2012-11-04

Javalite - An Operational Semantics for Modeling Java Programs

Saint Oming'o Wesonga
Brigham Young University - Provo

Javalite - An Operational Semantics for Modeling Java Programs

Saint O. Wesonga

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric Mercer, Chair
Jay McCarthy
Mark Clement

Department of Computer Science

Brigham Young University

November 2012

ABSTRACT

Javalite - An Operational Semantics for Modeling Java Programs

Saint O. Wesonga
Department of Computer Science, BYU
Master of Science

Java is currently a widely used programming language. However, there is no formal definition of Java's semantics. Consequently, Java code does not have a universal meaning. This work discusses recent attempts to formalize Java and presents a new formalism of Java called Javalite. In contrast to common approaches to formalization, Javalite is purely syntactic in its definition. Syntactic operational semantics use the structure of the language to define its behavior. Javalite models most Java features with notable exceptions being threads, reflection, and interfaces. This work presents an executable semi-formal model of Javalite in PLT Redex. Being executable means that Javalite programs can be run using this model. We then render the semi-formal model in the Coq theorem prover and present theorems stating that the operational semantics are decidable and deterministic. This formal model can then be used to facilitate research in areas such as proving properties of algorithms that perform various analyses on Java code, e.g. verification, optimization, and refactoring.

Keywords: Javalite, Operational Semantics, PLT Redex, Coq

ACKNOWLEDGMENTS

I would like to thank Eric Mercer and Jay McCarthy for their assistance in co-authoring this paper. I would specifically like to thank Dr. Mercer for his hours of work and invaluable advice, and Dr. McCarthy for excellent advice on and help with PLT Redex tools and the Coq theorem prover.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Java™ is a widely adopted object-oriented programming language and has multiple implementations, both open and closed source, by various vendors. It is currently defined in the continuously evolving document referred to as the Java Language Specification (Gosling et al. [2005]). This document is the standard that details the syntax and semantics of the Java language. Unfortunately, it is a prose-only definition of Java's semantics and therefore allows for ambiguity, contradiction, and incompleteness in the specification. These undesirable characteristics affect language and compiler designers as well as Java programmers. An example of the impact of these shortcomings is illustrated by Chan *et al* who present sample programs for which different Java compilers produced different results and list suggestions for Java programmers to navigate these hurdles (Chan et al. [2004]). Formal semantics are one way of resolving such ambiguity in programming languages.

In order to rigorously define the meaning of programs in any programming language, a formal semantics of that language must be defined. There are various approaches to formally defining the meaning of programs. Some common ones include axiomatic semantics, denotational semantics, and operational semantics. Axiomatic semantics is a technique based on logical deduction and assertions about relationships that remain the same each time a given program executes (Slonneger and Kurtz [1995]) whereas the key idea behind denotational semantics is the direct mapping of a program to its meaning (or denotation), which is usually a mathematical object such as a number, a function, or a tuple (Schmidt [1986]).

Operational semantics, on the other hand, is an approach to specifying the meaning of programs by describing how computations are performed i.e., how a program transitions from one configuration to another (Plotkin [2004]). Unlike axiomatic or denotational semantics, operational semantics are typically runnable. They are generally categorized into *small step* and *big step* operational semantics. The difference between small and big-step operational semantics is illustrated by the example in Fig. 1.1, which compares small and big-step evaluation semantics for the arithmetic expression $(1 + 1) + (1 + 1)$. Small step operational semantics formally define how the individual steps of a program are interpreted. The inference tree in

$$\frac{\dfrac{\dfrac{(1+1)+(1+1)}{2+(1+1)}}{2+2}}{4}$$

(a) Small-step evaluation semantics

$$\frac{\dfrac{1 \Downarrow 1 \quad 1 \Downarrow 1}{(1+1) \Downarrow 2} \quad \dfrac{1 \Downarrow 1 \quad 1 \Downarrow 1}{(1+1) \Downarrow 2}}{(1+1)+(1+1) \Downarrow 4}$$

(b) Big-step evaluation semantics

Figure 1.1: Comparison of small vs big-step evaluation of an arithmetic expression.

Fig. 1.1a illustrates the small step approach to evaluating the given expression. The parenthesized sub-expressions are each evaluated step by step until the value of the whole expression is determined.

On the other hand, big step operational semantics formally define what evaluation should yield rather than exactly how evaluation should be done (Hennessy). An example of big-step evaluation of the arithmetic expression $(1+1)+(1+1)$ is shown in Fig. 1.1b. It demonstrates the evaluation logic with two main rules, the first of which is the axiom that an integer should evaluate to itself ($n \Downarrow n$). The second rule requires that the sum of two expressions E_1 and E_2 be equal to the sum of the integers to which each of the expressions evaluate (say n_1 and n_2 respectively). In other words, $(E_1 \Downarrow n_1 \wedge E_2 \Downarrow n_2) \rightarrow (E_1 + E_2) \Downarrow (n_1 + n_2)$. The use of such logic by big-step operational semantics leads to the determination of the result of a computation without necessarily executing each individual step as is the case with small-step operational semantics.

Formalizing the semantics of programming languages using such approaches not only enables a thorough and consistent description of such languages, but also enables rigorous research and reasoning about the properties of the languages and their models. However, modeling the Java language and all its supported features is a non-trivial undertaking. Different models include or exclude different features based on the research objectives of their authors. Our research objective is to define formal model checking algorithms for a formal model of the Java language and then prove properties about these algorithms. One such algorithm is the slicing and dicing partial order reduction scheduler (Rungta and Mercer [2010]). It searches for error traces in concurrent programs by scheduling threads using criteria that attempt to minimize the number of interleavings that must be checked to reveal an error condition. We intend to show that this algorithm is sound and complete. In this context, soundness means that the error traces it generates are valid traces for the concurrent program whereas completeness means that the algorithm will always find an error trace if there does exist one for a given concurrent program.

In order to create formal models of such algorithms and prove properties about them, we need formal semantics of the Java language. These semantics need to be minimal enough to enable formal reasoning yet concise enough to enable their results to apply directly to the Java language. Our thesis is that it is possible to create an executable formalization of a Java-like language, and to prove its semantics deterministic and

```
public class Swap {

    public boolean boolFalse;
    public boolean boolTrue;

    public Swap() {
        boolFalse = false;
        boolTrue = true;
    }

    public void swap() {
        boolean tmp = boolTrue;
        boolTrue = boolFalse;
        boolFalse = tmp;
    }
}
```

(a) A Java "Swap" Class Declaration

```
(class Swap extends Object
  ( [bool bFalse]
    [bool bTrue])

  ((Swap construct () (begin
        (this $ bFalse := false)
        (this $ bTrue := true)
        this))

   (unit swap () (begin
        (var bool tmp := (this $ bTrue) in
             (begin
               (this $ bTrue :=
                   (this $ bFalse))
               (this $ bFalse := tmp)
               ))
        unit))))
```

(b) A Javalite "Swap" Class Declaration

Figure 1.2: Comparison of Java and Javalite class declarations.

automatically generate an interpreter for it. Consequently, we create Javalite (an operational semantics of the Java language) to facilitate rapid prototyping of model checking algorithms and proofs about them. Javalite supports the following Java features: classes, fields, and methods, inheritance, typecasts, dynamic dispatch, field hiding, local variables, and program state. Javalite is based on the CEKS machine (Felleisen et al. [1987]) and defines a Java-like syntax in which programs are written as well as a set of reduction rules for syntactically executing Javalite programs. A comparison of Java and Javalite's syntax is shown is Fig. 1.2. Having created the Javalite semantics, we can then modify them to facilitate researching the model checking algorithms we are interested in. For the partial order reduction scheduler, our approach will be to augment the Javalite semantics with concurrency related primitives and semantics (such as threads and schedulers respectively) and then show that the partial order reduction scheduler's semantics are sound and complete i.e., show that any possible Javalite trace has a corresponding trace in the augmented Javalite partial order reduction semantics.

We create the Javalite semantics using the following approach. First, we rapidly prototype Javalite's semantics in PLT Redex. PLT Redex is a tool that simplifies the process of creating prototypes of models by supporting semantics engineering tasks in domain-specific notations (Felleisen et al. [2009]). A key advantage of this prototyping process is that it is simpler than creating a complete model in a programming language like Java. The semi-formal PLT Redex Javalite model can then be used to prototype model checking algorithms, e.g. a new symbolic execution algorithm that operates by redefining Javalite's language features (such as the new operator, field references, and so on) is currently in the works.

Although the Javalite Redex models represent the functionality of Java well, the PLT Redex environment is not designed for proving properties about models. Such formal proof environments are typically

more involving but rigorous. Therefore, we translate the PLT Redex semantics of both Javalite and the augmented Javalite (which models the algorithms of interest) into equivalent models in Coq. Coq is a formal proof management system that provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs (The Coq development team [2010]). With the Coq formal model, research can now be performed on a variety of algorithms based on the Java language including optimization algorithms, code transformation algorithms, and static analysis algorithms. We will use the formal Coq Javalite model to prove the soundness and completeness of the aforementioned partial order reduction scheduler.

The scope of this thesis is the definition of the deterministic sequential Javalite semantics, the translation of these semantics into the Coq theoreom prover, and the writing of the proof that the semantics are indeed decidable and deterministic. The rest of this thesis is organized as follows: Chapter 2 reviews related Java models. Chapter 3 discusses the PLT Redex Javalite model. Chapter 4 then presents the Coq Javalite model as well as some of the properties that we have proved about Javalite using the Coq theorem prover. It also discusses the implications of these theorems. Chapter 5 summarizes the approaches we used and the significance of the models presented then discusses possible avenues of future work.

Chapter 2

Related Work

Different models of Java-like languages have been created to reason about various features of the Java langauge. Usually, complex features are excluded from these mini-languages to simplify (or even enable) the task of reasoning about language features and semantics. Some of these models of Java are shown in Table 2.1, which compares the sets of features supported by each of the models.

Table 2.1: Comparison of Language Features Supported by Java Models

Feature	Featherweight Java	Lightweight Java	Welterweight Java	Classic Java	Javalite
Classes	✓	✓	✓	✓	✓
Fields	✓	✓	✓	✓	✓
Methods	✓	✓	✓	✓	✓
Inheritance	✓	✓	✓	✓	✓
Typecasts	✓	-	✓	✓	✓
Dynamic Dispatch	✓	✓	✓	✓	✓
Field Hiding	✓	-	✓	✓	✓
Side Effects/State	-	✓	✓	✓	✓
Executable	-	-	-	-	✓
Local Variables	-	-	✓	✓	✓
Interfaces	-	-	-	✓	-
Concurrency	-	-	✓	-	-
Exceptions	-	-	-	-	-
Reflection	-	-	-	-	-

Featherweight Java is billed by its authors as a minimal core calculus for modeling Java's type system. It omits almost all features of Java including interfaces and (more critically) assignment to enable straightforward yet rigorous proofs (Igarashi et al. [2001]). However, it still provides classes, methods, fields, inheritance, and dynamic typecasts. There is also an extension to it that supports generics (Featherweight Generic Java). Featherweight Java can be viewed as a purely functional subset of Java. Consequently, it does not need to model the heap and the order of evaluation of expressions does not affect programs. Featherweight Java therefore uses a non-deterministic small-step reduction relation. Its authors present a proof of type safety for Featherweight Java. A lot of work on language models for researching various programming language features is based on FeatherWeight Java. Igarashi *et al*, for instance, present *union types* for statically typed class-based object-oriented languages in order to enhance the flexibility of subtyping (Igarashi and Nagira [2006]). They formalize their idea on top of Featherweight Java in order to prove their type system sound.

Although functional calculi like Featherweight Java are more convenient for studying various properties, they are not suitable for certain types of studies. It has been shown in the study of type systems, for example, that traditional treatment of existential types is sound for Featherweight Generic Java but unsound for Java itself (Summers [2009]). More accurate models of Java therefore require the concept of program state. In addition to allowing for possibly unsound scenarios (in relation to the full Java language), functional core calculi cannot (by their nature) be used for the analysis of language features dependent on state. Separation logic, for example, is an extension of Hoare logic that simplifies reasoning about low-level imperative programs that use shared mutable data structures (Reynolds [2002]). The imperative nature of the programs being reasoned about naturally demands that program state be a part of the model. Functional calculi are therefore not directly applicable to separation logic. State is an essential component of the Javalite model we present in this paper. The Javalite model therefore captures a larger subset of behaviors possible in Java programs than the functional calculi. Unfortunately, modelling state also results in more effort expended in maintaining the rigorousness of any proofs about the model.

There are imperative models of the Java language that also account for state, for example, Lightweight Java (Strniša et al. [2007]). It supports the standard object-oriented features such as classes, fields, methods, inheritance, and dynamic method dispatch. Since it serves as the base langauge for the formalization of module systems, it excludes support for local variables, field hiding, interfaces, and method overloading since these are orthogonal to the module system (Strniša et al. [2007]). We consider many of these features to be important for our goal of studying model checking algorithms because supporting them vastly increases the variety of programs that can be reasoned about. A comparison of the features supported by Javalite, Lightweight Java, and other models of the Java language is shown in Table 2.1. Unlike Lightweight Java (and the other semantics), a key feature of Javalite is that its semantics are executable and it has a formally verified implementation.

Welterweight Java is another semantics that attempts to create an alternative to FeatherWeight Java by modelling imperativeness and program state (Östlund and Wrigstad [2010]). It is similar to LightWeight Java but also models Java's thread-based concurrency and lock-based synchronization. It is noteworthy that Javalite does not support these concurrency and synchronization features. Also noteworthy is the fact that Welterweight Java's stated objective is to serve as a completely minimal core calculus. It intentionally excludes expressions such as if-statements, equality tests, and instance-of expressions. In addition to this, Welterweight Java's semantics are not executable. Javalite, on the other hand supports the aforementioned expressions since it will serve as the base language for model checking algorithms that deal with language features such as branching. As mentioned earlier, Javalite's semantics are also executable, which enables rapid prototyping of extensions to the model.

ClassicJava (Flatt et al. [1999]) is a small subset of sequential Java with syntax and semantics modeled using Scheme and ML rewriting techniques. It supports interfaces, which Javalite does not because Javalite's objective is to provide a semantics that captures the overall semantics of Java while still remaining minimal enough to simplify reasoning about it. Moreover, interfaces are mainly related to the type system (which Javalite does not support) and do not have significant runtime behavior (other than through reflection, which neither ClassicJava nor Javalite support). Javalite's lack of interface support means that the process of converting classes that implement multiple interfaces to Javalite would be more tedious. Nonetheless, Javalite supports inheritance and its semantics can be readily extended to support interfaces. Something the two semantics have in common is that their evaluation rules are defined in terms of individual expressions. It is noteworthy, however, that Javalite does not yet have a type-checker. Unlike ClassicJava, however, we present a formal proof (in the Coq theorem prover) of the decidability and determinism of Javalite. We use reduction relations for modeling evaluation of Javalite programs in Coq in a fashion similar to the operational semantics of the simplified language, "Imp", in the *Software Foundations* tutorial for the Coq theorem prover (Pierce et al. [2012]).

Another imperative class-based language is used in the design of a type system for Java-like languages that enforces the *exception-safety strong guarantee* (Lagorio and Servetto [2010]). The idea is that exceptions should be thrown without causing visible side effects to the caller. The minimalistic language used by Lagorio and Servetto lacks inheritance and casts, but naturally supports exception handling. For their purposes, inheritance and casting are uninteresting for their model. However, our Javalite model supports these two features because they are an essential part of most Java programs, which therefore minimizes the distance between our model and the actual Java language while still enabling our objective of formal reasoning about the model and algorithms based on it. We currently do not model exceptions as part of this work but they can be readily added to our formalism.

Defining semantics is also necessary in the verification of concurrent software. As an example, Flanagan *et al* describe how thread-modular and procedure-modular reasoning can be combined to verify concurrent programs (Flanagan et al. [2005]). In this work, the authors define a language called *Plato* to avoid the complexity of reasoning about programs in complex languages like Java and use it in their discussion of their verification methodology. They use a tool that parses Java programs into abstract syntax trees which are then translated into an intermediate language that can express their Plato syntax since verification conditions are handled by external tools. As mentioned earlier, our research objective is to enable rigorous reasoning about model checking algorithms such as symbolic execution and partial order reduction algorithms. We therefore created the Javalite model with executable semantics, which none of

the other models possess. We hope to extend our Javalite model with concurrency to support partial order reduction algorithms and proofs.

Another type of analysis that is performed on programs is pointer analysis. Smaragdakis *et al* introduce the concept of full-object-senstitive analysis as a candidate for replacing previous object-sensitive analyses (Smaragdakis et al. [2011]). In order to evaluate these analyses, the authors formalize them using an abstract interpretation over FeatherWeight Java in A-Normal Form (ANF). ANF was introduced by Flanagan et al. in 1993 and requires that all arguments to functions be trivial, meaning that the arguments are reduced to variable lookups for values. ANF is common for language compilation and is a standard transformation to most languages. It is typically called single-static assignment form. Smaragdakis et al. use ANF to simplify the language semantics for points-to analysis. Their base language is therefore imperative. The formal Coq model of Javalite that we present also uses ANF in order to simplify its method invocation expression and therefore simplify proofs about the model. Smaragdakis *et al* use a small-step state machine. Javalite is also imperative and also uses a small-step state machine. Consequently, Javalite would be a suitable language for the formalization of pointer analysis performed by Smaragdakis *et al* because in addition to being imperative, it also uses a heap whose mappings from locations to objects provides sufficient context information for their analysis.

Chapter 3

PLT Redex Model

PLT Redex is a tool that supports semantics engineering tasks in domain-specific notations and also includes a suite of tools embedded into PLT Racket (Felleisen et al. [2009]). It enables the creation of semi-formal but runnable semantics. One of the key advantages of PLT Redex is that it accelerates the prototyping process by enabling low effort creation, testing, and debugging of models. It is then easier to discover potential issues with a model before attempting to prove properties about it (Klein et al. [2012]). Similarly, we first present a semiformal model of Javalite in PLT Redex and then formalize that model in Coq. Our model of the Java language is called Javalite. This chapter presents the semi-formal PLT Redex Javalite model.

Javalite is an imperative model of Java and its feature set relative to Java is summarized in Table 2.1 in Chapter 2. The most notable omissions from the Java language are concurrency, interfaces, exceptions, and integers. We have left the implementation of these features for future work. The semi-formal PLT Redex implementation of Javalite is composed of three main parts: its surface syntax, its machine syntax, and its operational semantics. Its operational semantics are defined using the structure of the language only (i.e. its syntax). Javalite is therefore considered a syntactic machine. A syntactic machine is comparable to a computer because it has states and deterministic instructions that take it from one state to another. However, it also differs from a computer because it uses programs as states and reductions as instructions (Felleisen et al. [2009]). A reducible expression in a program is called a redex and the text surrounding a redex is called its *context*. The reduction rules of rewriting systems (such as the Javalite syntactic machine) determine which of a program's redexes should be reduced to some term. We now present the main components of Javalite in the next sections.

3.1 Javalite Surface Syntax

Javalite programs and expressions are written in a syntax specified by the grammar in Fig. 3.1. Ellipses in a production of a PLT Redex grammar represent a possibly empty sequence of instances of the symbol immediately preceding them. The notation Y ..., for example, represents zero or more instances of the

symbol Y. The production rules of grammar 3.1 correspond to the various features supported by Javalite such as classes, fields, methods, and expressions. The production $M \rightarrow$ *(T m ([T x] ...) e)*, for example, declares that a method M is defined by a return type T, a method identifier m, a list of argument declarations (each of which is defined by a type T and some formal argument name x), and a method body e.

Javalite's primitive types are *unit* (which is equivalent to Java's *void* type) and *bool*, which has the values *true* and *false*. Javalite does not include the notion of integers or floating point numbers. However, it is still Turing complete. The non-primitive type of Javalite is the class type C. Its value is a pointer. Javalite objects are stored in a heap and are accessed through references as is done in Java. These references are the *pointer* values in Javalite. These pointer values are typed, meaning that they specify both the location of an object on the heap as well as the type of the object. Pointers are defined by the rule *pointer ::= (addr loc C)* | *null* where *loc* is a numeric heap location and C is the class identifier specifying the type cast of the object. Javalite also supports the following statements which have the usual semantics.

x : A variable reference which can also be the reserved keyword *this*.

v : A value which can be *unit*, a boolean, or a typed pointer as discussed previously.

(new C) : The new operator allocates space on the heap for the fields of the class C being instantiated then writes the default values of each of the fields into their corresponding locations on the heap.

(e $ f) : A field access operation where e evaluates to a pointer to an object.

(e @ m (e ...)) : An invocation of the method m on the object referenced by the pointer obtained by evaluating the first expression e. The parenthesized list of expressions are the arguments of the invocation.

(e == e) : The equality operator compares any two expressions for equality and evaluates to a boolean value.

(C e) : The cast operator whose argument e evaluates to a pointer which is then cast to type C.

(e instanceof C) : The instance-of operator determines whether expression e evaluates to a pointer to an instance of class C.

(x := e) : The assignment to identifier x of the value that expression e evaluates to.

(x $ f := e) : Field assignment (to a field f of some object identified by x) of the value to which expression e evaluates.

(if e e else e) : A branching operation whose predicate is the first expression and whose targets are the subsequent expressions.

```
P ::= (μ (C m))
μ ::= (CL ...)
T ::= bool
    | unit
    | C
CL ::= (class C extends C ([T f] ...) (M ...))
M ::= (T m ([T x] ...) e)
e ::= x
    | v
    | (new C)
    | (e $ f)
    | (e @ m (e ...))
    | (e == e)
    | (C e)
    | (e instanceof C)
    | (x := e)
    | (x $ f := e)
    | (if e e else e)
    | (var T x := e in e)
    | (begin e ...)
x ::= this
    | id
f ::= id
m ::= id
C ::= Object
    | id
id ::= variable-not-otherwise-mentioned
pointer ::= (addr loc C)
    | null
v ::= pointer
    | true
    | false
    | unit
    | error
loc ::= number
```

Figure 3.1: The Javalite surface syntax as defined in PLT Redex.

(var T x := e_1 in e_2) : The variable declaration statement represents the evaluation of an expression e_1 whose value is then assigned to a newly created variable with identifier x. The scope of the variable is the expression e_2 and this expression is evaluated next.

(begin e ...) : A list of expressions to be evaluated in left-to-right order.

Note that Javalite does not support explicit looping constructs. Iteration is therefore only possible using recursion.

As an illustration of the Javalite syntax, a sample syntax-highlighted Javalite class declaration (CL) is shown in Fig. 3.2. The example presented is of the class named *Swap* (presented in Chapter 1) that extends the base class *Object*. The keyword *Object* is a reserved word in Javalite. As per the $C := Object$ |

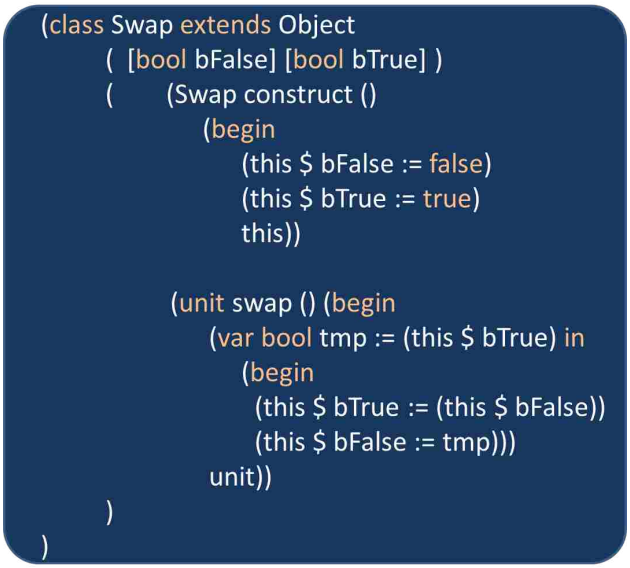

Figure 3.2: A simple Javalite program.

id production in Fig. 3.1, Javalite class identifiers can be either the keyword *Object* or a Javalite identifier. The production for identifiers (*id*) uses the special PLT Redex keyword *variable-not-otherwise-mentioned* to indicate that an identifier can be any symbol not used as a terminal in the grammar. The declaration of the *Swap* class is an instantiation of the class declaration production *CL ::= (class C extends C ([T f] ...) (M ...))*. This production represents the fact that some class CL has two class identifiers (its own as well as its parent's), a list of fields f of types T, and a list of method declarations defined according to the syntax specified by the non-terminal M. The *Swap* class has two fields (*bFalse* and *bTrue*), a method that performs field initialization (*construct*), and a *swap* method.

Other than the surface syntax, there are some important differences between Java and Javalite's semantics. First, Javalite does not have an explicit return statement. Instead, the last subexpression in a method's body expression is considered the return value. In the *Swap* program in Fig. 3.2, for example, the *construct* method's last expression is the object *this*, which is then the return value of the method. In addition to this, the *swap* method has a return type of *unit*. This means that the expression *unit* is the only valid return value and must be the last subexpression of the method body. This differs from Java's approach, which entirely excludes a return expression for *void* return types.

Unlike Java, Javalite does not have the concept of constructors. Consequently Javalite's *new* operator does not invoke any method. It only initializes the new object's locations on the heap with their default values. This is in contrast to Java's *new* operator which invokes the class constructor of the newly created object in addition to initializing its fields to their default values. When Java programs are translated into

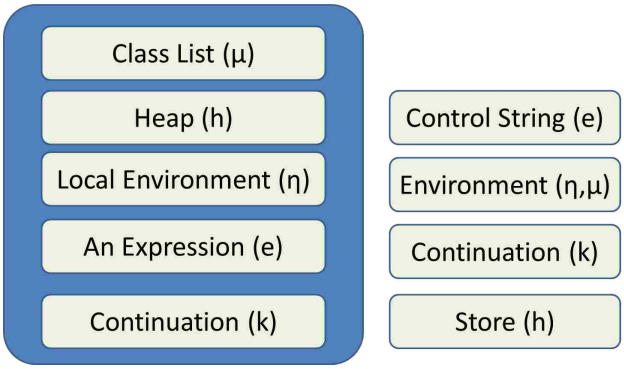

Figure 3.3: Components of a Javalite state and their CEKS equivalents.

Javalite, Java constructors have to be created as a separate method which must be explicitly invoked. We currently use a naming convention that picks a unique name (typically "construct") for the Javalite equivalent of the constructor. Despite these differences between Java and Javalite, the process of converting between them could easily be performed automatically.

3.2 Javalite Machine Syntax

Javalite uses a variant of the CEKS syntactic machine (Felleisen et al. [1987]) for its operational semantics. A CEKS machine has four major components. A control string (C), an environment (E), a continuation (K), and a store (S). The control string is a program, command, or instruction to be evaluated. Initially, the entire program is the control string. An environment is a finite mapping from variables to locations whereas a store is a finite mapping from locations to closures (i.e. to values) (Felleisen et al. [2009]). To look up the value of a variable, its location is determined from the environment mapping E in order to retrieve its value from the store S. A continuation K is a last-in-first-out data structure that saves the contexts of redexes as they are evaluated to values which can then fill the holes in the contexts.

Our Javalite semantics are closely modeled after a CEKS machine. The various components of a Javalite state are shown in Fig. 3.3. The expression e is equivalent to a CEKS machine's control string. The program heap h, the local environment η, and the continuation k are equivalent to a CEKS machine's store, environment, and continuation respectively. The Javalite program state has an additional component (the class list μ), which is equivalent to the source code of all classes used by the program. All these components of a Javalite program state are defined syntactically by a grammar in PLT Redex. This grammar is an extension of the surface syntax grammar and is shown in Fig. 3.4. The next sections present these Javalite program state components as well as their structure and functionality.

```
    e ::= ....
        | (raw v @ m (v ...))
object ::= ((C [f loc] ...) ...)
   hv ::= v
        | object
    h ::= mt
        | (h [loc -> hv])
    η ::= mt
        | (η [x -> loc])
state ::= (μ h η e k)
    k ::= ret
        | (* $ f -> k)
        | (* @ m (e ...) -> k)
        | (v @ m (v ...) * (e ...) -> k)
        | (* == e -> k)
        | (v == * -> k)
        | (C * -> k)
        | (* instanceof C -> k)
        | (x := * -> k)
        | (x $ f := * -> k)
        | (if * e else e -> k)
        | (var T x := * in e -> k)
        | (begin * (e ...) -> k)
        | (pop η k)
```

Figure 3.4: The Javalite machine syntax as defined in PLT Redex.

3.2.1 The Heap (h):

The Javalite heap is a mapping of locations (integers) to objects. It is defined by the production $h ::= mt \mid (h\ [loc \rightarrow hv])$ in Fig. 3.4 in which the symbol mt represents an empty heap. The definition is recursive since a mapping of a location to a heap value $[loc \rightarrow hv]$ is paired with a heap definition h, which is either the mt symbol or a heap definition containing other location mappings. An example of a heap as defined by the machine syntax is the expression $(((mt\ [0 \rightarrow true])\ [1 \rightarrow (Object)])\ [2 \rightarrow false])$. Locations 0 and 2 are mapped to boolean values while location 1 is mapped to an instance of the base class *Object*. In Javalite, both class fields and local variables are stored on the heap. Therefore, the mappings in the sample heap just presented could be for either fields or locals.

An important attribute of Javalite heaps is that they are always in canonical form. This means that identical objects map to exactly the same heap location (Iosif [2001]). This canonical ordering ensures that heap entries are differentiated by their structure and shape as opposed to their location on the heap. We have also implemented a garbage collector for Javalite. Since these operational semantics are runnable, it serves to contain the state space explosion problem by implementing the aforementioned canonicalization scheme. It runs after a configurable number of transitions of the state machine, performing a mark and sweep collection. The garbage collector is also responsible for applying the canonicalization transformations

```
class foo {
}

class Assertion {
  public static void main(String[] args) {
    foo a = new foo();
    foo b = new foo();

    assert (a != b); // This assertion is valid in Java, but not in Javalite
  }
}
```

Figure 3.5: A sound Java assertion that does not hold in Javalite.

of the heap for heap symmetry. The canonical order is based on the number of fields in objects. Objects with fewer fields are assigned lower numeric heap locations than objects with more fields. Ties are broken between objects with the same number of fields by using the lexicographical ordering of the field names. Therefore, the entire heap is sorted by number of fields and their lexicographical ordering. The garbage collector also applies *hash consing*, a technique that ensures that different objects with the same future all map to the same heap location. This means that two instances of a class without fields are equal as shown in Fig. 3.5. This is a deviation from Java's semantics in which creating more than one new instance of a class that has no fields will always yield unique heap references. In Javalite, creating more than one new instance of a class with no fields yields the same heap references.

An important aspect of the heap is its storage format. The format of objects on the heap is determined by the grammar production *object ::= ((C [f loc] ...) ...)* in the machine syntax definition. An object is stored as a list of the classes in its hierarchy, each of which is grouped with its fields and their corresponding heap locations. This format makes it easy to implement the instance-of and cast operators since heap entries contain type information.

3.2.2 The Local Environment

Since local variables are stored on the heap, the local environment (η) is a mapping from local variables to heap locations. It is defined by the production $\eta ::= mt \mid (\eta\ [x \rightarrow loc])$ in the machine syntax grammar. The symbol mt represents an environment without any local variable declarations. Non-empty environments are defined recursively using a tuple of the form $(\eta\ [x \rightarrow loc])$ where each mapped variable $[x \rightarrow loc]$ is paired with a previously defined (possibly empty) local environment. Consider a method that declares three local variables A, B, and tmp, of type *Object*. Its local environment would be $(((mt\ [A \rightarrow 0])\ [B \rightarrow 1])\ [tmp \rightarrow 2])$ where the locations 0, 1, and 2 represent the appropriate numeric

locations where the corresponding variables are stored on the heap. Whenever the value of a local variable is needed, its heap location is looked up from the local environment and then a heap-lookup of the determined location is performed to obtain the value of the local variable.

3.2.3 The Continuation (k)

The continuation k in a Javalite program state represents the computation to be performed after the control string has been evaluated. The value resulting from the evaluation of a control string may have been required as an intermediate value of another computation. If this is the case, then such a pending computation (the continuation) is described as having a *hole*. Holes are marked with asterisks in the continuation productions as shown in the Javalite machine syntax in Fig. 3.4. Continuations can also be chained using the arrow symbol $\rightarrow$. The syntax $a \ \rightarrow \ b$ means that the continuation a will be computed followed by continuation b. We present a brief overview of Javalite's continuations next.

($*$ \$ **f** $\rightarrow$ **k**) : Reducing the current expression to an object in preparation for a field access before continuing with k.

($*$ @ **m** (**e** ...) $\rightarrow$ **k**) : Reducing the current expression to the object on which method invocation will be performed.

(**v** @ **m** (**v** ...) $*$ (**e** ...) $\rightarrow$ **k**) : Evaluating the current expression as an argument to a method invocation.

($*$ == **e** $\rightarrow$ **k**) : Reducing the left operand of the equality operator to a value.

(**v** == $*$ $\rightarrow$ **k**) : Reducing the right operand of the equality operator to a value.

(**C** $*$ $\rightarrow$ **k**) : Reducing the current expression to an object for casting to an instance of C.

($*$ **instanceof C** $\rightarrow$ **k**) : Reducing the current expression to an object on which to check membership in the class hierarchy of C.

(**x** := $*$ $\rightarrow$ **k**) : Evaluating an expression for assignment to a variable.

(**x** \$ **f** := $*$ $\rightarrow$ **k**) : Reducing the expression for assignment to a field.

(**if** $*$ **e else e** $\rightarrow$ **k**) : Reducing the expression to a boolean value which is the predicate of an if-statement.

(**var T x** := $*$ **in e** $\rightarrow$ **k**) : Reducing an expression for assignment to a local variable.

(**begin** $*$ (**e** ...) $\rightarrow$ **k**) : Reducing an expression in a list of expressions to be reduced.

(**pop** η **k**) : Restoring the local environment η before continuing with k.

Having discussed Javalite's surface and machine syntax, we now present a discussion of its reduction rules.

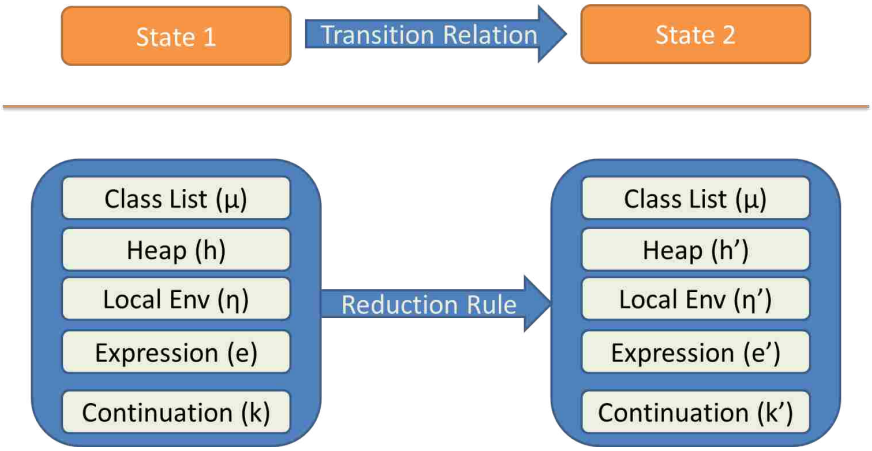

Figure 3.6: The relationship between transition relation and the reduction relation.

3.3 Reduction Rules

Javalite's operational semantics are based on the concept of evolving program state. Consequently, they can be represented as a transition relation on states. This transition relation is encoded as a set of reduction rules that transform one state into another. An overview of this process is shown in Fig. 3.6. The individual state components of the initial state on the left are reduced by the reduction rule to new components in the resultant state on the right.

An example of the evolution of states by the reduction rules is shown in Fig. 3.7, which shows the full state of a Javalite program defined by the *state* production in the machine syntax as well as a new state to which it is reduced. The initial state shown on the left has a raw method invocation, meaning that a method (*swap*) is being invoked on a typed pointer into the heap. Therefore a location in the heap needs to be allocated for the object reference *this* (since the object pointed to by *this* depends on the invoked method). Location 10 is allocated and the typed pointer being used for the method invocation is written to that location. Next, a mapping of the local environment variable *this* to the allocated location (10) is created. Note that a new local environment $\eta 1[this \rightarrow 10]$ is created. $\eta 1$ inherits all the mappings of η and adds this new mapping.

For a method call, the control string invoking a method is replaced with the control string of the target method. The continuation after a method call simply restores the previous local environment η. This is the continuation *pop* η k. Continuations are stacked as various operations unfold. Recall that all these transformations of the control string and the continuation stack are purely syntactic operations defined by a transition relation which is implemented as a set of rewriting rules. Following below is a discussion of all the Javalite reduction relation rules.

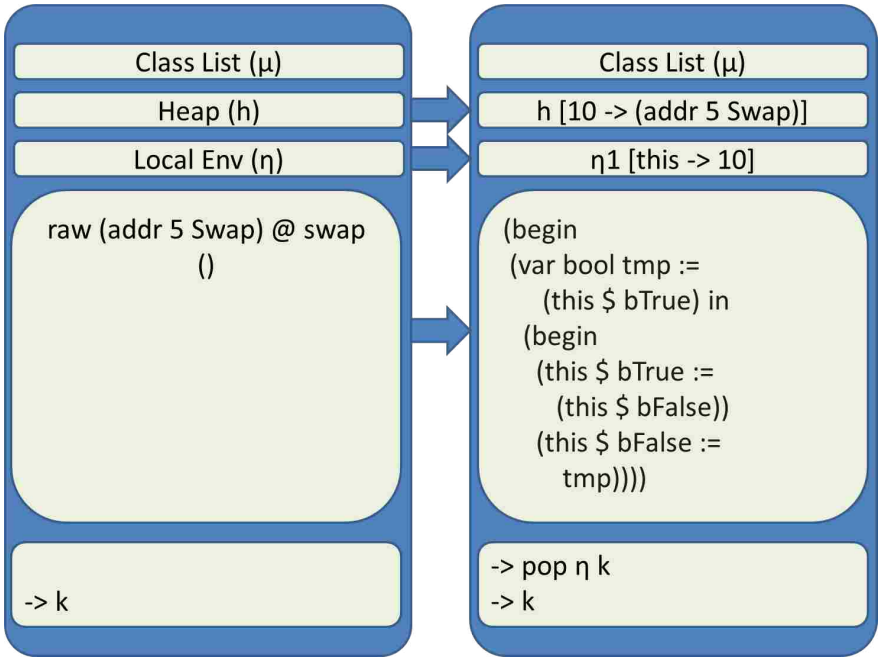

Figure 3.7: Reduction from initial state at a method invocation.

Variable Access Rule. Javalite's reduction rules are defined by a Redex pattern and expression. The pattern describes redexes and their contexts (Felleisen et al. [2009]) and is displayed on the first line of the rule. The expression determines the result of a use of such a rule on a specific expression and is displayed on the second line. For the *variable access* rule shown below, the pattern is the state $(\mu\ h\ \eta\ x\ k)$ whereas the expression is the state $(\mu\ h\ \eta\ v\ k)$.

$$(\mu\ h\ \eta\ x\ k) \qquad \text{[variable access]}$$
$$(\mu\ h\ \eta\ v\ k)$$
$$\text{where } v = \text{h-lookup}[[h, \text{η-lookup}[[\eta, x]]]]$$

Each rule also has an assigned name displayed in square brackets on the first line of the rule. For side conditions required for a successful pattern match, a *where* clause is included after the rule's expression. Such side conditions may use meta-functions, which are mathematical functions on the entities within the model (Felleisen et al. [2009]). *h-lookup* and *η-lookup* are examples of meta-functions in the variable access rule. The complete listing of Javalite's meta-functions is presented in Appendix A.

The variable access rule states that for the state $(\mu\ h\ \eta\ x\ k)$ (whose control string x is a variable identifier) it will first look up the identifier in the local environment η to determine its heap location using the *η–lookup* meta-function. The resultant heap location is then passed to *h-lookup* to find the variable v which the identifier x refers to on the heap h. The application of this rule then yields a new state whose control string is the variable found on the heap.

Class Instantiation Rule. The class instantiation expression (*new C*) indicates that a new object needs to be created on the heap. The *new object creation* rule proceeds as follows. It calls the *fields-parents+self* meta-function to create a list of fields for each class in the hierarchy of class *C*. *fields-parents+self* returns a hierarchical (nested) list of the form (([T_0 f_0] ...) ...). The ith component of this nested list is the field declaration list of the ith class in the hierarchy of C.

(μ h η (new C) k) [new]
(μ h_1 η (addr loc_1 C) k)
where ((\[T_0 f_0\] ...) ...) = fields-parents+self[[μ, C]],
(C_0 ...) = class-parents+self[[μ, C]],
((v_0 ...) ...) = (default-value*[[(T_0 ...)]] ...),
($number_0$ ...) = (get-length[[(T_0 ...)]] ...),
((loc_0 ...) ...) = h-malloc-n*[[h, $number_0$, ...]],
$object$ = (C (C_0 [f_0 loc_0] ...) ...),
h_0 = h-extend*[[h, [loc_0 -> v_0], ..., ...]],
loc_1 = h-malloc[[h_0]],
h_1 = h-extend*[[h_0, [loc_1 -> $object$]]]

As mentioned earlier, field values are stored on the heap. In order to allocate heap locations for each field of the object, the *get-length* meta-function is used to compute the length of the field list of each class. These lengths are then passed to the *h-malloc-n** metafunction which allocates enough heap locations for all the fields in the hierarchy of the object. Since these locations must be initialized with their default values, the class instantiation rule also calls the *default-value** meta-function to create a hierarchial list of the default values of each of the fields in the new object's class hierarchy. Recall that the storage format of an object on the heap is defined by the grammar production *object ::= ((C [f loc] ...) ...)* in the machine syntax definition. A list of the actual class identifiers in the hierarchy is therefore required. This list is created by the call to the *class-parents+self* meta-function. The object to be stored on the heap is then created by the assignment $object$ = (C(C_0 [f_0 loc_0] ...) ...), which groups all the fields by their class identifiers and lists each of the fields with its allocated heap location.

Note that the Javalite heap allocation functions return new unused heap locations. They do not alter the passed in heap. A new heap h_0 is therefore explicitly created by extending the input heap with mappings from each of the allocated field locations to the default values of the corresponding fields. The reduction rule also calls *h-malloc* on the newly created heap h_0 to allocate a new location loc_1 to point to the newly created object. It then extends heap h_0 to include a mapping from the new location loc_1 to the newly created object. The reduction rule finally creates a new state (μ h_1 η (*addr* loc_1 C) k) using the new heap h_1 and a new (pointer) expression (*addr* loc_1 C) containing the location of the newly created object.

Field Access Rule. For a field access expression e \$ f, the object e must be evaluated before the field access can be performed. Therefore, the *field access - object evaluation* reduction rule creates a new state whose control string is the expression e.

$(\mu\ h\ \eta\ (e\ \$\ f)\ k)$ [field access - object eval]
$(\mu\ h\ \eta\ e\ (*\ \$\ f \text{ -> } k))$

$(\mu\ h\ \eta\ (\text{addr } loc\ C)\ (*\ \$\ f \text{ -> } k))$ [field access]
$(\mu\ h\ \eta\ v\ k)$
where $object = \text{cast}[[\text{h-lookup}[[h, loc]], C]]$,
$loc_0 = \text{field-lookup}[[object, f]]$,
$v = \text{h-lookup}[[h, loc_0]]$

When a state's continuation is a field access continuation $(* \$ f \rightarrow k)$, a pointer expression control string $(addr\ loc\ C)$ indicates that the field access can proceed since the pointer expression specifies the object on which to perform the field access. In this scenario, the *field access* reduction rule retrieves the object at the location *loc* on the heap by calling *h-lookup*. It then casts the retrieved object to an instance of class C and determines the location loc_0 on the heap where field f is stored by calling the *field-lookup* meta-function. The rule then calls *h-lookup* to retrieve the value of field f stored at loc_0. The *field access* reduction rule creates a new state $(\mu\ h\ \eta\ v\ k)$ whose control string is the newly found value v. Since the field access operation is complete, it also restores the continuation k in the new state.

Method Invocation Rules. In a method invocation expression $(e_0\ @\ m\ (e_1\ ...))$, the *object eval* rule is used to evaluate the subexpression e_0 on which the method is being invoked. To indicate that method invocation must be performed after this expression is evaluated, the rule creates the method invocation continuation $(* @ m\ (e_1\ ...) \rightarrow k)$, which contains the method identifier and arguments as well as the original continuation k that indicates the compuation to be performed after the method invocation completes.

$(\mu\ h\ \eta\ (e_0\ @\ m\ (e_1\ ...))\ k)$ [method invocation - object eval]
$(\mu\ h\ \eta\ e_0\ (*\ @\ m\ (e_1\ ...)\ ->\ k))$

$(\mu\ h\ \eta\ v\ (*\ @\ m\ (e_0\ e_1\ ...)\ ->\ k))$ [method invocation - arg0 eval]
$(\mu\ h\ \eta\ e_0\ (v\ @\ m\ ()\ *\ (e_1\ ...)\ ->\ k))$

$(\mu\ h\ \eta\ v_i\ (v_o\ @\ m\ (v_a\ ...)\ *\ (e_0\ e_1\ ...)\ ->\ k))$ [method invocation - argi eval]
$(\mu\ h\ \eta\ e_0\ (v_o\ @\ m\ (v_a\ ...\ v_i)\ *\ (e_1\ ...)\ ->\ k))$

$(\mu\ h\ \eta\ v_o\ (*\ @\ m\ ()\ ->\ k))$ [method invocation - no args]
$(\mu\ h\ \eta\ (\text{raw}\ v_o\ @\ m\ ())\ k)$

$(\mu\ h\ \eta\ v_1\ (v_o\ @\ m\ (v_0\ ...)\ *\ ()\ ->\ k))$ [method invocation - args]
$(\mu\ h\ \eta\ (\text{raw}\ v_o\ @\ m\ (v_0\ ...\ v_1))\ k)$

Once the expression on which a method invocation is being performed has been evaluated to an object, the resultant state is of the form $(\mu\ h\ \eta\ v\ (*\ @\ m\ (e_0\ e_1\ ...)\ \rightarrow\ k))$. The argument expressions $e_0, e_1, ...$ need to be evaluated next. The *method invocation - arg0 eval* reduction rule therefore starts the execution of e_0 by creating a new state whose the control string is e_0. It then creates an argument evaluation continuation $(v\ @\ m\ ()*\ (e_1\ ...)$, which contains the information required to continue the method invocation operation after the first argument expression has been evaluated. This continuation holds the object on which the method is being invoked, the method identifier, and the rest of the argument expressions that still need to be evaluated. Note the position of the hole in this continuation. It occurs immediately after the empty parentheses and this indicates that the value of the first argument will be the first value to be added to the method's argument list. If more than one argument is being passed to the method m, the remaining arguments are evaluated by the *argi eval* rule. This rule creates a new state whose control string is the next argument to be evaluated and whose continuation includes the value v_i of the last argument.

In the absence of arguments to the method being invoked, the *no args* reduction rule generates a state with an argument-less *raw method invocation* expression. In a raw method invocation, all the argument expressions have been evaluated. In the *args* rule, for example, all the final argument expressions have just been evaluated yielding the value *v_1*. Since the method can now be invoked, the rule creates a state whose control string is a raw method invocation and whose continuation k is the operation that succeeds the method invocation.

Raw Method Invocation. A raw method invocation expression is comprised of a pointer, a method identifier, and a list of primitive values (arguments). In order to syntactically evaluate a raw method invocation, we need to perform a virtual method lookup to determine the appropriate class in the hierarchy

on which the method should be invoked. The first requirement is a list of class identifiers in the hierarchy of the class C on which the method is being invoked. This list is created by retrieving the object on which the call is being invoked (using the *h-lookup* metafunction) and passing this object to the *class-list-from-object* meta-function.

$(\mu\ h\ \eta\ (\text{raw (addr } loc\ C) \text{ @ } m\ (v_x\ ...))\ k)$ [raw method invocation]
$(\mu\ h_0\ \eta_0\ e_m\ (\text{pop } \eta\ k))$
where $(C_0\ C_p\ ...) = \text{class-list-from-object}[[\text{h-lookup}[[h, loc]]]]$,
$(any_0\ ...\ (C_t\ (x_m\ ...)\ e_m)\ \text{error}\ ...) = (\text{method-lookup}[[\text{class-lookup}[[\mu, C_p]], m]]\ ...)$,
$(loc_o\ loc_x\ ...) = \text{h-malloc-n}[[h, \texttt{(add1 (length } (v_x\ ...))\texttt{)}]]$,
$h_0 = \text{h-extend*}[[h, [loc_o \text{ -> (addr } loc\ C_t)], [loc_x \text{ -> } v_x], ...]]$,
$\eta_0 = \text{η-extend*}[[\eta, [\text{this -> } loc_o], [x_m \text{ -> } loc_x], ...]]$

The virtual method lookup is then performed by calling the *method-lookup* meta-function on each of the class declarations for each class identifier in the hierarchy of C. Class declarations for a class identifier are retrieved by the *class-lookup* meta-function. *method-lookup* returns the class identifier, method argument names, and method body of the class if the specified method is defined for the specified class identifier. Otherwise, it returns the *error* keyword. The syntax $(any_0\ ...\ (C_t\ (x_m\ ...)e_m)\ error\ ...)$ selects the appropriate virtual method from this list since the method immediately preceding the trailing list of zero or more *error* values is the virtual method closest in the hierarchy to the class C.

Now that the exact method being invoked is known, we need to allocate heap locations for the arguments to the method and the reference *this* (recall that all variables, fields, and arguments are stored on the heap). The number of locations for the arguments and the object *this* is computed by the expression ($add1$ ($length$ (v_x ...)) locations) and is passed to the *h-malloc-n* meta-function, which returns a list of new heap locations for the respective arguments. These locations are then mapped to the arguments and the object *this* by a call to the *h-extend** meta-function which returns a new heap h_0.

A new local environment η_0 is also created by extending the initial environment η with bindings for the arguments and the reference *this*. The reduction rule then creates a new state containing the new heap h_0 and the new local environment η_0. The rule also replaces the raw method invocation control string with the control string e_m representing the body of the virtual method being invoked. The continuation of the new state is the ($pop\ \eta\ k$) continuation which will restore the pre-invocation local environment and continuation (η and k respectively) after the method invocation evaluation is complete.

Equals Rules. The equality testing expression (e_0 == e) requires the evaluation of the expressions on both sides of the equality. The *l-operand eval* and *r-operand eval* rules shown below initiate the evaluation of the left and right expressions respectively. Once the right hand side expression e has been evaluated to

a value v_0, the Racket *equal?* function is used to compute the result of the comparison. The resultant Racket boolean value is converted to a Javalite boolean value v_{res} by the $->bool$ meta-function. The *equals* reduction rule then restores the original continuation k since the comparison is complete.

$(\mu\ h\ \eta\ (e_0 == e)\ k)$ [equals - l-operand eval]
$(\mu\ h\ \eta\ e_0\ (* == e \text{ -> } k))$

$(\mu\ h\ \eta\ v\ (* == e \text{ -> } k))$ [equals - r-operand eval]
$(\mu\ h\ \eta\ e\ (v == * \text{ -> } k))$

$(\mu\ h\ \eta\ v_0\ (v_1 == * \text{ -> } k))$ [equals]
$(\mu\ h\ \eta\ v_{res}\ k)$
where v_{res} = `(->bool (equal? ` v_0 v_1 `))`

Typecast Rules. The evaluation of a type cast expression $(C\ e)$ is initiated by evaluating the expression e to a pointer value. As expected, the *typecast - object eval* rule shown below creates a new state with e as the control string and a typecast continuation *(C * → k)*. Recall that Javalite pointers are typed and that the heap storage format for an object includes all the classes and their fields in the hierarchy of an object. Once the expression is evaluated to some pointer value $(addr\ loc\ C_0)$, there are two operations that need to be performed by the *typecast* rule. First, the object refered to by the pointer needs to be retrieved (using the *h-lookup* meta-function) to ensure that the cast operation is valid (class C_1 to which we are casting must be in the hierarchy of the pointer's current class C_0). The *cast?* meta-function returns a boolean value indicating whether the cast can be performed. Next, the reduction rule needs to create the new state $(\mu\ h\ \eta\ (addr\ loc\ C_1)\ k)$ whose control string is a pointer to the same heap location *loc* but of the cast type C_1, completing the cast operation.

$(\mu\ h\ \eta\ (C\ e)\ k)$ [typecast - object eval]
$(\mu\ h\ \eta\ e\ (C * \text{ -> } k))$

$(\mu\ h\ \eta\ (\text{addr } loc\ C_0)\ (C_1 * \text{ -> } k))$ [typecast]
$(\mu\ h\ \eta\ (\text{addr } loc\ C_1)\ k)$
where $object$ = h-lookup[[h, loc]],
`(cast? ` $object$ C_1`)`

Instanceof Rules. An instance-of expression $(e\ instanceof\ C)$ evaluates to a boolean value. The *object eval* rule presented below initiates the evaluation of expression e. Once e has been evaluated to a pointer value $(addr\ loc\ C_0)$, the *instanceof* rule checks whether the object at location *loc* on the heap can be cast to the type C_1 specified by the instanceof continuation. The object is retrieved with the *h-lookup* meta-function

as expected. The *cast?/->bool* meta-function then returns a Javalite boolean value v_{res} indicating whether the class C_0 can be cast to class C_1. The *instanceof* rule creates a new state whose control string is the value v_{res} and restores the continuation k.

$(\mu\ h\ \eta\ (e\ \text{instanceof}\ C)\ k)$ [instanceof - object eval]
$(\mu\ h\ \eta\ e\ (*\ \text{instanceof}\ C \rightarrow k))$

$(\mu\ h\ \eta\ (\text{addr}\ loc\ C_0)\ (*\ \text{instanceof}\ C_1 \rightarrow k))$ [instanceof]
$(\mu\ h\ \eta\ v_{res}\ k)$
where $object = \text{h-lookup}[[h, loc]]$,
$v_{res} =$ `(cast?/->bool` $object$ C_1`)`

Variable Assignment Rules. Given a state $(\mu\ h\ \eta\ (x\ :=\ e)\ k)$, the evaluation of the variable assignment expression is initiated by the *object eval* rule below. In order to perfom a variable assignment, the location of the variable is required in order to update the heap with the new value v being assigned. The *assign* rule determines the location of variable x using the *η-lookup* meta-function and then updates the heap h with a mapping $[loc \rightarrow v]$ from the resultant location loc to the value v.

$(\mu\ h\ \eta\ (x\ :=\ e)\ k)$ [assign -- object eval]
$(\mu\ h\ \eta\ e\ (x\ :=\ *\ \rightarrow k))$

$(\mu\ h\ \eta\ v\ (x\ :=\ *\ \rightarrow k))$ [assign]
$(\mu\ h_0\ \eta\ v\ k)$
where $loc = \text{η-lookup}[[\eta, x]]$,
$h_0 = \text{h-extend*}[[h, [loc \rightarrow v]]]$

Field Assignment Rules. The evaluation of a field assignment expression $(x\ \$\ f\ :=\ e)$ is similar to the variable assignment evaluation. The *object eval* rule initiates the evaluation of expression e. To complete the assignment, we need to determine the location of the field f on the heap. The object referred to by x has this information. Therefore, an *η–lookup* call returns the heap location loc_0 of the pointer x.

$(\mu\ h\ \eta\ (x\ \$\ f\ :=\ e)\ k)$ [assign field -- object eval]
$(\mu\ h\ \eta\ e\ (x\ \$\ f\ :=\ *\ \rightarrow k))$

$(\mu\ h\ \eta\ v\ (x\ \$\ f\ :=\ *\ \rightarrow k))$ [assign field]
$(\mu\ h_0\ \eta\ v\ k)$
where $loc_0 = \text{η-lookup}[[\eta, x]]$,
$(\text{addr}\ loc_1\ C) = \text{h-lookup}[[h, loc_0]]$,
$object = \text{cast}[[\text{h-lookup}[[h, loc_1]], C]]$,
$loc_2 = \text{field-lookup}[[object, f]]$,
$h_0 = \text{h-extend*}[[h, [loc_2 \rightarrow v]]]$

The actual value of the pointer ($addr\ loc_1\ C$) is retrieved by passing loc_0 to the heap lookup meta-function *h-lookup*, at which point we can look up the object referred to by x by passing loc_1 to *h-lookup*. The object is cast to an instance of C to match the typed pointer x to ensure that the right field will be used for the assignment. The location loc_2 of the field f is returned by the *field-lookup* meta-function. The *assign field* rule finally extends the heap h with a mapping $[loc_2 \rightarrow v]$ of the field f's location to the computed value v of the expression being assigned. The resultant state therefore has the newly updated heap h_0.

If-Then-Else Rules. The evaluation of the predicate e_p of a branching expression ($if\ e_p\ e_t\ else\ e_f$) is initiated by the *if-then-else – object eval* rule shown below. Once the predicate e_p has been evaluated, its value v is used to fill the hole in the if-then-else continuation ($if\ *\ e_t\ else\ e_f\ \rightarrow\ k$) of the *if-then-else* rule. The Racket *if* function compares the terms v and *true* in order to select one of the expressions e_t and e_f for the new control string.

$(\mu\ h\ \eta$ (if $e_p\ e_t$ else e_f) $k)$ [if-then-else -- object eval]
$(\mu\ h\ \eta\ e_p$ (if * e_t else e_f -> k))

$(\mu\ h\ \eta\ v$ (if * e_t else e_f -> k)) [if-then-else]
$(\mu\ h\ \eta$ `(if (equal? v true)` $e_t\ \ e_f)\ \ k)$

Variable Declaration Rules. As expected, a variable declaration expression ($var\ T\ x\ :=\ e_0\ in\ e_1$) proceeds by first evaluating the expression e_0 (as shown by the *variable-declaration-object-eval* rule) to a value v. The *variable-declaration* rule then performs the variable assignment by allocating an unused location loc_x for the variable x on the heap with a call to the *h-malloc* meta-function.

$(\mu\ h\ \eta$ (var $T\ x$:= e_0 in e_1) $k)$ [variable declaration -- object eval]
$(\mu\ h\ \eta\ e_0$ (var $T\ x$:= * in e_1 -> k))

$(\mu\ h\ \eta\ v$ (var $T\ x$:= * in e_1 -> k)) [variable declaration]
$(\mu\ h_0\ \eta_0\ e_1$ (pop $\eta\ k$))
where loc_x = h-malloc[[h]],
h_0 = h-extend*[[h, [loc_x -> v]]],
η_0 = η-extend*[[η, [x -> loc_x]]]

It then extends the heap h with a mapping from the new location to the value v of the variable ($[loc_x \rightarrow v]$) yielding a new heap h_0 and then creates an entry for the new variable x in the local environmnent by adding the mapping $[x \rightarrow loc_x]$ to it. The scope of the newly declared variable is the expression e_1, which is then evaluated next (and therefore becomes the new control string). After the evaluation of e_1, the variable x will no longer be valid. To ensure that the previous local environment is restored, the *variable*

declaration rule creates a pop continuation (*pop* η k) which will restore the previous local environment η and continuation k after the evaluation of e_1.

Begin Expression Rules. A Javalite begin expression is a sequence of any of the expressions defined by the surface syntax in Fig. 3.1. Sequences of expressions are evaluated in left-to-right order. The simplest case is the empty expression list (*begin*). The *begin-empty-expression-list* rule simply rewrites the (*begin*) control string into the Javalite *unit* keyword. For a non-empty expression list such as (*begin* e_0 e_1 ...), the *begin-*e_0*-evaluation* rule initiates the evaluation of the first expression e_0 and creates a *begin* continuation of the form (*begin* $*$ (e_1 ...), which indicates that there rest of the expressions (e_1 ...) will need to be evaluated before the original continuation k can be considered.

Once a state's expression control string has been evaluated to a value v, one of two reduction rules may apply if the evaluated expression had been from a begin (sequence) expression. If the evaluated expression had been the last one in the list, then the *begin-complete* rule restores the original continuation k that specified the next operation after the evaluation of the list of expressions. Otherwise, if the evaluated expression was not the last one in the sequence, the *begin-*e_i*-evaluation* rule retrieves the next expression in the sequence (e_i) from the continuation and substitutes it for the current control string (which is simply v, the computed value of the last expression). The *begin-*e_0*-evaluation* and *begin-*e_i*-evaluation* rules result in a left-to-right evaluation of the expressions in an expression list.

(μ h η (begin) k) [begin -- empty expression list]
(μ h η unit k)

(μ h η (begin e_0 e_1 ...) k) [begin -- $e_{0\ \text{evaluation}}$]
(μ h η e_0 (begin * (e_1 ...) -> k))

(μ h η v (begin * (e_i e_{i+1} ...) -> k)) [begin -- $e_{i\ \text{evaluation}}$]
(μ h η e_i (begin * (e_{i+1} ...) -> k))

(μ h η v (begin * () -> k)) [begin -- complete]
(μ h η v k)

Pop Rule. The *pop* rule creates a new state by replacing the local environment η in the initial state with the environment η_0 encoded in the *pop* continuation. It also restores the continuation k. This rule is therefore suitable for cleaning up after a method call (by restoring the caller's environment and continuation) or variable declaration.

(μ h η v (pop η_0 k)) [pop η]
(μ h η_0 v k)

3.4 Custom Redex Compiler

We also created a customized version of the Redex tool which allows developers to specify their languages' known invariants, such as determinism of reductions. The ability to give such hints to the compiler enables more compiler optimizations than are possible in the standard Redex compiler. The custom compiler also allows for the direct use of native Racket data structures such as hash tables within Redex terms. This optimization prevents the use of slower data structures while still supporting the same syntax as the Redex compiler (with the minor annotation changes required to specify determinism/non-determinism of reduction relations). It also caches the results of some intermediate computations based on the extra information it has about a reduction relation.

3.5 Results

Since it is a semantics engineering toolkit, Redex provides various tools to experiment with and view the reduction behavior of various terms. One such tool is the *traces* tool. It consumes a Redex-specified language, a reduction relation, and an expression and then opens a window and draws a directed graph of all reduction sequences starting from the given expression according to the specified reduction system (Felleisen et al. [2009]). A snapshot of the *traces* tool's output when applied to the Javalite language and it's reduction relation is shown in Fig. 3.8. The states are shown as rectangles with solid borders within which the components of each state are displayed. A partial heap and the complete local environment, control string, and continuation components of the states (respectively) are visible in the example in Fig. 3.8. Another similar tool is the *stepper* tool. It also displays a directed graph of reduction sequences but allows for customizable stepping through the reduction sequence. It can break on a named reduction rule or after every single reduction, allowing the semantics engineer to study the progression of a reduction sequence. An example of the *stepper* tool running is shown in Fig. 3.9.

As part of the Javalite development process, we have implemented a version of the Lambda Calculus and Church numerals in Java and ported it to Javalite. The complete source listing for these programs is available from our Javalite repository[1]. The average run time of the Java unit tests is 2ms while the average run time of the equivalent Javalite implementations is 46.62s without Javalite garbage collection and 1h 2min with Javalite garbage collection.

[1] https://github.com/ericmercer/javalite/tree/master/sequential/examples/Church

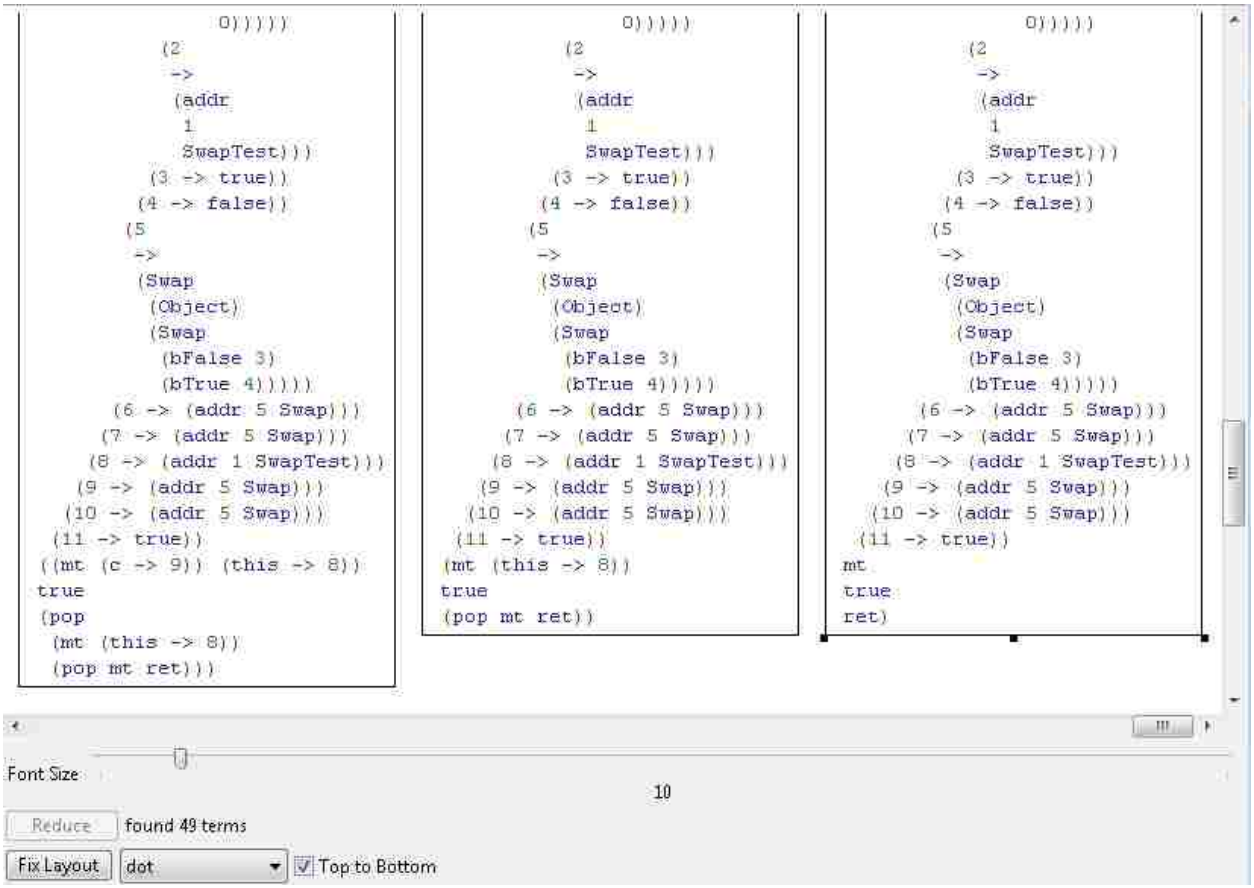

Figure 3.8: The PLT Redex traces tool

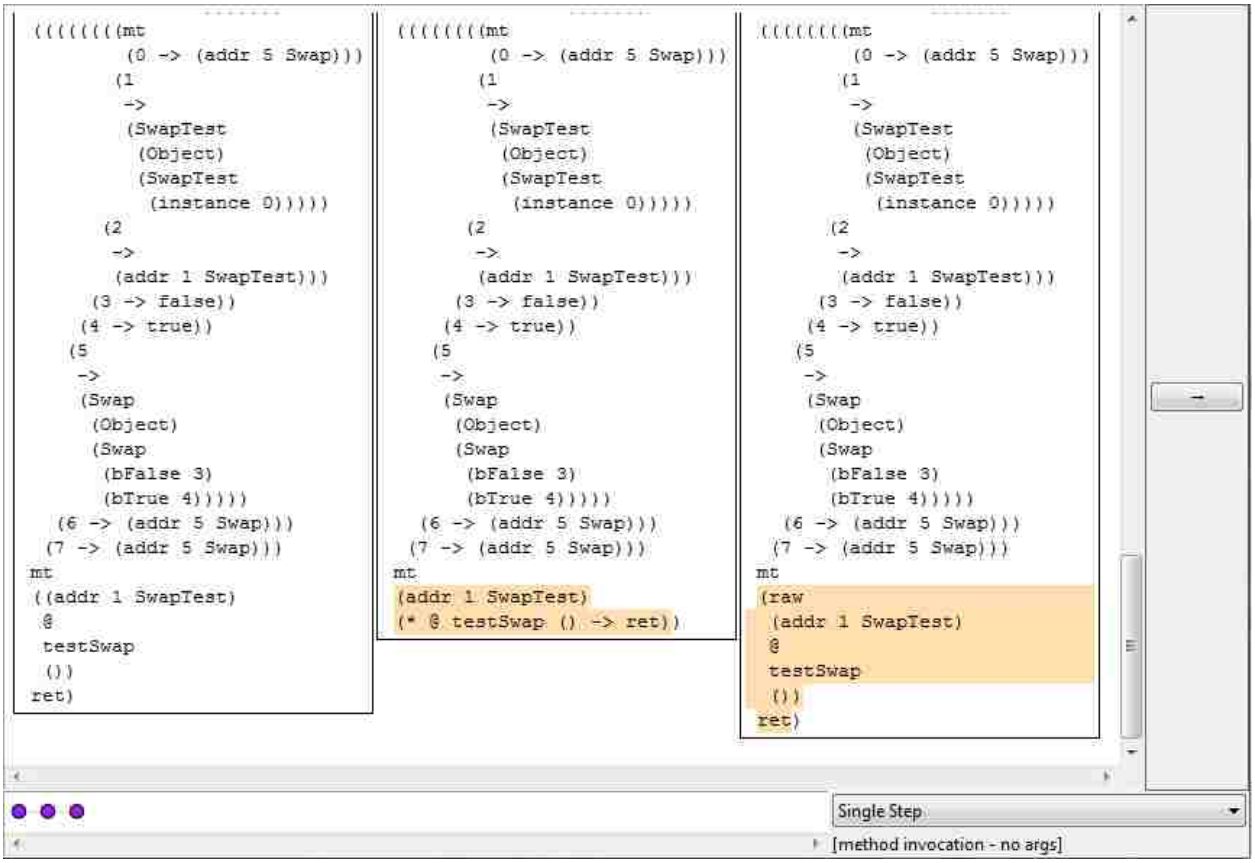

Figure 3.9: The PLT Redex stepper tool

3.6 Conclusion

We have presented a semi-formal model of the Java language. Unlike other semantics, the Javalite model we have presented is based on the PLT Redex environment and is therefore executable. Consequently, it enables rapid prototyping of ideas, analysis algorithms such as model checking algorithms, and language extensions. We have illustrated how the language's reduction system works, including demonstrating two of the key PLT Redex tools available for inspecting the executable reductions. We have also implemented a lambda calculus in both Java and Javalite to compare the run times of both execution systems. Unfortunately, the PLT Redex run times are much higher than the Java runtime's. We have found that there is therefore potential for significant improvement of the PLT Redex compiler, which will result in a drastic reduction in run time of the Redex model. However, we have still found the prototyping flexibility afforded by PLT Redex and the executability of the resultant models to be worth the effort invested in using the tools.

Chapter 4

Coq Model

Coq is a formal proof management system that provides a formal language for writing mathematical definitions, executable algorithms and theorems (The Coq development team [2010]). It also provides an environment for semi-interactive development of machine-checked proofs. Having implemented a working prototype of the Javalite syntax and reduction relation in PLT Redex, we ported them into the Coq theorem prover. Since Javalite is encoded as a reduction relation, it is necessary to ascertain whether or not the relation is decidable. We proved that Javalite's semantics are decidable. Moreover, since reduction relations can be either deterministic or non-deterministic, we also proved that a program state's successor state is not only decidable but that a state always has at most one successor. These are significant properties because they validate the fact that we created a sequential model of Java. In this chapter, we compare the definition and encoding of operational semantics in both Coq and PLT Redex. We then present the Coq definition of the Javalite surface and machine syntax, followed by a discussion of the decidability and determinism proofs.

4.1 Coq Implementation Details

4.1.1 Modeling PLT Redex Syntax Definitions

The Coq theorem prover is, naturally, a significantly different environment from PLT Redex. One of its advantages is that it type checks the definitions of the Javalite syntax. We made several corrections to our original PLT Redex model as that model was not completely type correct. The revisions were trivial and did not substantively change the model itself. Since Coq's primary goal is enabling theorem proving, it does not provide semantics engineers with a toolset geared toward the simplified definition of languages via grammars and extension of such languages by adding new productions. Consequently, the structure of our Javalite Coq syntax is defined by the requirements imposed by Coq.

Coq's syntax is much more verbose than Redex's. In PLT Redex, definitions of the surface syntax closely follow the BNF style as shown in Fig. 3.1. Coq, however, requires every right hand side choice of every production to have an associated name (its constructor). This requirement applies to the reduction rules as well. Each rule must have a named constructor and must be quantified over all the variables it uses. These

requirements lead to a more verbose definition of the Javalite syntax and semantics in Coq. Second, Coq requires that every symbol encountered in a top-down parse of the source script must have been previously defined, unlike the grammar definitions accepted by PLT Redex. This restriction leads to a completely different structure of the Javalite syntax. In addition to this, Coq does not have a way of directly defining a grammar (since the straightforward way of defining grammars conflicts with Coq's requirement that all symbols encountered must already be completely defined). Therefore, there are no separate definitions for the Javalite surface and machine syntax in the Coq model. The layout of the surface and machine syntax is based on the order in which the symbols they define are required.

Our main objective in porting the PLT Redex Javalite model to Coq is to enable formal machine-checked proofs about the model. Consequently, we have altered the PLT Redex model to simplify proofs about the model in Coq. One of the most significant changes we made to the Javalite model when porting it to Coq was to how method calls are handled. The Javalite surface syntax in Fig. 3.1 defines the method invocation expression as taking a list of expressions. In Coq, the equivalent definition would require the use of mutually recursive data types (*Expression* and *ExpressionList*). Using these mutually recursive types would significantly complicate any proofs that could be done about the model in Coq. Therefore, we have modified the Coq model to use A-Normal Form (ANF). Methods are therefore defined to accept only identifiers as arguments. The arguments to a method must therefore first be evaluated and assigned to a new set of identifiers. The method can then be invoked with the identifiers. These method invocation semantics are in constrast to the Redex model's, in which we did not implement ANF. We also changed the Coq equivalent (*SeqExp*) of the PLT Redex sequence of expressions definition *(begin e ...)* because of the same issue of mutually recursive types. The *SeqExp* constructor accepts two *Expression* arguments, either or both of which could also be a *SeqExp*. It is therefore possible to create expression lists using this constructor while avoiding mutually recursive data types.

Another implementation choice that arose when porting the Redex model to Coq was how to handle lists. Consider the Redex class declaration rule *(CL (class C extends C ([T f] ...) (M ...)))*. It includes a list of fields each with their associated types as well as a list of method definitions. To simplify some of the manipulation of these fields and methods in Coq, we used hash maps to store the class declaration information. Field lists are then stored as a mapping from field identifiers to their types. These hashmaps are defined as functions in Coq and can therefore be readily used in any Coq model. Unfortunately, the PLT Redex environment does not have an equivalent predefined utility. Therefore, we used purely syntactic representations for all PLT Redex definitions.

It is noteworthy that syntactically, the traversal of the class hierarchies (which is done when searching for fields or methods, for example) is achieved by pattern matching from the head of the list to the tail and

selecting the most recently defined instance of the field or method. This enables field shadowing and virtual method invocation. Since the Coq environment does not provide such pattern matching facilities, we have implemented field lookup functionality as a function (*field_lookup*) whose arguments are the field, the type of the pointer being used for the field access, and the object on the heap through which to search (since objects store all field values for their class hierarchies). This function calls the *get_reversed_class_hierarchy_CL_to_CLList* function to create a list of the class declarations to be searched from subclass to superclass for the corresponding fields and methods. The complete listing for these functions is presented Appendix C.

4.1.2 Modeling PLT Redex Reduction Rules

In Coq, evaluation semantics can be encoded using either functions or relations. Functions in Coq are required to be both deterministic and total. A key advantage of using functions in Coq is that the deterministic nature of any computation trivially follows from the restriction. However, using functions can be cumbersome. The aforementioned requirements generally necessitate using step indices to enforce termination, which is an additional cumbersome step. See the *get_class_hierarchy_gas_CL_to_CLList* Coq function defined in Appendix C for an example of step indices. Another disadvantage of functions in Coq is that they can be too restrictive for certain applications. When dealing with concurrency, for example, the definition of evaluation needs to be non-deterministic. This automatically rules out the use of functions as the primary definition of concurrent evaluation.

Relations on the other hand provide much more flexibility. They can readily handle both non-deterministic and deterministic evaluation. Coq also has better support for working with relations when working on proofs based on induction (Pierce et al. [2012]). It is therefore natural to use relations for encoding evaluation semantics. However, relations do require that properties such as determinism or totality of the computation they represent be explicitly proved. We express the evaluation of a Javalite program as a relation between states. A Coq proposition is a statement expressing a factual claim and can be either provable or unprovable (Pierce et al. [2012]). The Javalite reduction relation is expressed in Coq as a proposition on pairs of states. Coq allows the stating of either provable or unprovable propositions. Therefore, we explicitly show that the Javalite reduction relation is decidable (Theorem 1). In other words, it is a provable proposition on pairs of states. Since Coq relations defined by propositions can also encode either deterministic or non-deterministic computations, we explicitly show that if a state s reduces to some state s' and also reduces to another state s", then the states s' and s" are equal, which implies that the evaluation relation is deterministic (Theorem 2).

4.1.3 Extraction of Programs From Coq Proofs

An interesting feature of the Coq theorem prover is that a program can be extracted from each theorem that is proven. We have extracted a program from the theorem stating that the Javalite reduction relation is decidable. This program is a Javalite interpreter. The source listing of the Javalite interpreter is available from our Javalite github repository.[1] We ran the extracted interpreter on the Javalite Swap program presented in Chapter 1 and on the Church numerals example. Table 4.1 shows the results. The OCaml version of the extracted Javalite interpreter evaluates the Church numerals test program about seventy times faster than the PLT Redex environment with garbage collection turned off, and an order of magnitude faster than the PLT Redex model with garbage collection enabled. We ran these tests on a Windows 7 Enterprise machine with 8GB of RAM and an Intel Core 2 Quad 2.67Ghz Q9400 CPU using OCaml 4.00.1 and Racket version 5.2.900.1. We now discuss the definition of Javalite in the Coq theorem prover.

Table 4.1: Comparison of PLT Redex and OCaml Javalite Interpreter Performance

Environment	Swap Test	Church Numerals Test
PLT Redex (Garbage Collection)	0.568s	62.63m
PLT Redex (No Garbage Collection)	0.193s	46.62s
Extracted Ocaml Interpreter	0.172s	0.63s

4.2 Javalite Syntax

One basic requirement in Javalite is the ability to identify variables, classes, fields, and so on. The key property of identifiers in Coq is not necessarily their names. Rather, it is the ability to differentiate between them. Therefore, we use the predefined numeric type *positive* in Coq for Javalite identifiers. Predefined types and values in Coq are imported from libraries using the *Require Import* syntax as shown below. There are a couple of notation styles worth pointing out. First, comments in a Coq script are delimited by (* and *), and Coq supports nested comments. The comments in this code block are the equivalent PLT Redex definitions of the subsequent Coq code. Second, the *Definition* keyword in Coq aliases two symbols: the new symbol is defined to be an alias of the old symbol. Lastly, sets can be defined inductively. Using the *Inductive* keyword, a vertical bar is used to separate the individual rules that specify the form of valid members of the set. The command *Inductive C : Set :=* | *SomeClass : id -> C* informs Coq that we are declaring a new type C that has just one constructor called *SomeClass*, which takes a single argument of type *id* to create a member of the set C.

```
Require Import Arith Bool List.
```

[1] https://github.com/ericmercer/javalite/tree/master/sequential/coq/javalite.ml

```
Require Import FMaps.
Require Import Coq.NArith.BinPos.

(* (id variable-not-otherwise-mentioned) *)
Definition id := positive.

(* (f id) *)
Definition F := id.

(* (m id) *)
Definition M := id.

(* (C Object
     id)
 *)
Inductive C : Set :=
 | SomeClass : id -> C.
```

In the above definition, the Coq production for the non-terminal C does not include a rule for the *Object* class identifier used in Redex. This is because Coq does not allow the use of "free" identifiers like Redex. Excluding an explicit constructor for the *Object* class is not problematic because its only purpose is to indicate the upper boundary of class hierarchies when retrieving fields and class identifiers in the heirarchy of some class, at which point there are no longer any results to return. Later in this section, we shall examine the storage format we use for classes and fields and consequently see that upper bounds can still be determined without defining a separate constructor for the *Object* class. In contrast to the definition of class identifier C, the declaration of Javalite's type T is straightforward and has three constructors shown in the next snippet below. The *T_Class* constructor requires a specific class to be specified. For the *location* production, we used Coq's *nat* type, which represents the natural numbers. The definition of Javalite primitive values and pointers is also shown below.

```
Definition Boolean := bool.

(* (T bool
     unit
     C)
 *)
Inductive T : Set :=
 | T_Bool
 | T_Unit
```

```
 | T_Class : C -> T.

(* (x this
     id)
 *)
Inductive X : Set :=
 | This
 | SomeId : id -> X.

(* (loc number) *)
Definition Location := nat.

(* (pointer (addr loc C)
            null)
 *)
Inductive Pointer : Set :=
 | Addr : Location -> C -> Pointer
 | Null.

(* (v pointer
     true
     false
     unit
     error)
 *)
Inductive V : Set :=
 | V_Pointer : Pointer -> V
 | V_Bool    : Boolean -> V
 | V_Error
 | V_Unit.
```

As mentioned earlier, there is no explicit separation of Javalite into surface and machine syntax in the Coq definitions. Nonetheless, the machine syntax definitions immediately follow the surface syntax definitions in the Coq script. The snippet below shows the definition of Javalite's expressions in Coq. Notice that the *Raw* method invocation, which is defined in the machine syntax in the Redex model, is defined with the other expressions (which were part of the surface syntax in the Redex Model) because Coq does not support a language extension mechanism like PLT Redex's. It is also noteworthy that since Coq does not allow for "free" symbols, the syntactic sugar of Javalite is dropped entirely and only the required terms are declared in Coq. Consider the PLT Redex definition of the variable declaration expression *var T x := e in*

e. Its format includes symbols such as *:=* that increase the readability of the expression. Its Coq equivalent is the constructor *VarDecExp*, whose arguments are a *VariableDeclaration* term and two *Expression* terms. There are no additional syntactic elements just like in an abstract syntax tree, for example.

```
Inductive VariableDeclaration : Set :=
 | VarDec : id -> T -> VariableDeclaration.


(*
  (e x
     v
     (new C)
     (e $ f)
     (e @ m (e ...))
     (raw v @ m (v ...))
     (e == e)
     (C e)
     (e instanceof C)
     (x := e)
     (x $ f := e)
     (if e e else e)
     (var T x := e in e)
     (begin e ...))
*)
Inductive Expression : Set :=
 | Expr_X           : X -> Expression
 | Expr_V           : V -> Expression
 | NewClass         : C -> Expression
 | FieldRef         : Expression -> F -> Expression
 | MethodInvocation : Expression -> M -> list X -> Expression
 | Raw              : Pointer -> M -> list V -> Expression
 | Equality         : Expression -> Expression -> Expression
 | Cast             : C -> Expression -> Expression
 | InstanceOf       : Expression -> C -> Expression
 | VarAssign        : X -> Expression -> Expression
 | FieldAssign      : X -> F -> Expression -> Expression
 | IfExpr           : Expression -> Expression -> Expression -> Expression
 | VarDecExp        : VariableDeclaration -> Expression -> Expression -> Expression
 | VoidExp          : Expression
 | SeqExp           : Expression -> Expression -> Expression.
```

Having defined Javalite's primitive values and expressions, we now present its class and method declarations shown below. Note that the class declaration (CL) uses a *FieldTypeMap* (which is a mapping from a *positive* identifier to a Javalite type T) and a *MethodMap* (which is a mapping from a *positive* identifier to a Javalite *Method* declaration). These mappings simplify the process of querying a class declaration to determine the type of a field or to look up a method declaration by its identifier. We also use a hash map (of class identifiers to class declarations) to define the class declaration list μ in Coq.

```
Definition ArgumentList := list VariableDeclaration.

(* (M (T m ([T x] ...) e))
 *)
Inductive Method : Set :=
 | AMethod : M -> T -> ArgumentList -> Expression -> Method.

Module HashMap := PositiveMap.

Definition FieldTypeMap  := HashMap.t T.
Definition FieldValueMap := HashMap.t V.
Definition MethodMap     := HashMap.t Method.

(* (CL (class C extends C ([T f] ...) (M ...)))
 *)
Inductive CL : Set :=
 | ClassDecl : C -> option CL -> FieldTypeMap -> MethodMap -> CL.

(* (mu (CL ...))
 *)
Definition Mu := HashMap.t CL.

Inductive ProgramEntryPoint : Set :=
 | Entrypoint : C -> M -> ProgramEntryPoint.

(* (P (mu (C m)))
 *)
Inductive P : Set :=
 | Program: Mu -> ProgramEntryPoint -> P.
```

The definitions corresponding to the machine syntax are shown below. A notable change in the Coq definition is that the Javalite heap h is defined as a mapping from identifiers to heap values Hv. This

simplifies heap lookups by reducing them to a predefined hash map lookup operation. We used this approach in Coq because it has libraries that predefine such mapping structures (unlike in PLT Redex where there is no syntactic library already defined to do this). The rest of the definitions follow the patterns we have encountered thus far. Next, we discuss the encoding of the Javalite transition relation in Coq.

```
Definition FieldLocationMap := HashMap.t Location.
Definition ClassToFieldLocationsMap := HashMap.t FieldLocationMap.

(* (object ((C [f loc] ...) ...))
 *)
Inductive HeapObject : Set :=
 | HeapObj : C -> ClassToFieldLocationsMap -> HeapObject.

(* (hv v
       object)
 *)
Inductive Hv : Set :=
 | Hv_v      : V -> Hv
 | Hv_object : HeapObject -> Hv.

(* (h mt
      (h [loc -> hv]))
 *)
Definition H := HashMap.t Hv.

(* (eta mt
      (eta [x -> loc]))
 *)
Inductive Eta : Set :=
 | Eta_mt    : Eta
 | Eta_NotMt : Eta -> X -> Location -> Eta.

(* (k ret
      ( * $ f -> k)
      ( * @ m (e ...) -> k)
      (v @ m (v ...) * (e ...) -> k)
      ( * == e -> k)
      (v == * -> k)
      (C * -> k)
      ( * instanceof C -> k)
      (x := * -> k)
```

```
      (x $ f := * -> k)
      (if * e else e -> k)
      (var T x := * in e -> k)
      (begin * (e ...) -> k)
      (pop eta k))
 *)
Inductive Continuation : Set :=
 | K_Return               : Continuation
 | K_FieldAccess          : F -> Continuation -> Continuation
 | K_MethodInvocation     : M -> list X -> Continuation -> Continuation
 | K_EqualityLeftOperand  : Expression -> Continuation -> Continuation
 | K_EqualityRightOperand : V -> Continuation -> Continuation
 | K_Cast                 : C -> Continuation -> Continuation
 | K_InstanceOf           : C -> Continuation -> Continuation
 | K_VarAssign            : X -> Continuation -> Continuation
 | K_FieldAssign          : X -> F -> Continuation -> Continuation
 | K_If                   : Expression -> Expression -> Continuation -> Continuation
 | K_VarAssignIn          : T -> X -> Expression -> Continuation -> Continuation
 | K_Seq                  : Expression -> Continuation -> Continuation
 | K_Pop                  : Eta -> Continuation -> Continuation.

Inductive State : Set :=
 | StateCons : Mu -> H -> Eta -> Expression -> Continuation -> State.
```

4.3 Reduction Rules in Coq

Variable Access Rule. The Coq proposition that defines Javalite's reduction rules is declared by the syntax *Inductive ExprReduces : State → State → Prop := ...* This syntax gives the proposition the name *ExprReduces* and indicates that *ExprReduces* is a proposition on two states. The ellipses are a placeholder for the individual definitions of the scenarios under which the proposition holds. If s and s' are states in Coq, then the expression *ExprReduces* s s' is the proposition that states that the reduction relation holds for s and s'. In other words, s reduces to s'. Each scenario where the proposition holds is required to have a constructor (which must be uniquely named) in Coq. Each of these constructors is considered to be evidence that the proposition holds. In addition to this, side conditions under which the proposition holds may be specified. Consider the *variable access* code snippet shown below.

```
Inductive ExprReduces : State -> State -> Prop :=
```

```
(* ------------------------------------------------------------------------ *)
(* Variable Access *)

| ER_VariableAccess :
 forall (x:X) (mu:Mu) (h:H) (eta:Eta) (k:Continuation) (l:Location) (hv:Hv),
  (eta_lookup eta x) = Some l ->
  (h_lookup h l)     = Some hv ->
  ExprReduces (StateCons mu h eta (Expr_X x) k)
              (StateCons mu h eta (Expr_V (Hv_To_V l hv)) k)
```

This snippet declares a constructor named *ER_VariableAccess* and specifies that the proposition *ExprReduces s s′* holds where s is the state *(StateCons mu h eta (Expr_X x) k)* and s' is the state *(StateCons mu h eta (Expr_V (Hv_To_V l hv)) k)*. The reduction rules in Coq utilize universal quantification to make the reduction claim precise. Note that there are preconditions that must hold for the proposition to be valid. For this variable access scenario, the control string of the "input" state is the identifier *(Expr_X x)*, which must exist in the local environment η, hence the precondition *((eta_lookup eta x) = Some l)*. The location l on the heap must also be mapped to a value for the variable access reduction to be valid hence the second precondition *((h_lookup h l) = Some hv)*. When implication is used in a set of preconditions as in this scenario, it is equivalent to conjunction of the individual preconditions. The *ER_VariableAccess* constructor is therefore evidence to Coq that, given the preconditions, a state whose control string is an identifier reduces to a new state whose control string is the value of the variable refered to by the identifier. The various helper functions such as *Hv_To_V* are presented in the Coq code listing in Appendix C.

Class Instantiation Rule. The class instantiation expression (*new C*) indicates that a new object needs to be created on the heap. The *new* rule proceeds as follows. It calls the *get_fields_of_parents_and_self_C* function to create a list of fields for each class in the hierarchy of class *C*. *get_fields_of_parents_and_self_C* returns a list of mappings (one per class in the hierarchy) of field identifiers to their types.

```
(* ------------------------------------------------------------------------ *)
(* New *)
(*
(--> (  h eta (new C) k)
     (  h_1 eta (addr loc_1 C) k)
     "new"
     (where (([T_0 f_0] ...) ...) (fields-parents+self   C))
     (where (C_0 ...) (class-parents+self   C))
     (where ((v_0 ...) ...) ((default-value* (T_0 ...)) ...))
```

```
     (where (number_0 ...) ((get-length (T_0 ...)) ...))
     (where ((loc_0 ...) ...) (h-malloc-n* h number_0 ...))
     (where object (C (C_0 [f_0 loc_0] ...) ...))
     (where h_0 (h-extend* h [loc_0 -> v_0] ... ...))
     (where loc_1 (h-malloc h_0))
     (where h_1 (h-extend* h_0 [loc_1 -> object])))
*)
| ER_New :
 forall (mu:Mu) (h h_0 h_1:H) (eta:Eta) (c:C) (loc_1:Location)
        (k                        : Continuation)
        (classlist                : CList)
        (defaultvalues            : list FieldValueMap)
        (hierarchicalfieldlist    : list FieldTypeMap)
        (hierarchicaltypelist     : list FieldTypeMap)
        (hierarchicallocations    : list LocationList)
        (listofclassfieldloclists : ClassToFieldLocationsMap)
        (hierarchicalfieldlocmap  : list FieldLocationMap),

  (classes_of_parents_and_self c mu)                     = Some classlist        ->
  (* ensure 'classlist' is not empty *)
  (beq_nat 0 (length classlist))                         = false                 ->
  (get_fields_of_parents_and_self_C c mu)                = hierarchicalfieldlist ->
  (get_hierarchical_type_map hierarchicalfieldlist)      = hierarchicaltypelist  ->
  (get_hierarchical_default_values hierarchicaltypelist) = defaultvalues         ->

  (h_malloc_n_star h (get_value_lengths defaultvalues))  = hierarchicallocations ->
  (create_hierarchical_field_location_map hierarchicallocations defaultvalues)
        = hierarchicalfieldlocmap  ->
  (h_extend_star_hierarchical_map h hierarchicalfieldlocmap defaultvalues)
        = Some h_0 ->

  (h_malloc h_0)                                                                 = loc_1 ->
  (build_class_loc_lists classlist hierarchicalfieldlist hierarchicallocations)
        = listofclassfieldloclists ->
  (h_extend h_0 loc_1 (Hv_object (HeapObj listofclassfieldloclists)))            = h_1 ->

  ExprReduces (StateCons mu h eta (NewClass c) k)
              (StateCons mu h_1 eta (convert_pointer_to_expr (Addr loc_1 c)) k).
```

The length of the field list of each class is needed in order to allocate heap locations for each field of the object. These lengths of the class fields are computed by the *get_value_lengths* function and passed to the *h_malloc_n_star* function, which allocates enough heap locations for all the fields in the hierarchy of the object. Since these locations must be initialized with their default values, the class instantiation rule uses the *get_hierarchical_default_values* function to create a hierarchial list of the default values of each of the fields in the new object's class hierarchy. Recall that the storage format of an object on the heap is defined by a hierarchical list of mappings from field identifiers to field locations. A list of the actual class identifiers in the hierarchy is therefore required and is created by the call to the *classes_of_parents_and_self* function. The object to be stored on the heap is then created by the expression *(Hv_object (HeapObj c listofclassfieldloclists))*. The term *listofclassfieldloclists* is created by the *build_class_loc_lists* function, which groups all the fields by their class identifiers and maps each of the fields to its allocated heap location.

Note that the Javalite heap allocation functions return new unused heap locations. They do not alter the passed in heap. A new heap *h_0* is therefore explicitly created by extending the input heap with mappings from each of the allocated field locations to the default values of the corresponding fields. The reduction rule also calls *h_malloc* on the newly created heap *h_0* to allocate a new location *loc_1* to point to the newly created object. It then extends heap *h_0* to include a mapping from the new location *loc_1* to the newly created object. The reduction rule finally creates a new state using the new heap *h_1* and a new (pointer) expression (from the term (*Addr* loc_1 *c*)), which contains the location of the newly created object.

Field Access Rule. The evaluation of the receiving object *e* of a field access expression *FieldRef e f* is represented by the *ER_FieldAccess1* constructor shown below.

```
(* ------------------------------------------------------------------------ *)
(* Field Access - Object Eval *)

| ER_FieldAccess1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (f:F) (k:Continuation),
  ExprReduces (StateCons mu h eta (FieldRef e f) k)
              (StateCons mu h eta e (K_FieldAccess f k))

(* Field access *)

| ER_FieldAccess2 :
 forall mu h eta loc C f1 k v1 object obj loc_0,
  (h_lookup h loc)              = Some (Hv_object obj) ->
  (cast obj C)                  = Some object          ->
```

```
   (field_lookup object C f1 mu) = Some loc_0           ->
   (h_lookup h loc_0)            = Some (Hv_v v1)       ->
   ExprReduces (StateCons mu h eta (Expr_V (V_Pointer (Addr loc C))) (K_FieldAccess f1 k))
               (StateCons mu h eta (Expr_V v1) k)
```

Once e has been evaluated to a pointer expression *(Expr_v (V_Pointer (Addr loc C)))*, the state's field access continuation *(K_FieldAccess f1 k)* indicates that the field $f1$ is to be accessed. The *ER_FieldAccess2* reduction rule requires that the object at the location *loc* on the heap (obtained by calling *h_lookup*) is a heap object. The reduction also specifies the precondition that casting the retrieved object to an instance of class C yields a valid object and that the value of the field f is stored at a valid heap location loc_0 (as determined by the *field-lookup* function). In addition to this, calling *h-lookup* to retrieve the value of field f must yield a valid heap value. As explained in Chapter 3, field look-ups in PLT Redex are performed by pattern matching on field lists. In the Coq environment, the field look-ups are implemented as a function whose arguments include the field to look up as well as the heap object on which to retrieve the field. The Coq definition of the *field_lookup* function is shown below. It calls the *hierarchical_field_lookup* function, which obtains the hierarchy of class declarations for the class on which the field is being looked up (using the *get_reversed_class_hierarchy_CL_to_CLList* function). By using these functions, the field shadowing behavior of Java is retained by the model.

```
Definition hierarchical_field_lookup (f:F) (c:C) (c2flm:ClassToFieldLocationsMap)
                                     (mu:Mu) : option Location :=
 match (convert_C_to_CL c mu) with
 | None    => None
 | Some cl => match get_reversed_class_hierarchy_CL_to_CLList
                                (HashMap.cardinal c2flm) cl mu with
              | nil   => None
              | cl::t => (hierarchical_field_lookup_from_list f (cl::t) c2flm)
              end
 end.

Definition field_lookup (object:HeapObject) (c:C) (f:F) (mu:Mu)
 : option Location :=
 match object with
 | HeapObj c2flm => (hierarchical_field_lookup f c c2flm mu)
 end.
```

The *ER_FieldAccess2* reduction creates a new state whose control string is the accessed value v and restores the continuation k.

Method Invocation Rules. The *ER_MethodInvocationObjectEval* reduction rule evaluates the receiving object expression e_0 in a method invocation expression *(MethodInvocation e_0 m args)*.

```
(* ------------------------------------------------------------------------ *)
(* Method invocation - object eval *)

| ER_MethodInvocationObjectEval :
 forall (mu:Mu) (h:H) (eta:Eta) (e_0:Expression) (args:list X) (m:M) (k:Continuation),
  ExprReduces (StateCons mu h eta (MethodInvocation e_0 m args) k)
              (StateCons mu h eta e_0 (K_MethodInvocation m args k))

(* Method invocation *)

| ER_MethodInvocation :
 forall (mu:Mu) (h:H) (eta:Eta) (pv_o:Pointer) (m:M) (k:Continuation) (args:list X)
  (primitive_args:list V),
  (lookup_arguments h eta args) = primitive_args ->
  ExprReduces (StateCons mu h eta (Expr_V (V_Pointer pv_o)) (K_MethodInvocation m args k))
              (StateCons mu h eta (Raw pv_o m primitive_args) k)
```

Once the method invocation's receiving expression e_0 has been evaluated to a pointer pv_o, the *ER_MethodInvocation* constructor requires that the list of argument identifiers *args* in the method invocation continuation *(K_MethodInvocation m args k)* be converted into a list of primitive values (by calling the *lookup_arguments* function). At this point, the method invocation can be represented as a *raw method invocation* expression. In a raw method invocation, all the argument expressions have been evaluated to primitive values.

Raw Method Invocation. A raw method invocation expression is comprised of a pointer, a method identifier, and a list of primitive values (arguments). In order to syntactically evaluate a raw method invocation, we need to perform a virtual method lookup to determine the appropriate class on which the method should be invoked. The prerequisites of the evaluation are specified by the $ER\_MethodInvocationRaw$ constructor. The first requirement is a list of class declarations in the hierarchy of the class C on which the method is being invoked. This list is created by retrieving the object on which the call is being invoked (using the *h_lookup* function) and passing this object to the *class_decls_of_parents_and_self* function.

```
| ER_MethodInvocationRaw :
 forall (mu:Mu) (h h_0 h_tmp:H) (eta eta_0:Eta) (e_m:Expression) (k:Continuation)
        (m:M) (varlist:list V) (c C_t:C) (loc loc_o:Location)
```

```
        (loclist:LocationList) (methodvars:IdList) (obj1:HeapObject)
        (arglist:ArgumentList) (t:T)
        (classlist:CLList) (CL_t:CL),
   (h_lookup h loc)                        = Some (Hv_object obj1) ->
   (class_decls_of_parents_and_self c mu) = Some classlist        ->

   (get_class_with_virtual_method m classlist) = Some CL_t ->
   (convert_CL_to_C CL_t)                      = C_t       ->
   (method_lookup m CL_t)                      = Some (AMethod m t arglist e_m) ->
   (argument_list_to_XList arglist)            = methodvars ->

   (* allocate locations for "this" and the arguments *)
   (h_malloc_n h (S (length varlist)))         = (loc_o::loclist) ->

   (* write "this" and the args into the heap.
      loclist and varlist must be the same length.
    *)
   (h_extend h loc_o (make_heap_pointer loc C_t)) = h_tmp   ->
   (h_extend_star h_tmp loclist varlist)          = Some h_0 ->

   (* create a new local environment with the bindings for "this" and args *)
   (eta_extend_star (eta_extend eta This loc_o) methodvars loclist) = Some eta_0 ->

   ExprReduces (StateCons mu h eta (Raw (Addr loc c) m varlist) k)
               (StateCons mu h_0 eta_0 e_m (K_Pop eta k))
```

The virtual method lookup is then performed by calling the *get_class_with_virtual_method* function. This function is shown below. Its arguments are the method identifier to look up (m) and the class hierarchy (subclass to superclass) in which to search (CL). It traverses the class hierarchy from the first listed class to the last, looking up the class declarations for the specified method and returning the first class in the hierarchy whose method hashmap contains a mapping for the requested method.

```
Definition method_lookup (m:M) (cl:CL) : option Method :=
                         HashMap.find m (get_method_list cl).

Fixpoint get_class_with_virtual_method (m:M) (classlist:list CL) : option CL :=
 match classlist with
 | nil   => None
 | cl::t => match method_lookup m cl with
            | Some method => Some cl
```

```
            | None        => get_class_with_virtual_method m t
            end
end.
```

See Appendix C for the full source listing of the functions used in the model. Calling *method_lookup* returns the body expression of the method being invoked as well as the argument list *arglist* and return type t of the method if the specified method is defined in the class declaration CL_t. Otherwise, *method_lookup* returns the term $None$. Now that the exact method being invoked is known, we need to allocate heap locations for the arguments to the method and the reference *this* (recall that all variables, fields, and arguments are stored on the heap). The number of locations for the arguments and the object *this* is computed by the expression (S (*length varlist*)) and is passed to the *h_malloc_n* function, which returns a list of new heap locations for the respective arguments. The first allocated location *loc_o* is mapped to the object (constructed by the expression *(make_heap_pointer loc C_t)*) on which the invocation is being performed (i.e. the reference "this"). Notice that the constructed pointer stored at *loc_o* points to the same location as the receiving object of the raw method invocation but has the appropriate type determined by virtual method resolution. The rest of the locations (*loclist*) are mapped to the method's arguments (*varlist*) by a call to the *h_extend_star* function which returns a new heap h_0.

A new local environment is also created by the expression *(eta_extend eta This loc_o)*, which extends the initial environment *eta* with a binding mapping the reference *this* to the location *loc_o*. The final local environment *eta_0* then extends the aforementioned local environment to include mappings of the method argument names to the heap locations where their values are stored. The reduction rule then creates a new state containing the new heap h_0 and the new local environment η_0. The $ER\_MethodInvocationRaw$ rule finally creates a new state whose control string is the expression e_m, which represents the body of the method being invoked. The continuation ($K\_Pop\ eta\ k$) of the new state will restore the pre-invocation local environment and continuation (η and k respectively) after the method invocation evaluation is complete.

Equals Rules. As expected, the equality testing expression *(Equality e_0 e)* requires the evaluation of the expressions on both sides of the equality. The $ER\_Equals1$ rule shown below initiates the evaluation of the left expression e_0 to a value v. The $ER\_Equals2$ rule then creates a new continuation with the value v and initiates the evaluation of the right hand side expression e. Once both expressions have been evaluated to primitive values, the *ER_Equals3* rule uses the *convert_to_boolean_expr* function to compute the Javalite boolean result of the comparison.

```
(* ------------------------------------------------------------------------ *)
```

```
(* Equals '==': l-operand eval *)

| ER_Equals1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e_0 e:Expression) (k:Continuation),
  ExprReduces (StateCons mu h eta (Equality e_0 e) k)
              (StateCons mu h eta e_0 (K_EqualityLeftOperand e k))

(* Equals '==': r-operand eval *)

| ER_Equals2 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (v:V) (k:Continuation),
  ExprReduces (StateCons mu h eta (Expr_V v) (K_EqualityLeftOperand e k))
              (StateCons mu h eta e (K_EqualityRightOperand v k))

(* Equals '==': equals *)

| ER_Equals3 :
 forall (mu:Mu) (h:H) (eta:Eta) (v_0 v_1:V) (k:Continuation),
  ExprReduces (StateCons mu h eta (Expr_V v_0) (K_EqualityRightOperand v_1 k))
              (StateCons mu h eta (convert_to_boolean_expr (V_equals v_0 v_1)) k)
```

Typecast Rules. The evaluation of a type cast expression (C e) is initiated by evaluating the expression e to a pointer value. Recall that Javalite pointers are typed and that the heap storage format for an object includes all the classes and their fields in the hierarchy of an object. Once the expression e is evaluated to some pointer value (*Addr loc C_0*), the *ER_Typecast2* rule specifies the preconditions for the casting reduction to succeed. First, retrieving the object refered to by the pointer (using the *h_lookup* function) must yield a valid object. Second, the cast operation must be valid, i.e. class C_1 to which we are casting must be in the hierarchy of the pointer's current class C_0. The *can_cast* function returns a boolean value indicating whether the cast operation can be performed. When these preconditions are met, the *ER_Typecast2* reduction rule creates a new state whose control string is a pointer to the same heap location *loc* but of the type C_1, completing the cast operation.

```
(* ------------------------------------------------------------------- *)
(* Typecast - Object eval *)

| ER_Typecast1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (c:C) (k:Continuation),
  ExprReduces (StateCons mu h eta (Cast c e) k)
```

```
              (StateCons mu h eta e (K_Cast c k))

(* Typecast *)

| ER_Typecast2 :
 forall (mu:Mu) (h:H) (eta:Eta) (c_0 c_1:C) (loc:Location) (object:HeapObject)
  (k:Continuation),
  (h_lookup h loc)      = Some (Hv_object object) ->
  (can_cast object c_1) = true ->
  ExprReduces (StateCons mu h eta (convert_pointer_to_expr (Addr loc c_0)) (K_Cast c_1 k))
              (StateCons mu h eta (convert_pointer_to_expr (Addr loc c_1)) k)
```

Instanceof Rules. An instance-of expression (*e instanceof C*) evaluates to a Javalite boolean value. The *object eval* rule starts the evaluation of *e*. Should it evaluate to a pointer value (*Addr loc C_0*), then the *ER_Instanceof2* rule specifies that the location *loc* on the heap must be mapped to a valid heap object. This is verified by the call to the *h_lookup* function. The boolean result of whether the the object at location *loc* can be cast to the type C_1 specified by the instance-of continuation is computed by the *can_cast* function.

```
(* ------------------------------------------------------------------------ *)
(* Instanceof - object eval *)

| ER_InstanceOf1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (c:C) (k:Continuation),
  ExprReduces (StateCons mu h eta (InstanceOf e c) k)
              (StateCons mu h eta e (K_InstanceOf c k))

(* Instanceof *)

| ER_InstanceOf2 :
 forall (mu:Mu) (h:H) (eta:Eta) (c_0 c_1:C) (v_res:Boolean) (k:Continuation)
        (loc:Location) (object:HeapObject),
  (h_lookup h loc)      = Some (Hv_object object) ->
  (can_cast object c_1) = v_res ->
  ExprReduces (StateCons mu h eta (convert_pointer_to_expr (Addr loc c_0))
                                  (K_InstanceOf c_1 k))
              (StateCons mu h eta (convert_to_boolean_expr v_res) k)
```

Variable Assignment Rules. In order to evaluate a variable assignment expression *(VarAssign x e)*, the expression *e* is evaluated to some value *v* and then the location of the variable on the heap is determined

in order to update the heap with the new value v being assigned. The *ER_Assign2* rule determines the location of variable x using the *eta_lookup* function and then updates the heap h with a call to the *h_extend* function which creates a mapping $[loc \rightarrow v]$ from the resultant location *loc* to the value v.

```
(* ------------------------------------------------------------------------ *)
(* Assign - Object eval *)

| ER_Assign1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (x:X) (k:Continuation),
  ExprReduces (StateCons mu h eta (VarAssign x e) k)
              (StateCons mu h eta e (K_VarAssign x k))

(* Assign *)

| ER_Assign2 :
 forall (mu:Mu) (h h_0:H) (eta:Eta) (v:V) (loc:Location) (x:X) (k:Continuation),
  (eta_lookup eta x)        = Some loc ->
  (h_extend h loc (Hv_v v)) = h_0 ->
  ExprReduces (StateCons mu h eta (Expr_V v) (K_VarAssign x k))
              (StateCons mu h_0 eta (Expr_V v) k)
```

Field Assignment Rules. The evaluation of a field assignment expression *(FieldAssign x f e)* is similar to the variable assignment evaluation. To complete the assignment, we need to determine the location of the field f on the heap. The object referred to by x has this information. An *eta_lookup* call returns the heap location loc_0 of the pointer x.

```
(* ------------------------------------------------------------------------ *)
(* Assign Field - Object eval *)

| ER_AssignField1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (x:X) (f:F) (k:Continuation),
  ExprReduces (StateCons mu h eta (FieldAssign x f e) k)
              (StateCons mu h eta e (K_FieldAssign x f k))

(* Assign Field *)

| ER_AssignField2 :
 forall (mu:Mu) (h h_0:H) (eta:Eta) (x:X) (f:F) (c:C) (v:V)
        (loc_0 loc_1 loc_2:Location) (obj object:HeapObject) (k:Continuation),
```

```
  (eta_lookup eta x)                = Some loc_0  ->
  (h_lookup h loc_0)                = Some (Hv_v (V_Pointer (Addr loc_1 c))) ->
  (h_lookup h loc_1)                = Some (Hv_object obj) ->
  (cast obj c)                      = Some object ->
  (field_lookup object c f mu)      = Some loc_2  ->
  (h_extend h loc_2 (Hv_v v))       = h_0 ->
  ExprReduces (StateCons mu h eta (Expr_V v) (K_FieldAssign x f k))
              (StateCons mu h_0 eta (Expr_V v) k)
```

The actual value (*addr loc_1 C*) of the pointer is retrieved by passing *loc_0* to the heap lookup function *h_lookup*, at which point we can look up the object referred to by x by passing *loc_1* to *h_lookup*. The object is cast to an instance of C to match the typed pointer x thus ensuring that the right field will be used for the assignment. The location *loc_2* of the field f is returned by the *field_lookup* function. The *ER_AssignField* rule finally extends the heap h with a mapping [*loc_2* $\rightarrow$ v] of the field f's location to the computed value v of the expression being assigned. The resultant state therefore has the newly updated heap h_0.

If-Then-Else Rules. The evaluation of the predicate e_p of a branching expression ($if\ e_p\ e_t\ else\ e_f$) is initiated by the *if-then-else – object eval* rule shown below. Once the predicate e_p has been evaluated, its value $v1$ is used by the *eval_if_then_else* function to select one of the expressions e_t and e_f for the new state's control string.

```
(* ------------------------------------------------------------------------ *)
(* If-then-else - object eval *)

| ER_IfThenElseObjectEval :
 forall (mu:Mu) (h:H) (eta:Eta) (e_p e_t e_f:Expression) (k:Continuation),
  ExprReduces (StateCons mu h eta (IfExpr e_p e_t e_f) k)
              (StateCons mu h eta e_p (K_If e_t e_f k))

(* If-then-else *)

| ER_IfThenElse :
 forall (mu:Mu) (h:H) (eta:Eta) (v1:Boolean) (e_t e_f:Expression) (k:Continuation),
  ExprReduces (StateCons mu h eta (convert_to_boolean_expr v1) (K_If e_t e_f k))
              (StateCons mu h eta (eval_if_then_else v1 e_t e_f) k)
```

Variable Declaration Rules. A variable declaration expression *(VarDecExp (VarDec x1 t) e_0 e_1)* evaluates the expression e_0 and assigns its value to the newly declared variable $x1$ and uses the resultant environment to evaluate the expression e_1. In other words, e_1 is the scope of the variable $x1$. Once e_0 has been evaluated to a value v, the *ER_VarDec2* rule performs the variable assignment by allocating an unused location loc_x for the variable x on the heap with a call to the *h_malloc* function.

```
(* ------------------------------------------------------------------------ *)
(* Variable declaration - object eval *)

| ER_VarDec1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e_0 e_1:Expression) (x1:id) (t:T) (k:Continuation),
  ExprReduces (StateCons mu h eta (VarDecExp (VarDec x1 t) e_0 e_1) k)
              (StateCons mu h eta e_0 (K_VarAssignIn t (SomeId x1) e_1 k))

(* Variable declaration *)

| ER_VarDec2 :
 forall (mu:Mu) (h h_0:H) (eta eta_0:Eta) (v:V) (e_1:Expression) (x1:id) (t:T)
  (k:Continuation) (loc_x:Location),
  (h_malloc h)                            = loc_x ->
  (h_extend h loc_x (Hv_v v))             = h_0   ->
  (eta_extend eta (SomeId x1) loc_x)      = eta_0 ->
  ExprReduces (StateCons mu h eta (Expr_V v) (K_VarAssignIn t (SomeId x1) e_1 k))
              (StateCons mu h_0 eta_0 e_1 (K_Pop eta k))
```

It then extends the heap h with a mapping from the new location to the value v of the variable yielding a new heap h_0. The reduction rule then creates an entry for the new variable x in the local environmnent by adding the mapping $[x \rightarrow loc\_x]$ to it. The scope of the newly declared variable is the expression e_1, which is then evaluated next (and therefore becomes the new control string). After the evaluation of e_1, the variable x will no longer be valid. To ensure that the previous local environment is restored, the *ER_VarDec2* rule creates a pop continuation (*K_Pop eta k*) which will restore the previous local environment *eta* and continuation k after the evaluation of e_1.

Begin Expression Rules. A Javalite begin expression is a sequence of any of Javalite's expressions. Sequences of expressions are evaluated in left-to-right order. The simplest case is the empty expression list $VoidExp$. The *ER_BeginEmptyExpList* rule simply rewrites the $VoidExp$ control string into the Javalite *V_Unit* value. Non-empty expression lists are expressed by chaining *SeqExp*'s together. This pattern is used

to prevent the use of mutually recursive data types in the definition of expressions in Coq. The expressions e_0 and e_1 in the sequence expression ($SeqExp\ e\_0\ e\_1$) can be any expressions, including sequence expressions. Their evaluation is handled by the *ER_Begin_e_0_evaluation* and *ER_Begin_e_i_evaluation* reduction rules shown below.

The *begin-e_0-evaluation* rule initiates the evaluation of the first expression e_0 and creates a *begin* continuation of the form ($begin\ *\ (e_1\ ...)$), which indicates that the rest of the expressions (e_1 ...) will need to be evaluated before the original continuation k can be considered.

```
(* ------------------------------------------------------------------------ *)
(* Begin - Empty expression list *)

| ER_BeginEmptyExpList :
 forall (mu:Mu) (h:H) (eta:Eta) (k:Continuation),
  ExprReduces (StateCons mu h eta VoidExp k)
              (StateCons mu h eta (Expr_V V_Unit) k)

(* Begin - e_0 evaluation *)

| ER_Begin_e_0_evaluation :
 forall (mu:Mu) (h:H) (eta:Eta) (k:Continuation) (e_0 e_1:Expression),
  ExprReduces (StateCons mu h eta (SeqExp e_0 e_1) k)
              (StateCons mu h eta e_0 (K_Seq e_1 k))

(* Begin - e_1 evaluation *)

| ER_Begin_e_1_evaluation :
 forall (mu:Mu) (h:H) (eta:Eta) (v:V) (k:Continuation) (e_1:Expression),
  ExprReduces (StateCons mu h eta (Expr_V v) (K_Seq e_1 k))
              (StateCons mu h eta e_1 k)
```

Pop Rule. The *ER_PopEta* reduction rule creates a new state by replacing the local environment η in the initial state with the environment eta_0 encoded in the *pop* continuation. It also restores the continuation k. This rule is therefore suitable for cleaning up after a method call (by restoring the caller's environment and continuation) or variable declaration.

```
(* ------------------------------------------------------------------------ *)
(* Pop \u03b7 (close scope) *)
```

```
| ER_PopEta :
 forall (mu:Mu) (h:H) (eta eta_0:Eta) (v:V) (k:Continuation),
  ExprReduces (StateCons mu h eta (Expr_V v) (K_Pop eta_0 k))
              (StateCons mu h eta_0 (Expr_V v) k)
```

Now that we have examined the formal definition of the reduction relation, let us discuss some of the proofs about the reduction.

4.4 Proofs About the Javalite Reduction Relation

4.4.1 Decidability of Next State

As mentioned earlier, a proposition in Coq is simply a statement making a claim. It can therefore be either provable or unprovable. *2 + 2 = 5* and *2 + 2 = 4* are both valid propositions in Coq. Since we are using the proposition (*ExprReduces*) to represent the evaluation of Javalite programs, the Coq proposition $ExprReduces\ s\ s'$ represents the claim that there is a Javalite reduction rule that evolves (reduces) state s to state s'. In order to reason about Javalite's operational semantics, the decidability of propositions such as $ExprReduces\ s\ s'$ is paramount. A key component of the ability to decide whether such a proposition is provable (that it holds) is the ability to determine whether or not a state has a successor i.e. another state to which it can be reduced using one of the defined rules. Therefore, we have proved Theorem 1, which states that any given state s either has some successor s' or has no successors at all.

Theorem 1 (ExprReduces_dec) *Let s be a program state. Then*

$$(\exists\ s'\ s.t.\ ExprReduces(s,\ s')) \ \oplus\ (\forall\ s'.\ \neg ExprReduces(s, s')).$$

We created a constructive proof of this theorem in Coq. This means that for scenarios where a successor state s' exists for the state s, we constructed such a state as evidence of the left clause.

Proof Sketch The proof proceeds by induction on each of the possible control strings of the state s. For each of these control strings, the proof performs one of two actions. It constructs a new state s' to which the state s reduces and provides the corresponding constructor name to show that $ExprReduces\ s\ s'$ (recall that constructors are considered evidence that a given proposition holds). Alternatively, it shows that each of the reduction rules cannot apply to the given expression in the initial state. The Coq proof consists of 383 lines of code and completes in 19 seconds. The complete proof listing is presented in Appendix E.

4.4.2 Determinism of the Reduction Relation

Having proved Theorem 1, we now know that it is possible to determine whether or not a state has successors. However, if a state does have some sucessor, Theorem 1 does not specify how many successors it has. It only guarantees the existence of at least one. When studying the evaluation semantics of a language, it is important to know whether or not the *ExprReduces* evaluation relation is function-like, i.e. whether it maps each element to at most one element.

Theorem 2 (ExprReduces_fun) *Let* $s,\ s',$ *and* s''' *be program states. Then*

$$\forall\ s\ s'\ s''.\ [ExprReduces(s,\ s')\ \wedge\ ExprReduces(s, s'')] \rightarrow s' = s''.$$

Proof Sketch Assume that state s reduces to both s' and s''. There are 26 reduction rules in the *ExprReduces* proposition. Therefore, there are 26 possible rules that can lead to $ExprReduces\ s\ s'$. For each of these rules, there are 26 ways in which the proposition $ExprReduces\ s\ s''$ could hold. Therefore, there are 676 (26^2) scenarios to examine. Some of these rules are not defined by the *ExprReduces* proposition and are therefore dropped as contradictory cases. Each of the remaining possible reductions can then be shown to be defined by the *ExprReduces* proposition. The corresponding Coq proof was originally 333 lines long due to the repetitive application of the described technique. However, we reduced it to 26 lines long by automating the examination of each of the 676 scenarios and the resulting code ran in 19 seconds.

4.4.3 Irreflexivity of the Transition Relation

At this point, Theorems 1 and 2 assert that the Javalite reduction relation is a decidable and deterministic relation. When studying the Javalite language with this insight, a natural question to address is whether the evaluation relation is reflexive. If it is, then this would imply that there exists some trivial (and therefore unnecessary) reduction rule (since such a rule does not perform any meaningful tranformation in terms of furthering program evaluation). Moreover, a reflexive relation would also imply that evaluation could get stuck in an infinite loop even in programs without any looping constructs. Therefore, we proved Theorem 3, which states that the reduction relation is not reflexive, i.e. that a state cannot reduce to itself. This guarantees that the expression reduction cannot get stuck in a trivial infinite loop in which a state is reducing to itself over and over again.

Theorem 3 (ExprReduces_not_reflexive) *Let* s *represent a program state. Then*

$$\forall\ s.\ \neg\ ExprReduces(s,\ s)$$

Proof Sketch This proof proceeds by contradiction. We assume that the proposition $ExprReduces(s\ s)$ holds. We then proceed by induction on the expression component of the state s. This generates subcases of the proof for each of the different types of declared expressions. Based on the rules defined by the *ExprReduces* proposition, we then show that for each expression, the proposition $ExprReduces(s\ s)$ is a contradiction since the definition of the *ExprReduces* relation has no expressions that reduce to themselves. This Coq proof is 44 lines long and ran in 6 seconds.

4.5 Conclusion

The Coq theorem prover provides a rigorous environment for defining models and proving properties about them. As a result of its strict type checker, we were able to detect some inconsistent definitions in the PLT Redex model that were not discovered by the unit or system tests. Defining Javalite in Coq enabled us to create machine checked proofs of the decidability and determinism of Javalite and potentially allows the research and development of a veriety of algorithms and tools in this formal environment. The formal model of Java presented in this chapter is a solid foundation for the development of formal models of model checking algorithms as well as the writing of machine verified proofs about the properties of these algorithms. In addition to this, the interpreter extracted from the decidability proof of the Coq Javalite model is much faster than PLT Redex. Our methodology of arriving at a formal model involved rapid prototyping of the model in PLT Redex, porting the model to Coq, proving its decidability, determinism, and irreflexivity, and finally extracting a fast interpreter for the formal Javalite model.

Chapter 5

Conclusion

Our objective in this work was to create a model of Java that would support rapid prototyping of program analysis algorithms such as model checking algorithms, while enabling the writing of rigorous proofs about these algorithms. We created the model in PLT Redex to enable fast prototyping and ported it into the Coq theorem prover to support the creation of formal proofs about the model. In addition to this, we have proved that our Javalite reduction relation is a decidable and deterministic relation. Having created our formal model of the Java langauge, it is now possible to research various algorithms such as static analysis and code transformation algorithms on a larger subset of Java programs and obtain concise results. However, there are still some avenues of future work that remain such as prototyping generalized symbolic execution and proving it to be sound and complete.

In addition to this, PLT Redex model will need to be updated to use ANF like the Coq model. Once this is done, the ability to execute Javalite programs using the interpreter extracted from the proof of Theorem 1 will complement the currently existing test-suite. All the Redex tests could then be executed and verified to generate the same results in both interpreters. We have an initial Redex-to-Coq conversion program that accounts for ANF but not for the exact types required by the method.

Next, our current Java model does not support exceptions. Nonetheless, nothing precludes the addition of exceptions to Javalite. It would require a new continuation for "catch" clauses. This continuation could be scanned by newly added reduction rules to determine which expression should handle the exception, at which point the continuation could be truncated. The surface syntax would need to be updated as well to support declaring, catching, and throwing exceptions.

In today's world of ubiquituous multiprocessors, research on concurrency models and features has become even more important. To facilitate such research, future work also involves adding concurrency primitives such as threads and locking to Javalite. This will facilitate research on the impact of various scheduling techniques and enable researchers to prove properties of schedulers. There are various progress properties of the scheduler in (Rungta and Mercer [2010]) that still need to be proved. A concurrent version Javalite would be well tailored to the study of these properties since it is a large enough subset of Java.

It is also noteworthy that the Javalite model we have presented does not include a type checker. This places the burden of type checking Javalite programs on their authors. A separate type checker could be created for both the Redex and Coq models. In addition to this, the Javalite languauge could be updated in Coq to allow for the creation of only valid Javalite programs. As an example, the Coq model's *if* expression has an *Expression* predicate like in the Redex model. The language could be rewritten to force if expressions to use boolean values only since Coq would type check such a construction and prevent the creation of a program that clearly doesn't type check. Implementing a type-checker to eliminate these issues remains future work. Similarly, there is no explicit proof showing that the Javalite reduction relation maintains the type soundness property of a program. Proving this property of the relation is dependent on implementing the idea of a type checker and therefore remains future work as well.

We found prototyping our model in PLT Redex to be an efficient way of creating a working and executable model. It is easy to extensively test, debug, and refine the model in Redex. Having done so, the conversion of the model into the Coq formal proof assistant was a straightforward process because many bugs in the model were detected and fixed in the PLT Redex environment. As a lesson for tool developers, the authors of (Klein et al. [2012]) explain that tests complement proofs. Their assessment was based on their examination of theorems presented in five papers which were false as stated. Our PLT Redex model is executable and allowed us to test the model before porting it to Coq. In addition to this, the strict type-checking of the Coq theorem prover also illuminated some bugs in the Redex model such as in the *variable access* reduction rule highlighted in the previous chapter. We have therefore found that the lightweight mechanization provided by PLT Redex and the formal environment provided by Coq are complementary properties of these tools. In conclusion, we believe that the Javalite model we have presented is a more suitable model than current alternatives for rapid prototyping of algorithms, properties about which can then be formally proved using the Coq implementation of the model.

Appendix A

Javalite Redex Meta-Functions

```
(define-metafunction javalite
  get-length : (any ...) -> number
  [(get-length (any_0 ...))
   ,(length (term (any_0 ...)))])

(define-metafunction javalite
  default-value : T -> v
  [(default-value bool)
   false]
  [(default-value unit)
   unit]
  [(default-value C)
   null])

(define-metafunction javalite
  default-value* : (T ...) -> (v ...)
  [(default-value* ())
   ()]
  [(default-value* (T_0 T_1 ...))
   ((default-value T_0) (default-value T_1) ...)])

(define-metafunction javalite
  h-max : h -> number
  [(h-max mt) -1]
  [(h-max (h [loc -> hv]))
   ,(max (term loc) (term (h-max h)))])

(define-metafunction javalite
  h-malloc : h -> number
  [(h-malloc h)
   ,(add1 (term (h-max h)))])
```

```
(define-metafunction javalite
  h-malloc-n-helper : number number -> (loc ...)
  [(h-malloc-n-helper number_b number_c)
   ,(let ([z (term number_b)]) (build-list (term number_c) (lambda (y) (+ y z))))])

(define-metafunction javalite
  h-malloc-n : h number -> (loc ...)
  [(h-malloc-n h number)
   (loc_0 ...)
   (where ((loc_0 ...)) (h-malloc-n* h number))])

(define-metafunction javalite
  internal-h-malloc-n* : number (number ...) -> (number (loc ...) ...)
  [(internal-h-malloc-n* number_b (number_0))
   (number_t (loc_1 ...))
   (where (loc_1 ...) (h-malloc-n-helper number_b number_0))
   (where number_t ,(if (empty? (term (loc_1 ...)))
                        (term number_b)
                        (add1 (apply max (term (loc_1 ...))))))]
  [(internal-h-malloc-n* number_b (number_0 number_1 number_2 ...))
   (number_t (loc_0 ...) (loc_1 ...) ...)
   (where (loc_0 ...) (h-malloc-n-helper number_b number_0))
   (where number_i ,(if (empty? (term (loc_0 ...)))
                        (term number_b)
                        (add1 (apply max (term (loc_0 ...))))))
   (where (number_t (loc_1 ...) ...)
          (internal-h-malloc-n* number_i (number_1 number_2 ...)))])

(define-metafunction javalite
  h-malloc-n* : h number ... -> ((loc ...) ...)
  [(h-malloc-n* h number_0 ...)
   ((loc_0 ...) ...)
   (where (number (loc_0 ...) ...) (internal-h-malloc-n* (h-malloc h) (number_0 ...)))])

(define-metafunction javalite
  storelike-lookup : any any -> any
  [(storelike-lookup mt any_0)
   ,(error 'storelike-loopup "~e␣not␣found␣in␣~e" (term any_0) (term mt))]
  [(storelike-lookup (any_0 [any_t -> any_ans]) any_t)
   any_ans]
  [(storelike-lookup (any_0 [any_k -> any_v]) any_t)
```

```
   (storelike-lookup any_0 any_t)
   (side-condition (not (equal? (term any_k) (term any_t))))])

(define (id-<= a b)
  (string<=? (symbol->string a) (symbol->string b)))
(define (storelike-extend <= storelike k hv)
  (match storelike
    ['mt `(mt [,k -> ,hv])]
    [`(,storelike [,ki -> ,hvi])
     (cond
       [(equal? k ki)
        `(,storelike [,ki -> ,hv])]
       [(<= k ki)
        `(,(storelike-extend <= storelike k hv) [,ki -> ,hvi])]
       [else
        `((,storelike [,ki -> ,hvi]) [,k -> ,hv])])]))

(define (storelike-extend* <= storelike extend*)
  (match extend*
    ['() storelike]
    [`([,k -> ,hv] . ,extend*)
     (storelike-extend* <= (storelike-extend <= storelike k hv) extend*)]))

(define-metafunction javalite
  h-lookup : h loc -> hv
  [(h-lookup h loc)
   (storelike-lookup h loc)])

(define-metafunction javalite
  h-extend* : h [loc -> hv] ... -> h
  [(h-extend* h [loc -> hv] ...)
   ,(storelike-extend* <= (term h) (term ([loc -> hv] ...)))])

(define-metafunction javalite
  ?-lookup : ? x -> loc
  [(?-lookup ? x)
   (storelike-lookup ? x)])

(define-metafunction javalite
  ?-extend* : ? [x -> loc] ... -> ?
  [(?-extend* ? [x -> loc] ...)
```

```
   ,(storelike-extend* id-<= (term ?) (term ([x -> loc] ...)))])

(define-metafunction javalite
  restricted-field-lookup : object f -> loc
  [(restricted-field-lookup (C_c
                 (C_0 [f_0 loc_0] ...) ...
                 (C_t [f_t0 loc_t0] ...
                      [f_target loc_target]
                      [f_t1 loc_t1] ...)
                 (C_1 [f_1 loc_1] ...) ...)
                f_target)
   loc_target
   ;; Allows for redefinition and restricts matching
   ;; to be the most recent definition by current cast.
   (side-condition
    (not (member (term f_target)
                 (term (f_t1 ... f_1 ... ...))))])

(define-metafunction javalite
  field-lookup : object f -> loc
  [(field-lookup object f_target)
   (restricted-field-lookup (restrict-object object) f_target)])

(define-metafunction javalite
  restrict-object : object -> object
  [(restrict-object (C_c (C_0 [f_0 loc_0] ...) ...
                         (C_c [f_c loc_c] ...)
                         (C_1 [f_1 loc_1] ...) ...))
   (C_c (C_0 [f_0 loc_0] ...) ...
        (C_c [f_c loc_c] ...))])

(define-metafunction javalite
  class-name : CL -> C
  [(class-name (class C_t extends C ([T f] ...) (M ...)))
   C_t])

(define-metafunction javalite
  parent-name : CL -> C
  [(parent-name (class C extends C_p ([T f] ...) (M ...)))
   C_p])
```

```
(define-metafunction javalite
  field-list : CL -> ([T f] ...)
  [(field-list (class C extends C_p ([T f] ...) (M ...)))
   ([T f] ...)])

(define-metafunction javalite
  class-list-extend : (C ...) C -> (C ...)
  [(class-list-extend (C_0 ...) C_1)
   (C_0 ... C_1)])

(define-metafunction javalite
  class-lookup :   C -> CL
  [(class-lookup (CL_0 ... CL_1 CL_2 ...) C)
   CL_1
   (side-condition (equal? (term (class-name CL_1)) (term C)))])

(define-metafunction javalite
  class-list-from-object : object -> (C ...)
  [(class-list-from-object (C_0 (C_1 [f_1 loc_1] ...) ...))
   ; Restrict out the current cast -- Object will be first class
   (C_1 ...)])

(define-metafunction javalite
  class-parents+self :   C -> (C ...)
  [(class-parents+self   Object)
   (class-list-extend () Object)]
  ; id retricts out the base case above
  [(class-parents+self   id)
   (class-list-extend (class-parents+self   C_p) id)
   (where CL (class-lookup   id))
   (where C_p (parent-name CL))])

(define-metafunction javalite
  field-lists-extend : (([T f] ...) ...) ([T f] ...) -> (([T f] ...) ...)
  [(field-lists-extend  (([T_0 f_0] ...) ...) ([T_1 f_1] ...))
   (([T_0 f_0] ...) ... ([T_1 f_1] ...))])

(define-metafunction javalite
  fields-parents+self :   C -> (([T f] ...) ...)
  [(fields-parents+self   Object)
   (field-lists-extend () ())]
```

```
  ; id restricts out the base case above
  [(fields-parents+self   id)
   (field-lists-extend (fields-parents+self   C_p) ([T f] ...))
   (where CL (class-lookup   id))
   (where C_p (parent-name CL))
   (where ([T f] ...) (field-list CL))])

(define-metafunction javalite
  method-name : M -> m
  [(method-name (T_0 m ([T_1 x] ...) e))
   m])

(define-metafunction javalite
  method-expression : M -> e
  [(method-expression (T_0 m ([T_1 x] ...) e))
   e])

(define-metafunction javalite
  method-args : M -> (x ...)
  [(method-args (T_0 m ([T_1 x] ...) e))
   (x ...)])

(define-metafunction javalite
  method-lookup : CL m -> any
  [(method-lookup (class C_0 extends C_1 ([T x] ...) (M_0 ... M_t M_1 ...)) m)
   (C_0 (method-args M_t) (method-expression M_t))
   (side-condition (equal? (term (method-name M_t)) (term m)))]
  [(method-lookup (class C_0 extends C_1 ([T x] ...) (M ...)) m)
   error
   (side-condition (equal? (findf (? (i) (equal? (term (method-name ,i)) (term m)))
                                  (term (M ...))) #f))])

(define (->bool v)
    (if v
        'true
        'false))

(define-metafunction javalite
  cast : object C -> object
  [(cast (C_1 (C_2 [f_2 loc_2] ...) ...
              (C_3 [f_3 loc_3] ...)
```

```
              (C_4 [f_4 loc_4] ...) ...) C_3)
   (C_3 (C_2 [f_2 loc_2] ...) ...
        (C_3 [f_3 loc_3] ...)
        (C_4 [f_4 loc_4] ...) ...)])

(define (cast? object C_t)
  (define inner-cast?
    (term-match/single
     javalite
     [(C_1 (C_2 [f_2 loc_2] ...) ...)
      (term (C_1 C_2 ...))]))
  (if (member C_t (inner-cast? object)) #t #f))

(define (cast?/->bool object C_t)
  (->bool (cast? object C_t)))

(define-metafunction javalite
     instanceof* : object C -> v
     [(instanceof* (C_1 (C_2 [f_2 loc_2] ...) ...) C_t)
      ,(->bool (member (term C_t) (term (C_2 ...))))])

(define-metafunction javalite
  inject : P -> state
  [(inject (  (C m)))
   (  mt mt ((new C) @ m ()) ret)])

(define-metafunction javalite
  inject/with-state : state m -> state
  [(inject/with-state (  h ? e k) m)
   (  h ? (e @ m ()) ret)])
```

Appendix B

Javalite Coq Syntax Definitions

```
Require Import Arith Bool List.
Require Import FMaps.
Require Import Coq.NArith.BinPos.

(* (id variable-not-otherwise-mentioned) *)
Definition id := positive.

(* (f id) *)
Definition F := id.

(* (m id) *)
Definition M := id.

(* (C Object
      id)

Drop "Object" from this definition since it is not necessary (it only serves
to mark the upper bounderaries of class hierarchies for field lookups, etc, in
the PLT Redex model.
 *)
Inductive C : Set :=
 | SomeClass : id -> C.

Definition Boolean := bool.

(* (T bool
      unit
      C)
 *)
Inductive T : Set :=
 | T_Bool
 | T_Unit
```

```
 | T_Class: C -> T.

(* (x this
      id)
 *)
Inductive X : Set :=
 | This
 | SomeId : id -> X.

(* (loc number) *)
Definition Location := nat.

(* (pointer (addr loc C)
            null)
 *)
Inductive Pointer : Set :=
 | Addr : Location -> C -> Pointer
 | Null.

(* (v pointer
     true
     false
     unit
     error)
 *)
Inductive V : Set :=
 | V_Pointer : Pointer -> V
 | V_Bool    : Boolean -> V
 | V_Error
 | V_Unit.

Inductive VariableDeclaration : Set :=
 | VarDec : id -> T -> VariableDeclaration.

(*
  (e x
     v
     (new C)
     (e $ f)
     (e @ m (e ...))
     (raw v @ m (v ...))
```

```
     (e == e)
     (C e)
     (e instanceof C)
     (x := e)
     (x $ f := e)
     (if e e else e)
     (var T x := e in e)
     (begin e ...))
*)
Inductive Expression : Set :=
 | Expr_X           : X -> Expression
 | Expr_V           : V -> Expression
 | NewClass         : C -> Expression
 | FieldRef         : Expression -> F -> Expression
 | MethodInvocation : Expression -> M -> list X -> Expression
 | Raw              : Pointer -> M -> list V -> Expression
 | Equality         : Expression -> Expression -> Expression
 | Cast             : C -> Expression -> Expression
 | InstanceOf       : Expression -> C -> Expression
 | VarAssign        : X -> Expression -> Expression
 | FieldAssign      : X -> F -> Expression -> Expression
 | IfExpr           : Expression -> Expression -> Expression -> Expression
 | VarDecExp        : VariableDeclaration -> Expression -> Expression -> Expression
 | VoidExp          : Expression
 | SeqExp           : Expression -> Expression -> Expression.

Definition ArgumentList := list VariableDeclaration.

(* (M (T m ([T x] ...) e))
 *)
Inductive Method : Set :=
 | AMethod : M -> T -> ArgumentList -> Expression -> Method.

Module HashMap := PositiveMap.

Definition FieldTypeMap  := HashMap.t T.
Definition FieldValueMap := HashMap.t V.
Definition MethodMap     := HashMap.t Method.

(* (CL (class C extends C ([T f] ...) (M ...)))
 *)
```

```
Inductive CL : Set :=
 | ClassDecl : C -> option CL -> FieldTypeMap -> MethodMap -> CL.

(* (Mu (CL ...))
 *)
Definition Mu := HashMap.t CL.

Inductive ProgramEntryPoint : Set :=
 | EntryPoint : C -> M -> ProgramEntryPoint.

(* (P (mu (C m)))
 *)
Inductive P : Set :=
 | Program: Mu -> ProgramEntryPoint -> P.

Definition FieldLocationMap := HashMap.t Location.
Definition ClassToFieldLocationsMap := HashMap.t FieldLocationMap.

(* (object ((C [f loc] ...) ...))
 *)
Inductive HeapObject : Set :=
 | HeapObj : ClassToFieldLocationsMap -> HeapObject.

(* (hv v
       object)
 *)
Inductive Hv : Set :=
 | Hv_v      : V -> Hv
 | Hv_object : HeapObject -> Hv.

(* (h mt
      (h [loc -> hv]))
 *)
Definition H := HashMap.t Hv.

(* (eta mt
      (eta [x -> loc]))
 *)
Inductive Eta : Set :=
 | Eta_mt    : Eta
 | Eta_NotMt : Eta -> X -> Location -> Eta.
```

```
(* (k ret
      ( * $ f -> k)
      ( * @ m (e ...) -> k)
      (v @ m (v ...) * (e ...) -> k)
      ( * == e -> k)
      (v == * -> k)
      (C * -> k)
      ( * instanceof C -> k)
      (x := * -> k)
      (x $ f := * -> k)
      (if * e else e -> k)
      (var T x := * in e -> k)
      (begin * (e ...) -> k)
      (pop eta k))
 *)
Inductive Continuation : Set :=
 | K_Return               : Continuation
 | K_FieldAccess          : F -> Continuation -> Continuation
 | K_MethodInvocation     : M -> list X -> Continuation -> Continuation
 | K_EqualityLeftOperand  : Expression -> Continuation -> Continuation
 | K_EqualityRightOperand : V -> Continuation -> Continuation
 | K_Cast                 : C -> Continuation -> Continuation
 | K_InstanceOf           : C -> Continuation -> Continuation
 | K_VarAssign            : X -> Continuation -> Continuation
 | K_FieldAssign          : X -> F -> Continuation -> Continuation
 | K_If                   : Expression -> Expression -> Continuation -> Continuation
 | K_VarAssignIn          : T -> X -> Expression -> Continuation -> Continuation
 | K_Seq                  : Expression -> Continuation -> Continuation
 | K_Pop                  : Eta -> Continuation -> Continuation.

Inductive State : Set :=
 | StateCons : Mu -> H -> Eta -> Expression -> Continuation -> State.
```

Appendix C

Javalite Coq Helper Functions

```
Definition Hv_To_V (loc:Location) (hv:Hv) : V :=
 match hv with
 | Hv_v v             => v
 | Hv_object heapobj => V_Error
 end.

Definition get_variable_name (vardec:VariableDeclaration) :=
 match vardec with
 | VarDec someid _ => someid
 end.

(* ------------------------------------------------------------------------ *)
(* ------------------------- Additional Definitions ------------------------- *)
(* ------------------------------------------------------------------------ *)
Definition CList        := list C.
Definition CLList       := list CL.
Definition IdList       := list id.
Definition XList        := list X.
Definition LocationList := list Location.

Definition id_eq_dec := positive_eq_dec.
Definition F_eq_dec := id_eq_dec.
Definition M_eq_dec := id_eq_dec.

(* ------------------------------------------------------------------------ *)
(* -------------------------- Decidability Proofs -------------------------- *)
(* ------------------------------------------------------------------------ *)
Theorem X_eq_dec:
 forall (x y:X), {x = y} + {x <> y}.
Proof.
 decide equality.
 apply id_eq_dec.
```

```
Qed.

Theorem C_eq_dec:
 forall (x y:C), {x = y} + {x <> y}.
Proof.
 decide equality.
 apply id_eq_dec.
Qed.

Theorem Pointer_eq_dec:
 forall (x y:Pointer), {x = y} + {x <> y}.
Proof.
 decide equality.
 apply C_eq_dec.
 apply eq_nat_dec.
Qed.

(*
 We don't need a "Boolean_eq_dec" theorem since we have aliased Boolean
 to bool. Therefore, "bool_dec" will suffice.
*)

Theorem V_eq_dec:
 forall (x y:V), {x = y} + {x <> y}.
Proof.
 decide equality.
 apply Pointer_eq_dec.
 apply bool_dec.
Qed.

(*
Theorem SimpleType_eq_dec:
 forall (x y:SimpleType), {x=y} + {x <> y}.
Proof.
 decide equality.
 apply bool_dec.
Qed.
*)

Theorem T_eq_dec:
 forall (x y:T), {x=y} + {x <> y}.
```

```
Proof.
 decide equality.
 apply C_eq_dec.
Qed.

Theorem VariableDeclaration_eq_dec:
 forall (x y:VariableDeclaration), {x = y} + {x <> y}.
Proof.
 decide equality.
 apply T_eq_dec.
 apply id_eq_dec.
Qed.

Theorem ValueList_eq_dec:
 forall (x y:list V), {x=y} + {x <> y}.
Proof.
 decide equality.
 apply V_eq_dec.
Qed.

Theorem Expression_eq_dec:
 forall (x y: Expression), {x=y} + {x <> y}.
Proof.
 decide equality.
 apply X_eq_dec.
 apply V_eq_dec.
 apply C_eq_dec.
 apply F_eq_dec.
 apply list_eq_dec.
 apply X_eq_dec.
 apply M_eq_dec.
 apply list_eq_dec.
 apply V_eq_dec.
 apply M_eq_dec.
 apply Pointer_eq_dec.
 apply C_eq_dec.
 apply C_eq_dec.
 apply X_eq_dec.
 apply F_eq_dec.
 apply X_eq_dec.
 apply VariableDeclaration_eq_dec.
```

```
Qed.

Theorem Method_eq_dec:
 forall (x y:Method), {x = y} + {x <> y}.
Proof.
 decide equality.
 apply Expression_eq_dec.
 apply list_eq_dec.
 apply VariableDeclaration_eq_dec.
 apply T_eq_dec.
 apply M_eq_dec.
Qed.

Theorem Eta_eq_dec:
 forall (x y:Eta), {x=y} + {x <> y}.
Proof.
 decide equality.
 apply eq_nat_dec.
 apply X_eq_dec.
Qed.

Theorem Continuation_eq_dec:
 forall (x y:Continuation), {x=y} + {x <> y}.
Proof.
 decide equality.
 apply F_eq_dec.
 apply list_eq_dec.
 apply X_eq_dec.
 apply M_eq_dec.
 apply Expression_eq_dec.
 apply V_eq_dec.
 apply C_eq_dec.
 apply C_eq_dec.
 apply X_eq_dec.
 apply F_eq_dec.
 apply X_eq_dec.
 apply Expression_eq_dec.
 apply Expression_eq_dec.
 apply Expression_eq_dec.
 apply X_eq_dec.
 apply T_eq_dec.
```

```
 apply Expression_eq_dec.
 apply Eta_eq_dec.
Qed.

(* ------------------------------------------------------------------------ *)
(* ---------------------------- Helper Functions ---------------------------- *)
(* ------------------------------------------------------------------------ *)

Definition location_to_key := P_of_succ_nat.
Definition h_lookup (h: H) (loc : Location) : option Hv :=
                    HashMap.find (location_to_key loc) h.
Definition h_lookup_optional_loc (h: H) (loc : option Location) : option V :=
 match loc with
 | None => None
 | Some loc => let res := HashMap.find (location_to_key loc) h in
               match res with
               | None => None
               | Some hv => Some (Hv_To_V loc hv)
               end
 end.

Fixpoint eta_lookup (eta: Eta) (x : X) : option Location :=
 match eta with
 | Eta_mt                => None
 | Eta_NotMt eta' x' loc => if X_eq_dec x x' then Some loc else eta_lookup eta' x
 end.

Definition lookup_argument_locations (eta:Eta) (args:list X) : list (option Location) :=
 map (eta_lookup eta) args.

Definition lookup_arguments_option (h:H) (eta:Eta) (args:list X) : list (option V) :=
 map (h_lookup_optional_loc h) (lookup_argument_locations eta args).

Definition remove_optional_v optionv :=
 match optionv with
 | Some v => v
 | _ => V_Error
 end.

Definition lookup_arguments (h:H) (eta:Eta) (args:list X) : list V :=
 map remove_optional_v (lookup_arguments_option h eta args).
```

```
(* Look up the class "c" in the registry "mu" *)

Definition convert_C_to_CL (c:C) (mu:Mu) : option CL :=
 match c with
 | SomeClass id => HashMap.find id mu
 end.

Definition convert_CL_to_C (cl:CL) : C :=
 match cl with
 | ClassDecl classname _ _ _ => classname
 end.

Definition class_lookup                    := convert_C_to_CL.
Definition convert_CList_to_CLList (mu:Mu):= map (fun c => convert_C_to_CL c mu).
Definition convert_CLList_to_CList         := map convert_CL_to_C.
Definition convert_CLList_to_CList_option := option_map convert_CL_to_C.

Definition get_parent_of_CL_as_CL (cl:CL) : option CL :=
 match cl with
 | ClassDecl _ superclass _ _ => superclass
 end.

Definition get_parent_of_C_as_CL (c:C) (mu:Mu) : option CL :=
 match (convert_C_to_CL c mu) with
 | None  => None
 | Some cl => get_parent_of_CL_as_CL cl
 end.

Definition get_parent_of_cl_as_C (cl:CL) : option C :=
 match (get_parent_of_CL_as_CL cl) with
 | None     => None
 | Some cl' => Some (convert_CL_to_C cl')
 end.

Definition get_parent_of_c_as_C (c:C) (mu:Mu) : option C :=
 match (convert_C_to_CL c mu) with
 | None    => None
 | Some cl => get_parent_of_cl_as_C cl
 end.
```

```
Definition get_id_from_C (c:C) : id :=
 match c with
 | SomeClass someid => someid
 end.

Definition get_C_from_id (someid:id) : C := SomeClass someid.
Definition get_CL_from_id (someid:id) (mu:Mu) : option CL :=
           convert_C_to_CL (get_C_from_id someid) mu.

Fixpoint get_class_hierarchy_gas_CL_to_CLList (n:nat)
         (cl: option CL) (mu:Mu) : option CLList :=
 match n with
 | O    => None
 | S n' => match cl with
           | None    => None
           | Some cl => match cl with
                        | ClassDecl class superclass _ _ =>
                                    match superclass with
                                    | None    => (* Important! Do not return nil *)
                                                 Some (cl::nil)
                                    | Some cl => option_map (fun x => (x ++ (cl :: nil)))
                                                 (get_class_hierarchy_gas_CL_to_CLList
                                                  n' (get_parent_of_CL_as_CL cl) mu)
                                    end
                        end
           end
 end.

Definition get_class_hierarchy_CL_to_CLList (n:nat) (cl:CL) (mu:Mu) : CLList :=
 match get_class_hierarchy_gas_CL_to_CLList n (Some cl) mu with
 | None        => cl :: nil
 | Some cllist => cllist
 end.

Definition get_reversed_class_hierarchy_CL_to_CLList (n:nat) (cl:CL) (mu:Mu) : CLList :=
 match get_class_hierarchy_gas_CL_to_CLList n (Some cl) mu with
 | None        => cl :: nil
 | Some cllist => rev cllist
 end.

Definition class_list_from_class_to_field_locations_map
```

```
                        (c2flm:ClassToFieldLocationsMap) : CList :=
 map (fun p => (get_C_from_id (fst p))) (HashMap.elements c2flm).

Definition classes_of_parents_and_self (c:C) (mu:Mu) : option CList :=
 match (convert_C_to_CL c mu) with
 | None    => Some (c :: nil)
 | Some cl => Some (convert_CLList_to_CList
                    (get_class_hierarchy_CL_to_CLList (HashMap.cardinal mu) cl mu))
 end.

Definition class_decls_of_parents_and_self (c:C) (mu:Mu) : option CLList :=
 match (convert_C_to_CL c mu) with
 | None    => None
 | Some cl => Some (get_class_hierarchy_CL_to_CLList (HashMap.cardinal mu) cl mu)
 end.

Fixpoint hierarchical_field_lookup_from_list (f:F) (cllist:CLList)
                                (c2flm:ClassToFieldLocationsMap) : option Location :=
 match cllist with
 | nil     => None
 | cl :: t => match HashMap.find (get_id_from_C (convert_CL_to_C cl)) c2flm with
              | None     => (hierarchical_field_lookup_from_list f t c2flm)
              | Some fls => match HashMap.find f fls with
                            | Some loc => Some loc
                            | None     => (hierarchical_field_lookup_from_list f t c2flm)
                            end
              end
 end.

Definition hierarchical_field_lookup (f:F) (c:C) (c2flm:ClassToFieldLocationsMap)
       (mu:Mu) : option Location :=
 match (convert_C_to_CL c mu) with
 | None    => None
 | Some cl => match get_reversed_class_hierarchy_CL_to_CLList
               (HashMap.cardinal c2flm) cl mu with
              | nil   => None
              | cl::t => (hierarchical_field_lookup_from_list f (cl::t) c2flm)
              end
 end.

Definition field_lookup (object:HeapObject) (c:C) (f:F) (mu:Mu) : option Location :=
```

```
 match object with
 | HeapObj c2flm => (hierarchical_field_lookup f c c2flm mu)
 end.

Definition contains_entry_for_class (c:C) (c2flm:ClassToFieldLocationsMap) : bool :=
 HashMap.mem (get_id_from_C c) c2flm.

Definition can_cast (object:HeapObject) (c:C) : bool :=
 match object with
 | HeapObj cfll => contains_entry_for_class c cfll
 end.

(* Only needs to ensure that the cast type is in the hierarchy *)
Definition cast (object:HeapObject) (c:C) : option HeapObject :=
 match object with
 | HeapObj cfll => if (can_cast object c) then Some object else None
 end.

Fixpoint h_max (h:H) : Location :=
 fold_left MinMax.max (map (fun el => nat_of_P (fst(el))) (HashMap.elements h)) O.

Definition h_malloc (h:H) : Location := S (h_max h).

(* Generate a list of numbers from [base .. start+base] (happens to be in reverse) *)

Fixpoint generate_n_numbers (start base:nat) : LocationList :=
 match start with
 | O   => nil
 | S n => (start + base) :: (generate_n_numbers n base)
 end.

Definition h_malloc_n (h:H) (number:nat) : LocationList :=
 match number with
 | O => nil
 | _ => generate_n_numbers number (h_max h)
 end.

Definition convert_to_boolean_expr (b:Boolean) : Expression := Expr_V (V_Bool b).
Definition convert_pointer_to_expr (p:Pointer) : Expression := Expr_V (V_Pointer p).
Definition Boolean_equals := eqb.
```

```
Definition Pointer_equals (p0 p1: Pointer) : Boolean :=
 match Pointer_eq_dec p0 p1 with
 | left _ => true
 | _ => false
 end.

Definition V_equals (v0 v1:V) : Boolean :=
 match V_eq_dec v0 v1 with
 | left _ => true
 | _ => false
 end.

Definition h_extend (h:H) (loc:Location) (hv:Hv) : H :=
 HashMap.add (location_to_key loc) hv h.

Fixpoint h_extend_star (h:H) (loclist:LocationList) (varlist:list V) : option H :=
  match loclist, varlist with
  | nil,    nil     => Some h
  | loc::t, var::t' => h_extend_star (h_extend h loc (Hv_v var)) t t'
  | _, _            => None
  end.

Theorem eq_lengths_means_some_h:
 forall (loclist:LocationList) (varlist:list V) h h',
  (length loclist) = (length varlist) -> (h_extend_star h loclist varlist = Some h').
Proof.
 intros.
 Admitted.

Fixpoint h_extend_star_hierarchical_list (h:H) (hl:list LocationList)
                                         (dvs:list (list V)) : option H :=
 match hl, dvs with
 | nil,   nil  => Some h
 | l::t, v::t' => match (h_extend_star h l v) with
                  | None    => None
                  | Some h' => h_extend_star_hierarchical_list h' t t'
                  end
 | _,       _  => None
 end.

Fixpoint assign_field_locations (loclist:LocationList)
```

```
              (fieldvaluelist:list (F * V)) : HashMap.t Location :=
 match loclist, fieldvaluelist with
 | loc::t, (field, val)::t' => HashMap.add field loc (assign_field_locations t t')
 | _, _ => HashMap.empty Location
 end.

Fixpoint create_hierarchical_field_location_map (hl:list LocationList)
         (dvs:list FieldValueMap) : list FieldLocationMap :=
 match hl, dvs with
 | loclist::t, fvm::t' => (assign_field_locations loclist (HashMap.elements fvm)) ::
                                  (create_hierarchical_field_location_map t t')
 | _, _ => nil
 end.

Fixpoint create_location_value_list_helper (fieldvalues:list (F*V))
      (flm:FieldLocationMap) : list (Location*V) :=
 match fieldvalues with
 | h::t => match (HashMap.find (fst h) flm) with
           | Some loc => (loc, (snd h)) :: (create_location_value_list_helper t flm)
           | None => nil
           end
 | _  => nil
 end.

Definition create_location_value_list (flm:FieldLocationMap) (fvm:FieldValueMap)
 : list (Location*V) := create_location_value_list_helper (HashMap.elements fvm) flm.

Definition create_hierarchical_location_value_list (hl:list FieldLocationMap)
 (dvs:list FieldValueMap) :=
   map (fun hl_dvs => create_location_value_list (fst hl_dvs) (snd hl_dvs))
       (combine hl dvs).

Fixpoint h_extend_star_hierarchical_map (h:H) (hl:list FieldLocationMap)
 (dvs:list FieldValueMap) : option H :=
 match hl, dvs with
 | flm::t, fvm::t' => match create_location_value_list flm fvm with
                      | nil => h_extend_star_hierarchical_map h t t'
                      | (loc, v)::tl => let (loclist, vlist) := (split tl) in
                                        match (h_extend_star (h_extend h loc (Hv_v v))
                                                             loclist vlist) with
                                        | None => None
```

```
                                        | Some h' =>
                                          (h_extend_star_hierarchical_map h' t t')
                                        end
                        end
 | _, _ => Some h
 end.

Definition eta_extend (eta:Eta) (x:X) (loc:Location) := (Eta_NotMt eta x loc).

(*
Ignores invalid arguments such as mismatched lengths for loclist & varlist
Check for extending with something that already exists - a hashmap would easily
handle these scenarios
*)
Fixpoint eta_extend_star (eta:Eta) (xlist:IdList) (loclist:LocationList) : option Eta :=
 match loclist, xlist with
 | nil, nil         => Some eta
 | loc::t, var::t' => eta_extend_star (Eta_NotMt eta (SomeId var) loc) t' t
 | _, _             => None
 end.

Definition eval_if_then_else (v1:Boolean) (e_t e_f:Expression) :=
 match v1 with
 | true  => e_t
 | false => e_f
 end.

Definition get_method_list (cl:CL) : MethodMap :=
 match cl with
 | ClassDecl _ _ _ methodlist => methodlist
 end.

Definition get_field_list (cl:CL) : FieldTypeMap :=
 match cl with
 | ClassDecl _ _ fieldlist _ => fieldlist
 end.

Definition method_lookup (m:M) (cl:CL) : option Method :=
 HashMap.find m (get_method_list cl).

Fixpoint get_class_with_virtual_method (m:M) (classlist:list CL) : option CL :=
```

```
 match classlist with
 | nil   => None
 | cl::t => match method_lookup m cl with
            | Some method => Some cl
            | None        => get_class_with_virtual_method m t
            end
 end.

Fixpoint get_virtual_method (m:M) (classlist:list CL) : option Method :=
 match (get_class_with_virtual_method m classlist) with
 | None    => None
 | Some cl => method_lookup m cl
 end.

Definition get_fields_of_parents_and_self_from_list (cllist:list CL) : list FieldTypeMap :=
 map (fun cl => (get_field_list cl)) cllist.

Definition get_fields_of_parents_and_self_from_CL (cl:CL) (mu:Mu) : list FieldTypeMap :=
 get_fields_of_parents_and_self_from_list
    (get_class_hierarchy_CL_to_CLList (HashMap.cardinal mu) cl mu).

Definition get_fields_of_parents_and_self_C (c:C) (mu:Mu) : list FieldTypeMap :=
 match (convert_C_to_CL c mu) with
 | None    => nil
 | Some cl => get_fields_of_parents_and_self_from_CL cl mu
 end.

Definition get_type_list_unordered (fl:FieldTypeMap) : list T :=
 map (fun fl => snd fl) (HashMap.elements fl).

Definition get_hierarchical_type_map (hfm:list FieldTypeMap) : list FieldTypeMap := hfm.

Definition get_default_value (t:T) : V :=
 match t with
 | T_Class _ => V_Pointer Null
 | T_Unit    => V_Unit
 | T_Bool    => V_Bool false
 end.

Fixpoint create_default_values_hash_map elems : HashMap.t V :=
 match elems with
```

```
 | nil  => HashMap.empty V
 | h::t => match h with
           | (field, type) => HashMap.add field (get_default_value type)
                                          (create_default_values_hash_map t)
           end
 end.

Definition get_default_values (ftm:FieldTypeMap) : FieldValueMap :=
 create_default_values_hash_map (HashMap.elements ftm).

Definition get_hierarchical_default_values (htl:list FieldTypeMap) : list FieldValueMap :=
 map (fun tm => (get_default_values tm)) htl.

Definition make_heap_pointer (loc:Location) (c:C) := (Hv_v (V_Pointer (Addr loc c))).

Definition argument_list_to_XList (arglist:ArgumentList) : IdList :=
 map (fun vardec => get_variable_name vardec) arglist.

Definition get_field_ids (ftm:FieldTypeMap) : list id :=
 map (fun el => fst el) (HashMap.elements ftm).

Definition create_field_location_pairs (ftm:FieldTypeMap)
                                       (loclist:LocationList) : list (id * Location) :=
 (combine (get_field_ids ftm) loclist).

Fixpoint create_field_location_map (pairs: list(id * Location)) : FieldLocationMap :=
 match pairs with
 | nil => HashMap.empty Location
 | (x, loc)::t => HashMap.add x loc (create_field_location_map t)
 end.

Fixpoint build_class_loc_lists_helper (arg:list (C * (FieldTypeMap * LocationList)))
 : ClassToFieldLocationsMap :=
 match arg with
 | nil  => HashMap.empty FieldLocationMap
 | ((SomeClass c), (ftm, loclist))::t => HashMap.add c
                   (create_field_location_map (create_field_location_pairs ftm loclist))
                   (build_class_loc_lists_helper t)
 end.

(* "New" helpers *)
```

```
Definition build_class_loc_lists (hcl:CList)
                                 (hfl:list FieldTypeMap)
                                 (hl:list LocationList) : ClassToFieldLocationsMap :=
 if (beq_nat (length hfl) (length hl)) then
     let fields_and_locations := (combine hfl hl) in
       (if (beq_nat (length hcl) (length fields_and_locations)) then
           build_class_loc_lists_helper (combine hcl fields_and_locations)
        else HashMap.empty FieldLocationMap)
 else
     HashMap.empty FieldLocationMap.

Definition get_required_heap_space (l:list nat) : nat := fold_left plus l 0.

Fixpoint partition_list (the_list:list Location)
        (element_lengths: list nat) : list LocationList :=
 if beq_nat (length the_list) (get_required_heap_space element_lengths) then
  match element_lengths, the_list with
  | len::t, h::t' => (firstn len the_list) :: partition_list (skipn len the_list) t
  | _, _ => nil
  end
 else nil.

Definition h_malloc_n_star (h:H) (l:list nat) : list LocationList :=
 match l with
 | nil  => nil
 | _    => partition_list (h_malloc_n h (get_required_heap_space l)) l
 end.

Definition get_value_lengths (hfvm:list FieldValueMap) : list nat :=
 map (fun fvm => HashMap.cardinal fvm) hfvm.

Definition inject (prog : P) : State :=
 match prog with
 | Program mu entrypoint =>
    match entrypoint with
    | EntryPoint c m => (StateCons mu (HashMap.empty Hv) Eta_mt
                                   (MethodInvocation (NewClass c) m nil) K_Return)
    end
 end.

Definition inject_with_state (s:State) (m:M) : State :=
```

```
match s with
| StateCons mu h eta e k => (StateCons mu h eta (MethodInvocation e m nil) K_Return)
end.
```

Appendix D

Javalite Coq Reduction Relation

```
(* ------------------------------------------------------------------------ *)
(* ------------------------- Expression Reductions ------------------------- *)
(* ------------------------------------------------------------------------ *)

Inductive ExprReduces : State -> State -> Prop :=

 (* ------------------------------------------------------------------------ *)
 (* Variable Access *)

 | ER_VariableAccess :
  forall (x:X) (mu:Mu) (h:H) (eta:Eta) (k:Continuation) (l:Location) (hv:Hv),
   (eta_lookup eta x) = Some l ->
   (h_lookup h l)     = Some hv ->
   ExprReduces (StateCons mu h eta (Expr_X x) k)
               (StateCons mu h eta (Expr_V (Hv_To_V l hv)) k)

 (* ------------------------------------------------------------------------ *)
 (* Field Access - Object Eval *)

 | ER_FieldAccess1 :
  forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (f:F) (k:Continuation),
   ExprReduces (StateCons mu h eta (FieldRef e f) k)
               (StateCons mu h eta e (K_FieldAccess f k))

 (* Field access *)

 | ER_FieldAccess2 :
  forall mu h eta loc C f1 k v1 object obj loc_0,
   (h_lookup h loc)            = Some (Hv_object obj) ->
   (cast obj C)                = Some object          ->
   (field_lookup object C f1 mu) = Some loc_0           ->
   (h_lookup h loc_0)          = Some (Hv_v v1)       ->
```

```
  ExprReduces (StateCons mu h eta (Expr_V (V_Pointer (Addr loc C))) (K_FieldAccess f1 k))
              (StateCons mu h eta (Expr_V v1) k)

(* ------------------------------------------------------------------------ *)
(* Equals '==': l-operand eval *)

| ER_Equals1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e_0 e:Expression) (k:Continuation),
  ExprReduces (StateCons mu h eta (Equality e_0 e) k)
              (StateCons mu h eta e_0 (K_EqualityLeftOperand e k))

(* Equals '==': r-operand eval *)

| ER_Equals2 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (v:V) (k:Continuation),
  ExprReduces (StateCons mu h eta (Expr_V v) (K_EqualityLeftOperand e k))
              (StateCons mu h eta e (K_EqualityRightOperand v k))

(* Equals '==': equals *)

| ER_Equals3 :
 forall (mu:Mu) (h:H) (eta:Eta) (v_0 v_1:V) (k:Continuation),
  ExprReduces (StateCons mu h eta (Expr_V v_0) (K_EqualityRightOperand v_1 k))
              (StateCons mu h eta (convert_to_boolean_expr (V_equals v_0 v_1)) k)

(* ------------------------------------------------------------------------ *)
(* Typecast - Object eval *)

| ER_Typecast1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (c:C) (k:Continuation),
  ExprReduces (StateCons mu h eta (Cast c e) k)
              (StateCons mu h eta e (K_Cast c k))

(* Typecast *)

| ER_Typecast2 :
 forall (mu:Mu) (h:H) (eta:Eta) (c_0 c_1:C) (loc:Location)
        (object:HeapObject) (k:Continuation),
  (h_lookup h loc)      = Some (Hv_object object) ->
  (can_cast object c_1) = true ->
  ExprReduces (StateCons mu h eta (convert_pointer_to_expr (Addr loc c_0)) (K_Cast c_1 k))
```

```
              (StateCons mu h eta (convert_pointer_to_expr (Addr loc c_1)) k)

(* ------------------------------------------------------------------------ *)
(* Assign - Object eval *)

| ER_Assign1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (x:X) (k:Continuation),
  ExprReduces (StateCons mu h eta (VarAssign x e) k)
              (StateCons mu h eta e (K_VarAssign x k))

(* Assign *)

| ER_Assign2 :
 forall (mu:Mu) (h h_0:H) (eta:Eta) (v:V) (loc:Location) (x:X) (k:Continuation),
  (eta_lookup eta x)        = Some loc ->
  (h_extend h loc (Hv_v v)) = h_0 ->
  ExprReduces (StateCons mu h eta (Expr_V v) (K_VarAssign x k))
              (StateCons mu h_0 eta (Expr_V v) k)

(* ------------------------------------------------------------------------ *)
(* Assign Field - Object eval *)

| ER_AssignField1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (x:X) (f:F) (k:Continuation),
  ExprReduces (StateCons mu h eta (FieldAssign x f e) k)
              (StateCons mu h eta e (K_FieldAssign x f k))

(* Assign Field *)

| ER_AssignField2 :
 forall (mu:Mu) (h h_0:H) (eta:Eta) (x:X) (f:F) (c:C) (v:V)
        (loc_0 loc_1 loc_2:Location) (obj object:HeapObject) (k:Continuation),
  (eta_lookup eta x)               = Some loc_0  ->
  (h_lookup h loc_0)               = Some (Hv_v (V_Pointer (Addr loc_1 c))) ->
  (h_lookup h loc_1)               = Some (Hv_object obj) ->
  (cast obj c)                     = Some object ->
  (field_lookup object c f mu)     = Some loc_2  ->
  (h_extend h loc_2 (Hv_v v))      = h_0 ->
  ExprReduces (StateCons mu h eta (Expr_V v) (K_FieldAssign x f k))
              (StateCons mu h_0 eta (Expr_V v) k)
```

```
(* ------------------------------------------------------------------------ *)
(* Pop eta (close scope) *)

| ER_PopEta :
 forall (mu:Mu) (h:H) (eta eta_0:Eta) (v:V) (k:Continuation),
  ExprReduces (StateCons mu h eta (Expr_V v) (K_Pop eta_0 k))
              (StateCons mu h eta_0 (Expr_V v) k)

(* ------------------------------------------------------------------------ *)
(* Begin - Empty expression list *)

| ER_BeginEmptyExpList :
 forall (mu:Mu) (h:H) (eta:Eta) (k:Continuation),
  ExprReduces (StateCons mu h eta VoidExp k)
              (StateCons mu h eta (Expr_V V_Unit) k)

(* Begin - e_0 evaluation *)

| ER_Begin_e_0_evaluation :
 forall (mu:Mu) (h:H) (eta:Eta) (k:Continuation) (e_0 e_1:Expression),
  ExprReduces (StateCons mu h eta (SeqExp e_0 e_1) k)
              (StateCons mu h eta e_0 (K_Seq e_1 k))

(* Begin - e_1 evaluation *)

| ER_Begin_e_1_evaluation :
 forall (mu:Mu) (h:H) (eta:Eta) (v:V) (k:Continuation) (e_1:Expression),
  ExprReduces (StateCons mu h eta (Expr_V v) (K_Seq e_1 k))
              (StateCons mu h eta e_1 k)

(* ------------------------------------------------------------------------ *)
(* Instanceof - object eval *)

| ER_InstanceOf1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e:Expression) (c:C) (k:Continuation),
  ExprReduces (StateCons mu h eta (InstanceOf e c) k)
              (StateCons mu h eta e (K_InstanceOf c k))

(* Instanceof *)

| ER_InstanceOf2 :
```

```
 forall (mu:Mu) (h:H) (eta:Eta) (c_0 c_1:C) (v_res:Boolean) (k:Continuation)
        (loc:Location) (object:HeapObject),
  (h_lookup h loc)      = Some (Hv_object object) ->
  (can_cast object c_1) = v_res ->
  ExprReduces (StateCons mu h eta (convert_pointer_to_expr (Addr loc c_0))
                          (K_InstanceOf c_1 k))
              (StateCons mu h eta (convert_to_boolean_expr v_res) k)

(* ------------------------------------------------------------------------ *)
(* Variable declaration - object eval *)

| ER_VarDec1 :
 forall (mu:Mu) (h:H) (eta:Eta) (e_0 e_1:Expression) (x1:id) (t:T) (k:Continuation),
  ExprReduces (StateCons mu h eta (VarDecExp (VarDec x1 t) e_0 e_1) k)
              (StateCons mu h eta e_0 (K_VarAssignIn t (SomeId x1) e_1 k))

(* Variable declaration *)

| ER_VarDec2 :
 forall (mu:Mu) (h h_0:H) (eta eta_0:Eta) (v:V) (e_1:Expression)
        (x1:id) (t:T) (k:Continuation) (loc_x:Location),
  (h_malloc h)                          = loc_x ->
  (h_extend h loc_x (Hv_v v))           = h_0   ->
  (eta_extend eta (SomeId x1) loc_x)    = eta_0 ->
  ExprReduces (StateCons mu h eta (Expr_V v) (K_VarAssignIn t (SomeId x1) e_1 k))
              (StateCons mu h_0 eta_0 e_1 (K_Pop eta k))

(* ------------------------------------------------------------------------ *)
(* If-then-else - object eval *)

| ER_IfThenElseObjectEval :
 forall (mu:Mu) (h:H) (eta:Eta) (e_p e_t e_f:Expression) (k:Continuation),
  ExprReduces (StateCons mu h eta (IfExpr e_p e_t e_f) k)
              (StateCons mu h eta e_p (K_If e_t e_f k))

(* If-then-else *)

| ER_IfThenElse :
 forall (mu:Mu) (h:H) (eta:Eta) (v1:Boolean) (e_t e_f:Expression) (k:Continuation),
  ExprReduces (StateCons mu h eta (convert_to_boolean_expr v1) (K_If e_t e_f k))
              (StateCons mu h eta (eval_if_then_else v1 e_t e_f) k)
```

```
(* ------------------------------------------------------------------------ *)
(* Method invocation - object eval *)

| ER_MethodInvocationObjectEval :
 forall (mu:Mu) (h:H) (eta:Eta) (e_0:Expression) (args:list X) (m:M) (k:Continuation),
  ExprReduces (StateCons mu h eta (MethodInvocation e_0 m args) k)
              (StateCons mu h eta e_0 (K_MethodInvocation m args k))

(* Method invocation *)

| ER_MethodInvocation :
 forall (mu:Mu) (h:H) (eta:Eta) (pv_o:Pointer) (m:M)
        (k:Continuation) (args:list X) (primitive_args:list V),
  (lookup_arguments h eta args) = primitive_args ->
  ExprReduces (StateCons mu h eta (Expr_V (V_Pointer pv_o)) (K_MethodInvocation m args k))
              (StateCons mu h eta (Raw pv_o m primitive_args) k)

(* Raw method invocation - there are a couple of Redex tests that can easily
   break this like setting only Object as the rv from class-list-from-object *)

| ER_MethodInvocationRaw :
 forall (mu:Mu) (h h_0 h_tmp:H) (eta eta_0:Eta) (e_m:Expression) (k:Continuation)
        (m:M) (varlist:list V) (c C_t:C) (loc loc_o:Location) (loclist:LocationList)
        (methodvars:IdList) (obj1:HeapObject) (arglist:ArgumentList) (t:T)
        (classlist:CLList) (CL_t:CL),
  (h_lookup h loc)                 = Some (Hv_object obj1) ->
  (class_decls_of_parents_and_self c mu)      = Some classlist ->

  (get_class_with_virtual_method m classlist) = Some CL_t ->
  (convert_CL_to_C CL_t)                      = C_t       ->
  (method_lookup m CL_t)                      = Some (AMethod m t arglist e_m) ->
  (argument_list_to_XList arglist)            = methodvars ->

  (* allocate locations for "this" and the arguments *)
  (h_malloc_n h (S (length varlist)))         = (loc_o::loclist) ->

  (* write "this" and the args into the heap.
     loclist and varlist must be the same length. *)
  (h_extend h loc_o (make_heap_pointer loc C_t)) = h_tmp   ->
  (h_extend_star h_tmp loclist varlist)          = Some h_0 ->
```

```
  (* create a new local environment with the bindings for "this" and args *)
  (eta_extend_star (eta_extend eta This loc_o) methodvars loclist) = Some eta_0 ->

  ExprReduces (StateCons mu h eta (Raw (Addr loc c) m varlist) k)
              (StateCons mu h_0 eta_0 e_m (K_Pop eta k))

(* ------------------------------------------------------------------------ *)
(* New *)

| ER_New :
 forall (mu:Mu) (h h_0 h_1:H) (eta:Eta) (c:C) (loc_1:Location)
        (k                         : Continuation)
        (classlist                 : CList)
        (defaultvalues             : list FieldValueMap)
        (hierarchicalfieldlist     : list FieldTypeMap)
        (hierarchicaltypelist      : list FieldTypeMap)
        (hierarchicallocations     : list LocationList)
        (listofclassfieldloclists : ClassToFieldLocationsMap)
        (hierarchicalfieldlocmap   : list FieldLocationMap),

  (classes_of_parents_and_self c mu)                     = Some classlist        ->
  (beq_nat 0 (length classlist))                         = false                 ->
  (get_fields_of_parents_and_self_C c mu)                = hierarchicalfieldlist ->
  (get_hierarchical_type_map hierarchicalfieldlist)      = hierarchicaltypelist  ->
  (get_hierarchical_default_values hierarchicaltypelist) = defaultvalues         ->

  (h_malloc_n_star h (get_value_lengths defaultvalues))  = hierarchicallocations ->
  (create_hierarchical_field_location_map hierarchicallocations
                                          defaultvalues)  = hierarchicalfieldlocmap  ->
  (h_extend_star_hierarchical_map h hierarchicalfieldlocmap defaultvalues)
                                  = Some h_0 ->

  (h_malloc h_0)                                                                = loc_1  ->
  (build_class_loc_lists classlist hierarchicalfieldlist hierarchicallocations)
                         = listofclassfieldloclists ->
  (h_extend h_0 loc_1 (Hv_object (HeapObj listofclassfieldloclists)))           = h_1 ->

  ExprReduces (StateCons mu h eta (NewClass c) k)
              (StateCons mu h_1 eta (convert_pointer_to_expr (Addr loc_1 c)) k).
```

Appendix E

Javalite Coq Proofs

```
Hint Constructors ExprReduces.

Require Import Coq.Sets.Relations_1.
Definition ExprReducesTransitive := Transitive ExprReduces.

Theorem ExprReduces_fun:
 forall (s s' s'':State),
  ExprReduces s s' ->
  ExprReduces s s'' ->
  s' = s''.
Proof.
 intros.
 destruct s.

 inversion H0; inversion H1; try congruence || (subst; discriminate).

 subst.
 inversion H14.
 inversion H15.
 reflexivity.

 subst.
 inversion H15.
 inversion H14.
 congruence.

 subst.
 inversion H12.
 inversion H13.
 reflexivity.
Qed.
```

```
Axiom MethodLookupSoundness:
 forall (cl:CL) (e:Expression) (arglist:ArgumentList) (t:T) (m1 m2:M),
  (method_lookup m1 cl) = Some (AMethod m2 t arglist e) -> m1 = m2.

Theorem ExprReduces_dec:
 forall (s:State),
  { s' | ExprReduces s s' } + { forall s', ~ ExprReduces s s' }.
Proof.
 intros. destruct s. generalize m h e c. clear m h e c.
 rename e0 into x.

 Ltac dispatch_invalid_state := right; intuition; inversion_clear H0.
 Ltac dispatch_invalid_state_with_congruence := right;
               intuition; inversion_clear H0; congruence.

 induction x.

 (* Expr_X *)
 intros.
 remember (eta_lookup e x) as eta_lookup_e_x.
 destruct eta_lookup_e_x.
 remember (h_lookup h l) as h_lookup_h_l.
 destruct h_lookup_h_l.

 left.
 exists (StateCons m h e (Expr_V (Hv_To_V l h0)) c).
 apply (ER_VariableAccess x m h e c l h0); symmetry; assumption.

 dispatch_invalid_state_with_congruence.
 dispatch_invalid_state_with_congruence.

 (* Expr_V *)
 intros.
 destruct c.

 (** K_Return is not a valid continuation *)
 dispatch_invalid_state.

 (** K_FieldAccess *)
 destruct v.
 destruct p as [loc|].
```

```
remember (h_lookup h loc) as h_lookup_h_loc.
destruct h_lookup_h_loc.
destruct h0 as [hv|obj].

(** h_lookup h loc must return an object *)
dispatch_invalid_state_with_congruence.

remember (cast obj c0) as cast_obj_c0.
destruct cast_obj_c0 as [object|].
remember (field_lookup object c0 f m) as field_lookup_object_f_mu.
destruct field_lookup_object_f_mu as [loc_0|].
remember (h_lookup h loc_0) as h_lookup_h_loc_0.
destruct h_lookup_h_loc_0 as [hv|].
destruct hv.

left.
exists (StateCons m h e (Expr_V v) c).
apply (ER_FieldAccess2 m h e loc c0 f c v object obj loc_0); symmetry; assumption.

(** (h_lookup h loc_0) must be (Hv_v v) *)
dispatch_invalid_state_with_congruence.

(** (h_lookup h loc_0) must be (Hv_v v) *)
dispatch_invalid_state_with_congruence.

(** (field_lookup object f1 mu) must be (Some loc_0) *)
dispatch_invalid_state_with_congruence.

(** (cast obj C) must be Some object *)
dispatch_invalid_state_with_congruence.

(** (h_lookup h loc) must be Some ... *)
dispatch_invalid_state_with_congruence.

(** Null pointers can't be used for field access *)
dispatch_invalid_state_with_congruence.

(** Only pointers can be used for field access *)
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
```

```
(** K_MethodInvocation *)
destruct v.
left.
exists (StateCons m h e (Raw p m0 (lookup_arguments h e l)) c).
apply ER_MethodInvocation.
reflexivity.

(** Only pointers can be used for method invocation *)
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.

(** K_EqualityLeftOperand *)
left.
exists (StateCons m h e e0 (K_EqualityRightOperand v c)).
apply ER_Equals2.

(** K_EqualityRightOperand *)
left.
exists (StateCons m h e (convert_to_boolean_expr (V_equals v v0)) c).
apply ER_Equals3.

(** K_Cast *)
destruct v.
destruct p as [loc|].
remember (h_lookup h loc) as h_lookup_h_loc.
destruct h_lookup_h_loc as [hl|].
destruct hl as [hv|object].

(*** h_lookup h loc must be Some heap object *)
dispatch_invalid_state_with_congruence.

remember (can_cast object c) as can_cast.
destruct can_cast.
left.
exists (StateCons m h e (Expr_V (V_Pointer (Addr loc c))) c0).
apply ER_Typecast2 with object.
symmetry. apply Heqh_lookup_h_loc.
symmetry. apply Heqcan_cast.
```

```
(*** can_cast should be true *)
dispatch_invalid_state_with_congruence.

(*** h_lookup should be Some ... *)
dispatch_invalid_state_with_congruence.

(*** Null pointer cannot be used for casting - do we need to allow this? *)
dispatch_invalid_state_with_congruence.

(*** Only pointers can be used for casting *)
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.

(** K_InstanceOf *)
destruct v.
destruct p as [loc|].
remember (h_lookup h loc) as h_lookup_h_loc.
destruct h_lookup_h_loc as [hl|].
destruct hl as [hv|object].

(*** h_lookup h loc must be Some heap object *)
dispatch_invalid_state_with_congruence.

remember (can_cast object c) as can_cast_object.
left.
exists (StateCons m h e (convert_to_boolean_expr can_cast_object) c0).
apply ER_InstanceOf2 with object.
symmetry. apply Heqh_lookup_h_loc.
symmetry. apply Heqcan_cast_object.

(*** h_lookup h loc must be Some heap object *)
dispatch_invalid_state_with_congruence.

(*** Null pointer cannot be used for instanceof - do we need to allow this? *)
dispatch_invalid_state_with_congruence.

(*** Only pointers can be used for instanceof *)
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
```

```
(** K_VarAssign *)
remember (eta_lookup e x) as elex.
destruct elex as [loc|].
remember (h_extend h loc (Hv_v v)) as extendedH.

left.
exists (StateCons m extendedH e (Expr_V v) c).
apply ER_Assign2 with loc.
symmetry. apply Heqelex.
symmetry. apply HeqextendedH.

(*** eta_lookup must be Some ... *)
dispatch_invalid_state_with_congruence.

(** K_FieldAssign *)
remember (eta_lookup e x) as elex.
destruct elex as [loc_0|].
remember (h_lookup h loc_0) as hlhl0.
destruct hlhl0 as [heapthing|].
destruct heapthing as [hv|heapobj].
destruct hv.
destruct p as [loc_1|].
remember (h_lookup h loc_1) as hlhl1.
destruct hlhl1 as [heapthing|].
destruct heapthing as [hv|obj].

(*** h_lookup h loc must be Some heap object *)
dispatch_invalid_state_with_congruence.

remember (cast obj c0) as cast_obj_c.
destruct cast_obj_c as [object|].
remember (field_lookup object c0 f m) as flofm.
destruct flofm as [loc_2|].
remember (h_extend h loc_2 (Hv_v v)) as h_0.

left.
exists (StateCons m h_0 e (Expr_V v) c).
apply (ER_AssignField2 m h h_0 e x f c0 v loc_0 loc_1
                       loc_2 obj object); symmetry; assumption.
```

```
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.

(** K_If *)
destruct v.
dispatch_invalid_state_with_congruence.
left.
exists (StateCons m h e (eval_if_then_else b e0 e1) c).
apply ER_IfThenElse.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.

(** K_VarAssignIn *)
remember (h_malloc h) as loc_x.
remember (h_extend h loc_x (Hv_v v)) as h_0.
destruct x as [|x1].
dispatch_invalid_state_with_congruence.
remember (eta_extend e (SomeId x1) loc_x) as eta_0.

left.
exists (StateCons m h_0 eta_0 e0 (K_Pop e c)).
apply ER_VarDec2 with loc_x.
symmetry. apply Heqloc_x.
symmetry. apply Heqh_0.
symmetry. apply Heqeta_0.

(** K_Seq *)
left.
exists (StateCons m h e e0 c).
apply ER_Begin_e_1_evaluation.

(** K_Pop *)
left.
```

```
exists (StateCons m h e0 (Expr_V v) c).
apply ER_PopEta.

(* NewClass *)
intros.
remember (classes_of_parents_and_self c m) as option_classlist.
destruct option_classlist as [classlist|].
remember (get_fields_of_parents_and_self_C c m) as hierarchicalfieldlist.
remember (get_hierarchical_type_map hierarchicalfieldlist) as hierarchicaltypelist.
remember (get_hierarchical_default_values hierarchicaltypelist) as defaultvalues.
remember (h_malloc_n_star h (get_value_lengths defaultvalues)) as hierarchicallocations.
remember (create_hierarchical_field_location_map hierarchicallocations
           defaultvalues) as hierarchicalfieldlocmap.
remember (h_extend_star_hierarchical_map h hierarchicalfieldlocmap
           defaultvalues) as option_h_0.
destruct option_h_0 as [h_0|].
remember (h_malloc h_0) as loc_1.
remember (build_class_loc_lists classlist hierarchicalfieldlist
           hierarchicallocations) as listofclassfieldloclists.
remember (h_extend h_0 loc_1 (Hv_object (HeapObj listofclassfieldloclists))) as h_1.
remember (beq_nat 0 (length classlist)) as nonempty_classlist.
destruct nonempty_classlist.
dispatch_invalid_state_with_congruence.

left.
exists (StateCons m h_1 e (convert_pointer_to_expr (Addr loc_1 c)) c0).
apply (ER_New m h h_0 h_1 e c loc_1 c0 classlist
              defaultvalues hierarchicalfieldlist
              hierarchicaltypelist hierarchicallocations
              listofclassfieldloclists hierarchicalfieldlocmap); symmetry; assumption.

dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.

(* FieldRef *)
left.
exists (StateCons m h e x (K_FieldAccess f c)).
apply ER_FieldAccess1.

(* MethodInvocation *)
left.
```

```
exists (StateCons m0 h e x (K_MethodInvocation m l c)).
apply ER_MethodInvocationObjectEval.

(* Raw *)
intros.
destruct p as [loc|].
remember (h_lookup h loc) as hlhl.
destruct hlhl as [heapthing|].
destruct heapthing as [hv|obj1].

dispatch_invalid_state_with_congruence.

remember (class_decls_of_parents_and_self c0 m0) as optionclasslist.
destruct optionclasslist as [classlist|].
remember (get_class_with_virtual_method m classlist) as optionCL_t.

destruct optionCL_t as [CL_t|].
remember (convert_CL_to_C CL_t) as C_t.
remember (method_lookup m CL_t) as optionAMethod.

destruct optionAMethod as [someAMethod|].
destruct someAMethod.
remember (argument_list_to_XList a) as methodvars.
remember (h_malloc_n h (S (length l))) as loclist.

destruct loclist as [|loc_o].
dispatch_invalid_state_with_congruence.

remember (h_extend h loc_o (make_heap_pointer loc C_t)) as h_tmp.
remember (h_extend_star h_tmp loclist l) as optionH_0.

destruct optionH_0 as [h_0|].
remember (eta_extend_star (eta_extend e This loc_o) methodvars loclist) as optionEta_0.

destruct optionEta_0 as [eta_0|].

left.
exists (StateCons m0 h_0 eta_0 e0 (K_Pop e c)).
apply (ER_MethodInvocationRaw m0 h h_0 h_tmp e eta_0 e0 c m l c0 C_t
                              loc loc_o loclist methodvars obj1 a t
                              classlist CL_t); try congruence.
```

```
symmetry in HeqoptionAMethod.
assert (m = m1) as H_same_method.
apply MethodLookupSoundness in HeqoptionAMethod.
exact HeqoptionAMethod.
rewrite <- H_same_method in HeqoptionAMethod.
apply HeqoptionAMethod.

dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.
dispatch_invalid_state_with_congruence.

(* Equality *)
left.
exists (StateCons m h e x1 (K_EqualityLeftOperand x2 c)).
apply ER_Equals1.

(* Cast *)
left.
exists (StateCons m h e x (K_Cast c c0)).
apply ER_Typecast1.

(* InstanceOf *)
left.
exists (StateCons m h e x (K_InstanceOf c c0)).
apply ER_InstanceOf1.

(* VarAssign *)
left.
exists (StateCons m h e x0 (K_VarAssign x c)).
apply ER_Assign1.

(* FieldAssign *)
left.
exists (StateCons m h e x0 (K_FieldAssign x f c)).
apply ER_AssignField1.
```

```
 (* IfExpr *)
 left.
 exists (StateCons m h e x1 (K_If x2 x3 c)).
 apply ER_IfThenElseObjectEval.

 (* VarDecExp *)
 left.
 destruct v as [name t].
 exists (StateCons m h e x1 (K_VarAssignIn t (SomeId name) x2 c)).
 apply ER_VarDec1.

 (* VoidExp *)
 left.
 exists (StateCons m h e (Expr_V V_Unit) c).
 apply ER_BeginEmptyExpList.

 (* SeqExp *)
 left.
 exists (StateCons m h e x1 (K_Seq x2 c)).
 apply ER_Begin_e_0_evaluation.
Qed.

Extraction "javalite.ml" ExprReduces_dec.

Theorem ExprReduces_not_reflexive:
 forall (s:State),
  ~ ExprReduces s s.
Proof.
 intuition.
 destruct s.
 induction e0; inversion H0; try congruence.

 absurd (k = k); eauto. rewrite H8 at 1.
 clear - k. induction k; intros H; inversion H.
 eapply IHk. rewrite H1. exact H2.

 absurd (k = k); eauto. rewrite H8 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. exact H2.

 absurd (k = k); eauto. rewrite H8 at 1.
```

```
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. exact H2.

 absurd (k = k); eauto. rewrite H8 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. exact H2.

 absurd (k = k); eauto. rewrite H7 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. rewrite H2. exact H3.

 absurd (k = k); eauto. rewrite H8 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. exact H2.

 absurd (k = k); eauto. rewrite H8 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. rewrite <- H1. exact H2.

 absurd (k = k); eauto. rewrite H8 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. rewrite H2. exact H3.

 absurd (k = k); eauto. rewrite <- H7 in H11. rewrite <- H11 at 1.
 clear - k. induction k; eauto; intros H; inversion H.
 eapply IHk. rewrite H1. exact H2.
Qed.
```

References

Jien-Tsai Chan, Wuu Yang, and Jing-Wei Huang. Traps in Java. *Journal of Systems and Software*, 72(1): 33–47, June 2004. ISSN 0164-1212. doi: 10.1016/S0164-1212(03)00040-2. URL http://dx.doi.org/10.1016/S0164-1212(03)00040-2.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009. ISBN 978-0-262-06275-6.

Mattias Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, June 1987. ISSN 0304-3975. doi: 10.1016/0304-3975(87)90109-5. URL http://dx.doi.org/10.1016/0304-3975(87)90109-5.

Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338:153–183, June 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.12.006. URL http://dl.acm.org/citation.cfm?id=1085260.1085266.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66158-1. URL http://dl.acm.org/citation.cfm?id=645580.658808.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification (3rd Edition).* Addison-Wesley Professional, 2005. ISBN 0321246780.

Matthew Hennessy. *Lecture Notes - Chapter 1.* 2012. URL https://www.scss.tcd.ie/Matthew.Hennessy/teaching/2012/slexternal2012/notes/ArithNotes.pdf.

Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1435–1441, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2. doi: 10.1145/1141277.1141610. URL http://doi.acm.org/10.1145/1141277.1141610.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL http://doi.acm.org/10.1145/503502.503505.

Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 254–261, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=872023.872566.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–296, New York, NY, USA, 2012. ACM.

ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103691. URL http://doi.acm.org/10.1145/2103656.2103691.

Giovanni Lagorio and Marco Servetto. Strong exception-safety for Java-like languages. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, pages 3:1–3:7, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0540-2. doi: 10.1145/1924520.1924523. URL http://doi.acm.org/10.1145/1924520.1924523.

The Coq development team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2010. URL http://coq.inria.fr. Version 8.3.

Johan Östlund and Tobias Wrigstad. Welterweight Java. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, pages 97–116, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13952-3, 978-3-642-13952-9. URL http://dl.acm.org/citation.cfm?id=1894386.1894392.

Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2012.

Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL http://dl.acm.org/citation.cfm?id=645683.664578.

Neha Rungta and Eric Mercer. Slicing and dicing bugs in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 195–198, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1810295.1810328. URL http://doi.acm.org/10.1145/1810295.1810328.

David A. Schmidt. *Denotational Semantics: A Methodology For Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986. ISBN 0-697-06849-2.

Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. ISBN 0201656973.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926390. URL http://doi.acm.org/10.1145/1926385.1926390.

Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: core design and semantic definition. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 499–514, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297064. URL http://doi.acm.org/10.1145/1297027.1297064.

Alexander J. Summers. Modelling Java requires state. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, pages 10:1–10:3, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-540-6. doi: 10.1145/1557898.1557908. URL http://doi.acm.org/10.1145/1557898.1557908.