,\ \mathcal{F}\ \rangle$, , and

$\langle\ c'.fd,\ t\ \rangle \notin_{P'} c$, or where

$\theta(\mathsf{ref})$ is undefined

$\mathbf{sto}\ \theta\Delta\ \mathsf{ref}.md(p_1, \ldots, p_m)$

where $\theta(\mathsf{ref}) = \langle\ t_0,\ \mathcal{F}\ \rangle$, and either

$\langle\ md,$

$((t_0\ \mathtt{at}\ (\rho_0, \mathcal{R}_0)), \ldots, (t_n\ \mathtt{at}\ (\rho_n, \mathcal{R}_n)) \xrightarrow{\rho_b} (t_r\ \mathtt{at}\ (\rho_r, \mathcal{R}_r))),$

$(var_1, \ldots, var_n),$

$e_b\ \rangle \notin_{P'} c,$ or

$m \neq n$, or where

$\theta(\mathsf{ref})$ is undefined

<u>Definition 5</u> (Redex)

A Redex is any expression, e, where there exists an e' such that $\mathrm{P}' \vdash e \hookrightarrow e'$.

We first want to show how expressions can be broken down.

Proposition 1 (Decomposition)

Every term e can be decomposed into one of the following forms:

p,

$\mathsf{A}[\mathbf{new}\ c]$, $\mathsf{A}[p\underline{:\ c.fd}]$, $\mathsf{A}[p\underline{:\ c}.fd = p']$, $\mathsf{A}[p.md(p_1 \ldots p_n)]$,

$\mathsf{A}[\mathbf{super}\underline{\equiv \mathbf{this} : c}.md(p_1 \ldots, p_n)]$, $\mathsf{A}[\mathbf{view}\ t\ p]$, or $\mathsf{A}[\mathbf{sto}\ \Delta\ p]$.

$\mathsf{E}[var]$, E[Redex], E[Faulty], an *error configuration*,

This can be shown by induction on the structure of terms.

Then we can use this information to state the following corollary about closed expressions.

Corollary 1

A closed expression of the form $\mathbf{sto}\ \Delta\ e$ can be decomposed into one of the following forms:

$\mathbf{sto}\ \Delta\ p$, which is an answer,

E[Redex],

E[Faulty], or

an *error configuration*.

This can be shown by induction on the structure of terms.

Now we have our four desired categories. It remains to be shown that those in the redex category are typable and reduce to answers of the given type, while those in the faulty category are not typable.

The following propositions are necessary to our subject reduction proposition.

First, we want to be able to add to or remove from the environment when doing so has no effect on the expression.

Proposition 2

If $\mathrm{P}'\ \Gamma \vdash_{\underline{e}} e\ :\ ((t\ \mathtt{at}\ (\rho_1, \mathcal{R}))\ !\ \rho_2)$ and $\Gamma(x) = \Gamma'(x)$ for all free x in e then $\mathrm{P}'\ \Gamma' \vdash_{\underline{e}} e\ :\ ((t\ \mathtt{at}\ (\rho_1, \mathcal{R}))\ !\ \rho_2)$.

This follows from the structure of derivations.

We also want to show that the hole in a context can be filled with any expression of the same type without changing the type of the overall context.

Proposition 3 (Replacement)

Let $\mathsf{C} = \mathsf{A}$ or E

If $\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \mathsf{C}[e]\ :\ ((t_1\ \mathtt{at}\ (\rho_1, \mathcal{R}_1))\ !\ \rho_2)$,

$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} e\ :\ ((t_2\ \mathtt{at}\ (\rho_3, \mathcal{R}_2))\ !\ \rho_4)$, and

$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} e'\ :\ ((t_2\ \mathtt{at}\ (\rho_3, \mathcal{R}_2))\ !\ \rho_4)$,

then $\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \mathsf{C}[e']\ :\ ((t_1\ \mathtt{at}\ (\rho_1, \mathcal{R}_1))\ !\ \rho_2)$.

This is a direct result of the fact that holes in either context correspond directly to antecedent rules. If e and e' both satisfy the antecedent then the consequent can be rebuilt using either.

Then we want to be able to replace variables in an expression with their corresponding values. However, we need to account for the fact that a variable may be assigned a value that is a subtypes of the variable's type.

Proposition 4 (Substitution)

Let $\sigma = \{(var_1, p_1), \ldots, (var_n, p_n)\}$

$\overline{\sigma} = [var_i \mapsto (t_i \texttt{ at } (\rho_i, \mathcal{R}_i))]_{i=1}^{n}$

If $\texttt{dom}(\Gamma) \cap \texttt{dom}(\overline{\sigma}) = \emptyset$,

$\mathrm{P}'\Gamma\overline{\sigma} \vdash_{\underline{e}} e \ : \ ((t \texttt{ at } (\rho', \mathcal{R})) \ ! \ \rho)$, and

$\mathrm{P}'\Gamma \vdash_{\underline{s}} p_j \ : \ ((t_j \texttt{ at } (\rho_j, \mathcal{R}_j)) \ ! \ \rho)$ for $j \in [1, n]$,

then $\mathrm{P}'\Gamma \vdash_{\underline{s}} \sigma(e) \ : \ ((t \texttt{ at } (\rho', \mathcal{R})) \ ! \ \rho)$

This follows by induction on the structure of e.

Since substitution leaves us with subtyping judgements, it is useful to have a replacement rule that addresses them directly.

Proposition 5 (Replacement with Subtyping)

Let $\mathsf{C} = \mathsf{A}$ or E

If $\mathrm{P}' \ \Gamma \vdash_{\underline{e}} \mathsf{C}[e] \ : \ ((t_1 \texttt{ at } (\rho_1, \mathcal{R}_1)) \ ! \ \rho)$,

$\mathrm{P}' \ \Gamma \vdash_{\underline{e}} e \ : \ ((t_2 \texttt{ at } (\rho_2, \mathcal{R}_2)) \ ! \ \rho)$, and

$\mathrm{P}' \ \Gamma \vdash_{\underline{e}} e' \ : \ ((t_3 \texttt{ at } (\rho_2, \mathcal{R}_2)) \ ! \ \rho)$, where

$t_3 \leq^c_{P'} t_2$,

then $\mathrm{P}' \ \Gamma \vdash_{\underline{s}} \mathsf{C}[e'] \ : \ ((t_1 \texttt{ at } (\rho_1, \mathcal{R}_1)) \ ! \ \rho)$.

This follows by induction on the depth of the evaluation contexts:

Since faulty expressions are not typable, we know that if a context which is filled with an expression is typable, that expression must also be typable. In addition, from the structure of our type system we also know that working regions are forced to match. This information is expressed in the following proposition.

Proposition 6 (Region Flow)

Let $\quad \mathsf{C} = \mathsf{A}$ or E

If $\quad \mathrm{P}'\ \Gamma \vdash \mathsf{C}[e]\ :\ ((t_1\ \mathtt{at}\ (\rho_1, \mathcal{R}_1))\ !\ \rho)$

then $\quad \mathrm{P}'\ \Gamma \vdash e\ :\ ((t_2\ \mathtt{at}\ \rho_2, \mathcal{R}_2))\ !\ \rho)$

This is a direct result of the fact that each context's hole participates directly in a antecedent of the type rule that requires the regions match.

Now we are ready to consider the subject reduction proposition. This proposition ensures that each rewrite preserves the overall type of the expression. Note that we consider it enough if the resulting type is a subtype of the original type.

Proposition 7 (Subject Reduction)

If $\quad \mathrm{P}'\ \Gamma \vdash_{\underline{e}} e\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'')$, and

$\quad \mathrm{P}'\ \vdash e \hookrightarrow e'$

then $\quad \mathrm{P}'\ \Gamma \vdash_{\underline{s}} e'\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'')$.

This follows from a case analysis on the reduction rules (see Appendix).

We now address faulty expressions.

Proposition 8 (Faulty Expressions are Untypable)

If e is faulty there is no P', Γ, t, ρ', and ρ such that $\mathrm{P}'\Gamma \vdash_{\underline{e}} e \ : \ ((t \ \texttt{at} \ (\rho', \mathcal{R})) \ ! \ \rho)$.

This follows from a case analysis of the definition of faulty expressions.

Finally, from Propositions 7 and 8 we have the following corollary:

Proposition 9 (Syntactic Soundness)

If $\quad P'\Gamma \vdash_{\underline{e}} \mathbf{sto} \ \Delta \ e \ : \ ((t \ \texttt{at} \ (\rho', \mathcal{R})) \ ! \ \rho)$

then $\quad \mathrm{P}' \vdash \mathbf{sto} \ \Delta \ e \Uparrow$,

$\mathrm{P}' \vdash \mathbf{sto} \ \Delta \ e \hookrightarrow\!\!\!\!\rightarrow$ an *error configuration*, or

$\mathrm{P}' \vdash \mathbf{sto} \ \Delta \ e \hookrightarrow\!\!\!\!\rightarrow \mathbf{sto} \ \Delta \ p$ where

$\mathrm{P}'\Gamma \vdash_{\underline{s}} \mathbf{sto} \ \Delta \ p \ : \ ((t \ \texttt{at} \ (\rho', \mathcal{R})) \ ! \ \rho)$.

This expresses the desired result that if an expression is typable, it must either diverge, reduce to an *error configuration*, or reduce to a value, if it is not one already, that has the same type as the expression.

CHAPTER VII

CONCLUSION AND FURTHER WORK

We have conducted an investigation into a type-and-effect system for encapsulating memory in Java. Our system looks at merging the syntax of ClassicJava with the semantics of Monadic ML. Our simple use of region names for the effects proved to be quite constraining. Our attempts to overcome those limitations by adding more effect variables to each single type caused a level of complexity that proved difficult to manage. In addition, they only provided a strange hybrid of functionality. Nonetheless, many of the issues of trying to encapsulate memory in an object oriented language are well illustrated. It is a complex task to manage region information in a language with inheritance, where types are frequently subsumed to other types, and where values are references to a collection of references. Much work remains to be done to resolve these issues more cleanly. In addition, the ability to infer types rather than check a type assignment is another major barrier that must be addressed before any such system that encapsulates memory can be implemented.

APPENDIX

SUBJECT REDUCTION

$$
\begin{array}{ll}
\text{If} & \mathrm{P}'\ \Gamma \vdash_{\underline{e}} e \ : \ ((t'' \ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2''), \text{and} \\
 & \mathrm{P}'\ \vdash e \hookrightarrow e' \\
\text{then} & \mathrm{P}'\ \Gamma \vdash_{\underline{e}} e' \ : \ ((t'' \ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'').
\end{array}
$$

By case analysis on the reduction rules.

Let:

$$
\begin{array}{rcl}
|\theta| & = & s, \\
\theta & = & \{(\mathsf{ref}_1, \mathsf{object}_1), \ldots, (\mathsf{ref}_s, \mathsf{object}_s)\}, \\
\overline{\theta} & = & [\mathsf{ref}_j \mapsto (t_j \ \mathtt{at}\ (\rho, \mathcal{R}_j))]_{j=1}^{s},
\end{array}
$$

$$
\begin{array}{rcll}
|\theta_i| & = & s_i, & 1 \leq i \leq k, \\
\theta_i & = & \{(\mathsf{ref}_{i1}, \mathsf{object}_{i1}), \ldots, (\mathsf{ref}_{is_i}, \mathsf{object}_{is_i})\}, & 1 \leq i \leq k, \\
\overline{\theta}_i & = & [\mathsf{ref}_{ij} \mapsto (t_{ij} \ \mathtt{at}\ (\rho_i, \mathcal{R}_{ij}))]_{j=1}^{s_i}, & 1 \leq i \leq k,
\end{array}
$$

$$
\begin{array}{rcl}
\Delta & = & \theta_1, \ldots, \theta_k, \\
\overline{\Delta} & = & \overline{\theta}_1 \ldots \overline{\theta}_k.
\end{array}
$$

$$
\begin{array}{rcll}
|\theta'_i| & = & s'_i, & 1 \le i \le k', \\
\theta'_i & = & \{(\mathsf{ref}'_{i1}, \mathsf{object}'_{i1}), \ldots, (\mathsf{ref}'_{is_i}, \mathsf{object}'_{is'_i})\}, & 1 \le i \le k', \\
\overline{\theta}'_i & = & [\mathsf{ref}'_{ij} \mapsto (t'_{ij} \texttt{ at } (\rho'_i, \mathcal{R}_{ij}))]_{j=1}^{s'_i}, & 1 \le i \le k',
\end{array}
$$

$$
\begin{array}{rcl}
\Delta' & = & \theta'_1, \ldots, \theta'_{k'}, \text{ and} \\
\overline{\Delta}' & = & \overline{\theta}'_1 \ldots \overline{\theta}'_k.
\end{array}
$$

Case $[new]$:

By assumption:

$$
\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{new}\ c]] \ :\ ((t'' \texttt{ at } (\rho''_1, \mathcal{R}''))\ !\ \rho''_2) \tag{A.1}
$$

Taking first the contents of E, by Proposition 6 we have

$$
\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{new}\ c] \ :\ ((t''' \texttt{ at } (\rho'''_1, \mathcal{R}'''))\ !\ \rho''_2) \tag{A.2}
$$

from (A.2) by [sto] we have

$$
\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_j \bowtie (t_j \texttt{ at } (\rho, \mathcal{R}_j)) \tag{A.3}
$$

$$
\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij} \texttt{ at } (\rho_j, \mathcal{R}_{ij})) \tag{A.4}
$$

$$
\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathbf{new}\ c] \ :\ ((t''' \texttt{ at } (\rho'''_1, \mathcal{R}'''))\ !\ \rho) \tag{A.5}
$$

$$
\rho, \rho_i, \rho'''_1 \text{ distinct} \tag{A.6}
$$

$$
\rho, \rho_i \notin FV(\Gamma) \tag{A.7}
$$

and from (A.5), [new], and using Proposition 6 again, we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \textbf{new } c \ : \ ((c \ \texttt{at} \ (\rho, \mathcal{R})) \ ! \ \rho) \tag{A.8}$$

Let ref be a new location not occurring in $\mathrm{dom}(\overline{\theta\Delta})$ then by Proposition 2 and (A.8)

$$\mathrm{P}'\Gamma\overline{\theta}[\mathsf{ref} \mapsto (c \ \texttt{at} \ (\rho, \mathcal{R}))] \ \overline{\Delta} \vdash_{\underline{e}} \textbf{new } c \ : \ ((c \ \texttt{at} \ (\rho, \mathcal{R})) \ ! \ \rho) \tag{A.9}$$

By (A.9), and [loc] we have

$$\mathrm{P}'\Gamma\overline{\theta}[\mathsf{ref} \mapsto (c \ \texttt{at} \ (\rho, \mathcal{R}))] \ \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref} \ : \ ((c \ \texttt{at} \ (\rho, \mathcal{R})) \ ! \ \rho) \tag{A.10}$$

By Proposition 2, (A.5), and (A.10) we have

$$\mathrm{P}'\Gamma\overline{\theta}[\mathsf{ref} \mapsto (c \ \texttt{at} \ (\rho, \mathcal{R}))] \ \overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[\textbf{new } c] \ : \ ((t''' \ \texttt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho) \tag{A.11}$$

and by Proposition 3, (A.9), (A.10), and (A.11)

$$\mathrm{P}'\Gamma\overline{\theta}[\mathsf{ref} \mapsto (c \ \texttt{at} \ (\rho, \mathcal{R}))] \ \overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathsf{ref}] \ : \ ((t''' \ \texttt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho) \tag{A.12}$$

By (A.12) and the definition of $\mathcal{F}^c_{init}$ we have

$$\begin{aligned} &\mathrm{P}'\Gamma\overline{\theta}[\mathsf{ref} \mapsto (c\ \mathtt{at}\ (\rho, \mathcal{R}))]\ \overline{\Delta} \vdash_o \\ &\quad \langle\ c,\ \mathcal{F}^c_{init}\ \rangle \bowtie (c\ \mathtt{at}\ (\rho, \mathcal{R})) \end{aligned} \tag{A.13}$$

Then by [sto], (A.3), (A.4), (A.6), (A.7), (A.12), and (A.13) we have

$$\begin{aligned} \mathrm{P}'\Gamma \vdash_{\underline{e}}\ & \mathbf{sto}\ \theta \cup \{(\mathsf{ref}, \langle\ c,\ \mathcal{F}^c_{init}\ \rangle)\}\ \Delta\ \mathsf{A}[\mathsf{ref}] \\ &:\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \end{aligned} \tag{A.14}$$

And finally using Proposition 3, (A.1), (A.2), and (A.14) we have

$$\begin{aligned} \mathrm{P}'\Gamma \vdash_{\underline{e}}\ & \mathsf{E}[\mathbf{sto}\ \theta \cup \{(\mathsf{ref}, \langle\ c,\ \mathcal{F}^c_{init}\ \rangle)\}\ \Delta\ \mathsf{A}[\mathsf{ref}]] \\ &:\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \end{aligned} \tag{A.15}$$

Case $[get]$:

By assumption:

$$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd]]\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.16}$$

Taking first the contents of E, using Proposition 6 we have

$$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd]\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.17}$$

from which by [sto]

$$\mathrm{P}'\Gamma\overline{\theta}\overline{\Delta} \vdash_o \mathsf{object}_j \bowtie (t_j \; \mathtt{at} \; (\rho, \mathcal{R}_j)) \tag{A.18}$$

$$\mathrm{P}'\Gamma\overline{\theta}\overline{\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij} \; \mathtt{at} \; (\rho_j, \mathcal{R}_{ij})) \tag{A.19}$$

$$\mathrm{P}'\Gamma\overline{\theta}\overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd] \; : \; ((t''' \; \mathtt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho) \tag{A.20}$$

$$\rho, \rho_i, \rho_1''' \text{ distinct} \tag{A.21}$$

$$\rho, \rho_i \notin FV(\Gamma) \tag{A.22}$$

From (A.20), and Proposition 6 we have

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref}\underline{:\ c}.fd \; : \; ((t''_{fd} \; \mathtt{at} \; (\rho''_{fd}, \mathcal{R}''_{fd})) \; ! \; \rho) \tag{A.23}$$

By (A.23) and [get] we have

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref} \; : \; ((c_o \; \mathtt{at} \; (\rho, \mathcal{R}_o)) \; ! \; \rho) \tag{A.24}$$

$$\langle \; c.fd, \; t''_{fd} \; \rangle \in_{P'} c_o \tag{A.25}$$

By (A.18), (A.23), and (A.24)

$$\mathsf{ref} = \langle \; c_o, \; \mathcal{F} \; \rangle \tag{A.26}$$

$$\mathcal{F}(c.fd) = p \tag{A.27}$$

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} p \; : \; ((t''_{fd} \; \mathtt{at} \; (\rho''_{fd}, \mathcal{R}''_{fd})) \; ! \; \rho) \tag{A.28}$$

By Proposition 3, (A.20), (A.23) and (A.28)

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[p] \;\; : \;\; ((t''' \; \texttt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho) \tag{A.29}$$

By (A.18), (A.19), (A.21), (A.22), (A.29), and [sto] we have

$$\mathrm{P}' \; \Gamma \vdash_{\underline{e}} \textbf{sto} \; \theta\Delta \; \mathsf{A}[p] \;\; : \;\; ((t''' \; \texttt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho_2'') \tag{A.30}$$

Then using Proposition 3, (A.16), (A.17), and (A.30) we can conclude

$$\mathrm{P}' \; \Gamma \vdash_{\underline{e}} \mathsf{E}[\textbf{sto} \; \theta\Delta \; \mathsf{A}[p]] \;\; : \;\; ((t'' \; \texttt{at} \; (\rho_1'', \mathcal{R}'')) \; ! \; \rho_2'') \tag{A.31}$$

Case $[set]$:

By assumption:

$$\mathrm{P}' \; \Gamma \vdash_{\underline{e}} \mathsf{E}[\textbf{sto} \; \theta\Delta \; \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd = p_1]] \;\; : \;\; ((t'' \; \texttt{at} \; (\rho_1'', \mathcal{R}'')) \; ! \; \rho_2'') \tag{A.32}$$

Taking first the contents of E, using Proposition 6 we have

$$\mathrm{P}' \; \Gamma \vdash_{\underline{e}} \textbf{sto} \; \theta\Delta \; \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd = p_1] \;\; : \;\; ((t''' \; \texttt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho_2'') \tag{A.33}$$

By (A.33) and [sto] we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_j \bowtie (t_j \;\texttt{at}\; (\rho, \mathcal{R}_j)) \tag{A.34}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij} \;\texttt{at}\; (\rho_j, \mathcal{R}_{ij})) \tag{A.35}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathsf{ref}\underline{:\ c}.fd = p_1] \;:\; ((t''' \;\texttt{at}\; (\rho_1''', \mathcal{R}''')) \;!\; \rho) \tag{A.36}$$

$$\rho, \rho_i, \rho_1''' \text{ distinct} \tag{A.37}$$

$$\rho, \rho_i \notin FV(\Gamma) \tag{A.38}$$

From (A.36), and Proposition 6

$$\mathrm{P}'\ \Gamma\ \overline{\theta}\ \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref}\underline{:\ c}.fd = p_1 \;:\; ((t''_{fd} \;\texttt{at}\; \mathcal{R}''_o(c.fd)) \;!\; \rho) \tag{A.39}$$

By (A.39) and [set]

$$\mathrm{P}'\ \Gamma\ \overline{\theta}\ \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref} \;:\; ((c''_o \;\texttt{at}\; (\rho, \mathcal{R}''_o)) \;!\; \rho) \tag{A.40}$$

$$\langle\ c.fd,\ t''_{fd}\ \rangle \in_{P'} c''_o \tag{A.41}$$

$$\mathrm{P}'\ \Gamma\ \overline{\theta}\ \overline{\Delta} \vdash_{\underline{e}} p_1 \;:\; ((t''_{fd} \;\texttt{at}\; \mathcal{R}''_o(c.fd)) \;!\; \rho) \tag{A.42}$$

By Proposition 3, (A.36), (A.39) and (A.42) we have

$$\mathrm{P}'\ \Gamma\ \overline{\theta}\ \overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[p_1] \;:\; ((t''' \;\texttt{at}\; (\rho_1''', \mathcal{R}''')) \;!\; \rho) \tag{A.43}$$

By (A.34), (A.39), and (A.40) we have

$$\mathsf{ref} = \langle\ c''_o,\ \mathcal{F}''_o\ \rangle \tag{A.44}$$

$$\mathcal{F}''_o(c.fd) = p_2 \tag{A.45}$$

$$\mathrm{P}'\ \Gamma\ \overline{\theta}\ \overline{\Delta} \vdash_{\underline{e}} p_2\ :\ ((t''_{fd}\ \texttt{at}\ \mathcal{R}''_o(c.fd))\ !\ \rho) \tag{A.46}$$

Since p_1 and p_2 have the same type by (A.42), and (A.46) replacing p_2 with p_1 in $\mathcal{F}''_o$ will maintain consistency of $\mathcal{F}''_o$ and $\mathcal{R}''_o$. Therefore by (A.34), (A.35), (A.37), (A.38), (A.43), and [sto] we have

$$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \textbf{sto}\ \theta\Delta\ \mathsf{A}[p_1]\ :\ ((t'''\ \texttt{at}\ (\rho'''_1, \mathcal{R}'''))\ !\ \rho''_2) \tag{A.47}$$

Then using Proposition 3, (A.32), (A.33), and (A.47) we can conclude

$$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \mathsf{E}[\textbf{sto}\ \theta\Delta\ \mathsf{A}[p_1]]\ :\ ((t''\ \texttt{at}\ (\rho''_1, \mathcal{R}''))\ !\ \rho''_2) \tag{A.48}$$

Case [*call*] :

By assumption

$$\begin{aligned}\mathrm{P}'\ \Gamma \vdash_{\underline{e}}\ & \mathsf{E}[\textbf{sto}\ \theta\Delta\ \mathsf{A}[\mathsf{ref}.md(p_1, \ldots, p_n)]] \\ & :\ ((t''\ \texttt{at}\ (\rho''_1, \mathcal{R}''))\ !\ \rho''_2)\end{aligned} \tag{A.49}$$

Taking first the contents of E, using Proposition 6 we have

$$\begin{aligned} \mathrm{P}' \; \Gamma \vdash_{\underline{e}} \; & \textbf{sto } \theta\Delta \; \mathsf{A}[\mathsf{ref}.md(p_1, \ldots, p_n)] \\ & : \; ((t''' \; \mathtt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho_2'') \end{aligned} \tag{A.50}$$

From (A.50) using [sto] we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_j \bowtie (t_j \; \mathtt{at} \; (\rho, \mathcal{R}_j)) \tag{A.51}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij} \; \mathtt{at} \; (\rho_j, \mathcal{R}_{ij})) \tag{A.52}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathsf{ref}.md(p_1, \ldots, p_n)] \; : \; ((t''' \; \mathtt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho) \tag{A.53}$$

$$\rho, \rho_i, \rho_1''' \text{ distinct} \tag{A.54}$$

$$\rho, \rho_i \notin FV(\Gamma) \tag{A.55}$$

and from (A.53), and Proposition 6

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref}.md(p_1, \ldots, p_n) \; : \; ((t_r'''' \; \mathtt{at} \; (\rho_r'''', \mathcal{R}_r'''')) \; ! \; \rho) \tag{A.56}$$

Then using (A.56) and [call] we can say

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} \mathsf{ref} \; : \; ((t_0'''' \; \mathtt{at} \; (\rho_0'''', \mathcal{R}_0'''')) \; ! \; \rho) \tag{A.57}$$

$$\begin{aligned} &\langle \; md, \\ &\quad ((t_0'''' \; \mathtt{at} \; (\rho_0'''', \mathcal{R}_0'''')), \ldots, (t_n'''' \; \mathtt{at} \; (\rho_n'''', \mathcal{R}_n'''')) \xrightarrow{\rho} (t_r'''' \; \mathtt{at} \; (\rho_r'''', \mathcal{R}_r''''))) \\ &\quad (var_1, \ldots, var_n) \\ &\quad e_b\rangle \in_{P'} t_0'''' \end{aligned} \tag{A.58}$$

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} p_j \;:\; ((t_j'''' \; \texttt{at} \; (\rho_j'''', \mathcal{R}_j'''')) \; ! \; \rho) \text{ for } j \in [1, n] \tag{A.59}$$

And by (A.58) and [meth]

$$\begin{aligned} \mathrm{P}' \; [\; & \mathbf{this} \mapsto (t_0'''' \; \texttt{at} \; (\rho_0'''', \mathcal{R}_0'''')), \\ & var_1 \mapsto (t_1'''' \; \texttt{at} \; (\rho_1'''', \mathcal{R}_1'''')), \\ & \ldots \\ & var_n \mapsto (t_n \; \texttt{at} \; (\rho_n'''', \mathcal{R}_n''''))] \vdash_s e_b \;:\; ((t''' \; \texttt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho) \end{aligned} \tag{A.60}$$

Let

$$e' = e_b[\mathsf{ref}/\mathbf{this}, p_1/var_1, \ldots, p_n/var_n]$$

Then by (A.60), Proposition 2, and Proposition 4

$$\mathrm{P}' \; \vdash_s e' \;:\; ((t_r'''' \; \texttt{at} \; (\rho_r'''', \mathcal{R}_r'''')) \; ! \; \rho) \tag{A.61}$$

By Proposition 5, (A.53), (A.56) and (A.61)

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{s}} \mathsf{A}[e'] \;:\; ((t''' \; \texttt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho) \tag{A.62}$$

Then there exists a t_s''' such that $t_s''' \leq_{P'} t'''$ and

$$\mathrm{P}' \; \Gamma \; \overline{\theta} \; \overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[e'] \;:\; ((t_s''' \; \texttt{at} \; (\rho_1''', \mathcal{R}''')) \; ! \; \rho) \tag{A.63}$$

Then by (A.51), (A.52), (A.54), (A.55), (A.63), and [sto] we have

$$\mathrm{P}'\ \Gamma \vdash_{\underline{e}} \mathbf{sto}\ \theta\Delta\ \mathsf{A}[e']\ :\ ((t_s''' \ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.64}$$

Since by definition $t_s''' \leq_{P'} t'''$ we can also say

$$\mathrm{P}'\ \Gamma \vdash_{\underline{s}} \mathbf{sto}\ \theta\Delta\ \mathsf{A}[e']\ :\ ((t''' \ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho) \tag{A.65}$$

Then using Proposition 5, (A.49), (A.50), and (A.65) we can conclude

$$\mathrm{P}'\ \Gamma \vdash_{\underline{s}} \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[e']]\ :\ ((t'' \ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.66}$$

Case [*super*] :

By assumption

$$\begin{aligned}\mathrm{P}'\Gamma \vdash_{\underline{e}}\ \ & \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{super}\underline{\equiv p_0 : c}.md(p_1, \ldots, p_n)]] \\ & :\ ((t'' \ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'')\end{aligned} \tag{A.67}$$

Taking first the contents of E, using Proposition 6 we have

$$\begin{aligned}\mathrm{P}'\Gamma \vdash_{\underline{e}}\ \ & \mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{super}\underline{\equiv p_0 : c}.md(p_1, \ldots, p_n)] \\ & :\ ((t''' \ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'')\end{aligned} \tag{A.68}$$

From (A.68) by [sto] we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_j \bowtie (t_j \ \mathtt{at}\ (\rho, \mathcal{R}_j)) \tag{A.69}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij} \ \mathtt{at}\ (\rho_j, \mathcal{R}_{ij})) \tag{A.70}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathbf{super}\underline{\equiv p_0 : c}.md(p_1, \ldots, p_n)] \tag{A.71}$$

$$: \ ((t''' \ \mathtt{at}\ (\rho_1''', \mathcal{R}''')) \ ! \ \rho)$$

$$\rho, \rho_i, \rho_1''' \text{ distinct} \tag{A.72}$$

$$\rho, \rho_i \notin FV(\Gamma) \tag{A.73}$$

From (A.71), and Proposition 6 we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \ \mathbf{super}\underline{\equiv p_0 : c}.md(p_1, \ldots, p_n) \tag{A.74}$$

$$: \ ((t_r'''' \ \mathtt{at}\ (\rho_r'''', \mathcal{R}_r'''')) \ ! \ \rho)$$

By (A.74) and [super] we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} p_0 \ : \ ((t_0'''' \ \mathtt{at}\ (\rho_0'''', \mathcal{R}_0'''')) \ ! \ \rho) \tag{A.75}$$

$$t_0'''' \prec^c_{P'} t_s'''' \tag{A.76}$$

$$\langle \ md, \tag{A.77}$$

$$((t_s'''' \ \mathtt{at}\ (\rho_0'''', \mathcal{R}_0'''')), \ldots, (t_n'''' \ \mathtt{at}\ (\rho_n'''', \mathcal{R}_n'''')) \xrightarrow{\rho} (t_r'''' \ \mathtt{at}\ (\rho_r'''', \mathcal{R}_r'''')))$$

$$(var_1, \ldots, var_n)$$

$$e_b\rangle \in_{P'} t_s''''$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} p_j \ : \ ((t_j'''' \ \mathtt{at}\ (\rho_j'''', \mathcal{R}_j'''')) \ ! \ \rho) \text{ for } j \in [1, n] \tag{A.78}$$

$$e_b \neq \mathbf{abstract} \tag{A.79}$$

And by (A.77) and [meth]

$$
\begin{aligned}
\mathrm{P}' \ [\ & \mathbf{this} \mapsto (t_s'''' \ \mathtt{at} \ (\rho_0'''', \mathcal{R}_0'''')), \\
& var_1 \mapsto (t_1'''' \ \mathtt{at} \ (\rho_1'''', \mathcal{R}_1'''')), \\
& \ldots \\
& var_n \mapsto (t_n'''' \ \mathtt{at} \ (\rho_n'''', \mathcal{R}_n''''))] \vdash_s e_b \ : \ ((t''' \ \mathtt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho)
\end{aligned} \tag{A.80}
$$

Let

$$e' = e_b[p_0/\mathbf{this}, p_1/var_1, \ldots, p_n/var_n]$$

Then by (A.80), Proposition 4, and Proposition 2 we have

$$\mathrm{P}' \ \vdash_{\underline{s}} e' \ : \ ((t_r'''' \ \mathtt{at} \ (\rho_r'''', \mathcal{R}_r'''')) \ ! \ \rho) \tag{A.81}$$

By Proposition 5, (A.71), (A.74) and (A.81)

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{s}} \mathsf{A}[e'] \ : \ ((t''' \ \mathtt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho) \tag{A.82}$$

Then there exists a t_s''' such that $t_s''' \leq_{P'} t'''$ and

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \mathsf{A}[e'] \ : \ ((t_b''' \ \mathtt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho) \tag{A.83}$$

Then by (A.69), (A.70), (A.72), (A.73), (A.83), and [sto] we have

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathbf{sto} \ \theta\Delta \ \mathsf{A}[e'] \ : \ ((t_b''' \ \mathtt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho_2'') \tag{A.84}$$

Since by definition $t_s''' \leq_{P'} t'''$ we can also say

$$\mathrm{P}'\Gamma \vdash_{\underline{s}} \mathbf{sto}\ \theta\Delta\ \mathsf{A}[e']\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.85}$$

Then using Proposition 5, (A.67), (A.68), and (A.85) we can conclude

$$\mathrm{P}'\Gamma \vdash_{\underline{s}} \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[e']]\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.86}$$

Case $[view]$:

By assumption:

$$\mathrm{P}'\Gamma \vdash \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{view}\ t_1\ p]]\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.87}$$

Taking first the contents of E, using Proposition 6 we have

$$\mathrm{P}'\Gamma \vdash \mathbf{sto}\ \theta\Delta\ \mathsf{A}[\mathbf{view}\ t_1\ p]\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.88}$$

from which by [sto]

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_j \bowtie (t_j\ \mathtt{at}\ (\rho, \mathcal{R}_j)) \tag{A.89}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij}\ \mathtt{at}\ (\rho_j, \mathcal{R}_{ij})) \tag{A.90}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathbf{view}\ t_1\ p]\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho) \tag{A.91}$$

$$\rho, \rho_i, \rho_1'''\ \text{distinct} \tag{A.92}$$

$$\rho, \rho_i \notin FV(\Gamma) \tag{A.93}$$

and from (A.91), and Proposition 6

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash \textbf{view}\ t_1\ p\ :\ ((t_1'''' \ \texttt{at}\ (\rho_1'''', \mathcal{R}_1'''')) \ !\ \rho) \tag{A.94}$$

By (A.94) and [view] we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash p\ :\ ((t_2'''' \ \texttt{at}\ (\rho_1'''', \mathcal{R}_1'''')) \ !\ \rho) \tag{A.95}$$

$$t_2'''' \leq^c_{P'} t_1'''' \text{ or } t_1'''' \in \mathrm{dom}(\prec\!\!\!\prec^c_{P'}) \text{ or } t_2'''' \in \mathrm{dom}(\prec\!\!\!\prec^c_{P'}) \tag{A.96}$$

By (A.95), (A.96), and [sub]

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash p\ :\ ((t_1'''' \ \texttt{at}\ (\rho_1'''', \mathcal{R}_1'''')) \ !\ \rho) \tag{A.97}$$

By Proposition 3, (A.91), (A.94) and (A.97)

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash \mathsf{A}[p]\ :\ ((t''' \ \texttt{at}\ (\rho_1''', \mathcal{R}''')) \ !\ \rho_2'') \tag{A.98}$$

and by (A.89), (A.90), (A.92), (A.93), (A.98), and [sto]

$$\mathrm{P}'\Gamma \vdash \textbf{sto}\ \theta\Delta\ \mathsf{A}[p]\ :\ ((t''' \ \texttt{at}\ (\rho_1''', \mathcal{R}''')) \ !\ \rho_2'') \tag{A.99}$$

Then using Proposition 3, (A.87), (A.88), and (A.99) we can conclude

$$\mathrm{P}'\Gamma \vdash \mathsf{E}[\mathbf{sto}\ \theta\Delta\ \mathsf{A}[p]]\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.100}$$

Case $[view^n]$:

By assumption:

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{view}\ t\ \mathsf{null}]\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.101}$$

Using (A.101) with Proposition 6 we have

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathbf{view}\ t\ \mathsf{null}\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.102}$$

Then by (A.102) and [null] we have

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{null}\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.103}$$

By Proposition 3, (A.101), (A.102) and (A.103)

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathsf{null}]\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.104}$$

Case $[let]$:

By assumption:

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{E}[\textbf{let}\ var = p\ \textbf{in}\ e]\ :\ ((t''\ \texttt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.105}$$

Using (A.105) with Proposition 6 we have

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \textbf{let}\ var = p\ \textbf{in}\ e\ :\ ((t'''\ \texttt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.106}$$

Then by (A.106) and [let] we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{e}} p\ :\ ((t''''\ \texttt{at}\ (\rho_1'''', \mathcal{R}''''))\ !\ \rho_2'') \tag{A.107}$$

$$\mathrm{P}'\Gamma\overline{\theta\Delta}[var \mapsto (t''''\ \texttt{at}\ (\rho_1'''', \mathcal{R}''''))] \vdash_{\underline{e}} \quad e\ :\ ((t'''\ \texttt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.108}$$

Then using Proposition 4 with (A.107), and (A.108) we have

$$\mathrm{P}'\Gamma\overline{\theta\Delta} \vdash_{\underline{s}} e[p/var]\ :\ ((t'''\ \texttt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.109}$$

By Proposition 5, (A.105), (A.106) and (A.109)

$$\mathrm{P}'\Gamma \vdash_{\underline{s}} \mathsf{E}[e[p/var]]\ :\ ((t''\ \texttt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.110}$$

Case $[mask]$:

By assumption:

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{sto}\ \Delta\ \mathsf{A}[\mathbf{sto}\ \Delta'\ p]]\ :\ ((t''\ \mathtt{at}\ (\rho_1'', \mathcal{R}''))\ !\ \rho_2'') \tag{A.111}$$

Looking first at the contexts of E, using Proposition 6 we have

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathbf{sto}\ \Delta\ \mathsf{A}[\mathbf{sto}\ \Delta'\ p]\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_2'') \tag{A.112}$$

From (A.112) and [sto] we have

$$\mathrm{P}'\Gamma\overline{\Delta} \vdash_o \mathsf{object}_{ij} \bowtie (t_{ij}\ \mathtt{at}\ (\rho_j, \mathcal{R}_{ij})) \tag{A.113}$$

$$\mathrm{P}'\Gamma\overline{\Delta} \vdash_{\underline{e}} \mathsf{A}[\mathbf{sto}\ \Delta'\ p]\ :\ ((t'''\ \mathtt{at}\ (\rho_1''', \mathcal{R}'''))\ !\ \rho_1) \tag{A.114}$$

$$\rho_i, \rho_1'''\ \text{distinct} \tag{A.115}$$

$$\rho_i \notin FV(\Gamma) \tag{A.116}$$

By (A.114) and Proposition 6 we have

$$\mathrm{P}'\Gamma\overline{\Delta} \vdash_{\underline{e}} \mathbf{sto}\ \Delta'\ p\ :\ ((t''''\ \mathtt{at}\ (\rho_1'''', \mathcal{R}''''))\ !\ \rho_1) \tag{A.117}$$

Using $[sto]$ again with (A.117) we have

$$\mathrm{P}'\Gamma\overline{\Delta\Delta}' \vdash_o \mathsf{object}'_{ij} \bowtie (t'_{ij}\ \mathtt{at}\ (\rho'_j, \mathcal{R}'_{ij})) \tag{A.118}$$

$$\mathrm{P}'\Gamma\overline{\Delta\Delta}' \vdash_{\underline{e}} p\ :\ ((t''''\ \mathtt{at}\ (\rho_1'''', \mathcal{R}''''))\ !\ \rho'_1) \tag{A.119}$$

$$\rho'_i, \rho_1''''\ \text{distinct} \tag{A.120}$$

$$\rho'_i \notin FV(\Gamma\overline{\Delta}) \tag{A.121}$$

Then since p is a value, from (A.119) we can also say

$$\mathrm{P}'\Gamma\overline{\Delta\Delta'} \vdash_{\underline{e}} p \;:\; ((t'''' \;\texttt{at}\; (\rho_1'''', \mathcal{R}'''')) \;!\; \rho_1) \tag{A.122}$$

Now we can use Proposition 3 and (A.114), (A.117), and (A.122) to say

$$\mathrm{P}'\Gamma\overline{\Delta\Delta'} \vdash_{\underline{e}} \mathsf{A}[p] \;:\; ((t''' \;\texttt{at}\; (\rho_1''', \mathcal{R}''')) \;!\; \rho_1') \tag{A.123}$$

Now we can use the [sto] rule with (A.113), (A.115), (A.116), (A.118), (A.120), (A.121), and (A.123) to say

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathbf{sto}\; \Delta\Delta'\; \mathsf{A}[p] \;:\; ((t''' \;\texttt{at}\; (\rho_1''', \mathcal{R}''')) \;!\; \rho_2'') \tag{A.124}$$

Finally, using (A.111), (A.112), (A.124), and Proposition 3 we have

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{sto}\; \Delta\Delta'\; \mathsf{A}[p]] \;:\; ((t'' \;\texttt{at}\; (\rho_1'', \mathcal{R}'')) \;!\; \rho_2'') \tag{A.125}$$

Case [*encap*].

By assumption:

$$\mathrm{P}'\Gamma \vdash \mathsf{E}[\mathbf{encap}\; e] \;:\; ((t'' \;\texttt{at}\; (\rho_1'', \mathcal{R}'') \;!\; \rho_2'') \tag{A.126}$$

By (A.126) and Proposition 6 we have

$$\mathrm{P}'\Gamma \vdash \mathbf{encap}\; e \;:\; ((t''' \;\texttt{at}\; (\rho_1''', \mathcal{R}''') \;!\; \rho_2'') \tag{A.127}$$

Then by (A.127), and (encap)

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} e \ : \ ((t''' \ \mathtt{at} \ (\rho_1''', \mathcal{R}''')) \ ! \ \rho_2'') \tag{A.128}$$

$$\rho_2'' \neq \rho_1''' \tag{A.129}$$

$$\rho_2'' \notin FV(\Gamma) \tag{A.130}$$

Then using (sto), since the rules regarding references and objects pass trivially and the remaining ones result from (A.128), (A.129), and (A.130), we can say

$$\mathrm{P}'\Gamma \vdash_{\underline{e}} \mathbf{sto} \ \{\} \ e \ : \ ((t''' \ \mathtt{at} \ (\rho_1''', \mathcal{R}''') \ ! \ \rho_2'') \tag{A.131}$$

Then by (A.126), (A.127), (A.130), and Proposition 3 we have

$$\mathrm{P}'\Gamma \vdash \mathsf{E}[\mathbf{sto} \ \{\} \ e] \ : \ ((t'' \ \mathtt{at} \ (\rho_1'', \mathcal{R}'') \ ! \ \rho_2'') \tag{A.132}$$

BIBLIOGRAPHY

[1] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer-Verlag, New York, N.Y., June 1997.

[2] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992.

[3] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report COMP TR 97-293, Department of Computer Science, Rice University, Houston, Texas, April 1997.

[4] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, N.Y., January 1998. ACM.

[5] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[6] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[7] Jon M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages.* ACM, ACM Press, January 1988.

[8] Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. Soon to appear in ACM SIGPLAN International Conference on Functional Programming, 1999.

[9] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In A. Scedrov, editor, *Proceedings of the 1992 Logics in Computer Science Conference*, pages 162–173. IEEE, 1992.

[10] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value [lambda]-calculus using a stack of regions. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*, pages 188–201, New York, NY, USA, 1994. ACM Press.

[11] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

[12] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer Science, Rice University, Houston, Texas, April 1991.