Dynamic Class Initialization Semantics: A Jinja Extension

Susannah Mansky
Department of Computer Science
University of Illinois at Urbana-Champaign
USA
sjohnsn2@illinois.edu

Elsa L. Gunter
Department of Computer Science
University of Illinois at Urbana-Champaign
USA
egunter@illinois.edu

Abstract

The Java Virtual Machine (JVM) postpones running class initialization methods until their classes are first referenced, such as by a new or static instruction. This process is called *dynamic class initialization*. Jinja is a semantic framework for Java and JVM developed in the theorem prover Isabelle that includes several semantics: Java-level big-step and small-step semantics, JVM-level small-step semantics, and an intermediate compilation step, J1, between these two levels. In this paper, we extend Jinja to include support for static instructions and dynamic class initialization. We also extend and re-prove related proofs, including Java-level type safety, equivalence between Java-level big-step and small-step semantics, and the correctness of a compilation from the Java level to the JVM level through J1. This work is based on the Java SE 8 specification.

CCS Concepts • **Theory of computation → Operational semantics**;

Keywords operational semantics, Java, Java Virtual Machine, dynamic class initialization, interactive theorem proving, compilation, type safety

ACM Reference Format:
Susannah Mansky and Elsa L. Gunter. 2019. Dynamic Class Initialization Semantics: A Jinja Extension. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3293880.3294104

CPP '19, January 14–15, 2019, Cascais, Portugal

ACM ISBN 978-1-4503-6222-1/19/01…$15.00
https://doi.org/10.1145/3293880.3294104

1 Introduction

In order to prove properties of programs, it is necessary to have a model for the behavior of the language they are written in. Jinja [Klein and Nipkow 2006] is an Isabelle development modeling a subset of Java, JVM byte code, and compilation from one to the other. This subset represents the core of the language and its behavior, but was not meant to be a complete representation. We sought to extend the subset covered.

Our motivation for this work is to achieve a semantics representing a large enough subset of the language to prove properties about the class use of programs. In particular, we wish to prove the safety of a regression testing algorithm described by Gligoric et al. [Gligoric et al. 2015], which collects the names of classes touched by a test during its run. A test is only rerun when a class in its touch-set has changed. Proving safety requires showing that any given test's behavior will be unchanged if its touch-set remains the same. Static instructions and dynamic class initialization are core uses of classes in typical programs, so any verification of this algorithm would need to address these features to be convincing. We therefore found it imperative to include them in the model we will use in our proof.

In this paper, we present an extension of Jinja's model to include static fields and methods and the instructions on them. We also give semantics for dynamic class initialization - the running of class initialization methods when classes are first referenced (interrupting the expression referencing the class), such as by static instructions - in order to more accurately represent the handling of statics. We then use these updated semantics to extend proofs at the Java level of progress, type safety, and equivalence of big-step and small-step semantics, and to extend the compiler from Java to JVM, including proofs of its correctness. While the language features we add here have been represented in some other models of Java (e.g., [Attali et al. 1998; Bogdanas and Roşu 2015]) and the JVM (e.g., [Atkey 2008; Belblidia and Debbabi 2007; Bertelsen 1997; Liu and Moore 2003]) and both ([Stärk et al. 2012]), these models variously have no supporting theory, do not support both features, or do not have the flexibility we need to prove properties over programs in general.

Initialization is described in the JVM specification as a procedure performed during the execution of a single instruction. We have chosen to represent this using a small-step approach, as a big-step approach is too unwieldy; however, the Java-level big-step semantics (and its equivalence with small-step semantics at the same level) gives us confidence in our representation.

Section 2 talks about Jinja, what it previously included, and how its semantics was represented. Sections 3 and 4 give a specification of the features that we added, based on the Java SE 8 specifications for Java and JVM [Gosling et al. 2015; Lindholm et al. 2015]. Section 5 briefly discusses the impact of these features on the structure of the semantics. Section 6 presents the actual syntax and semantics changes. Section 7 provides some details regarding the updated type safety and equivalence proofs. Section 8 presents the major details in our update to the Java to JVM compiler and the proof of its correctness. Section 9 provides some reflections on the work. Finally, Sections 10 and 11 discuss some related work, wrap up, and suggest future directions.

The Isabelle development for the work presented here can be found online at `https://github.com/susannahej/jinja-dci`.

2 Jinja

Jinja was developed to give a formal semantics for both Java and JVM byte code in a unified way. The authors wrote the semantics in the theorem prover Isabelle because this allowed them to write definitions and proofs based on this semantics. These proofs include type safety of the small-step semantics, equivalence of two semantics, and the correctness of a compiler from the Java level to the JVM level. The value of using a system like Isabelle to build a semantic framework is precisely the ability to prove these kinds of results, which is why we have chosen to extend this framework: in order to prove that a test behaves the same on two different programs, we need both a definition of the language and a framework that allows meta-reasoning at this level.

In this section, we will describe the features already covered by the Jinja framework to provide a basis for our extension.

2.1 Java Level

Jinja's Java-level semantics J defines 15 expressions: `new`, `Cast`, `Val`, `BinOp`, `Var`, `LAss` (local assignment), `FAcc` (field access), `FAss` (field assignment), `Call`, `Block`, `Seq`, `Cond`, `While`, `throw`, and `TryCatch`. Semantics are given for these in both big-step (evaluate) and small-step (reduce) style. Both styles are defined as inductive relations on pairs of expression and state relative to a given program P, and are written as $P \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle$ and $P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle$, respectively, where a state s is made of the heap h and the local variable mapping l.

FAcc:

$$\frac{P \vdash \langle e, s_0\rangle \Rightarrow \langle \texttt{addr}\ a, (h, l)\rangle \quad h\ a = \texttt{Some}(C, fs) \quad fs(F, D) = \texttt{Some}\ v}{P \vdash \langle e \bullet F\{D\}, s_0\rangle \Rightarrow \langle \texttt{Val}\ v, (h, l)\rangle}$$

FAccNull:

$$\frac{P \vdash \langle e, s_0\rangle \Rightarrow \langle \texttt{null}, s_1\rangle}{P \vdash \langle e \bullet F\{D\}, s_0\rangle \Rightarrow \langle \texttt{THROW NullPointer}, s_1\rangle}$$

FAccThrow:

$$\frac{P \vdash \langle e, s_0\rangle \Rightarrow \langle \texttt{throw}\ e', s_1\rangle}{P \vdash \langle e \bullet F\{D\}, s_0\rangle \Rightarrow \langle \texttt{throw}\ e', s_1\rangle}$$

Figure 1. Example rules from Jinja's Java-level big-step semantics

The small-step rules for `new` and the big- and small-step rules for `FAcc` can be seen in Figures 1 and 2. (The big-step and small-step rules for `new` are the same, since there are no subexpressions to reduce.)

In J the expression `new` takes one argument: the name of the class to be instantiated. In both semantic styles, there are two cases: the heap is out of memory or it is not. The former case results in an `OutOfMemory` exception being thrown; the latter results in the operation being completed. Each case is written as its own rule.

The expression `FAcc` takes three arguments: a subexpression e that should evaluate to an object pointer, the name of the field whose value is being fetched, and the name of the class that defines that field in the object being passed. The evaluation of e has three cases: e evaluates to a pointer, e evaluates to `null`, or e evaluates to a thrown expression. In big-step style, there is a rule for each of these cases. In small-step style, there are four rules: one reduces the subexpression; each of the others handles one these three cases. See also that this expression only works for fields belonging to a class instance, since e must evaluate to a pointer. Adding static fields requires storing them somewhere other than in a class instance, in order to ensure that they outlive any given object.

2.2 JVM Level

Jinja's JVM semantics covers 15 instructions: `load`, `store`, `push`, `new`, `getfield`, `putfield`, `checkcast`, `invoke`, `pop`, `return`, `iadd`, `goto`, `cmpeq`, `iffalse`, and `throw`. The rules for the execution of these instructions were written as a function `exec_instr` in small-step style. This function takes nine arguments: the instruction `i` to be performed, the program `P`, the heap `h`, the stack `stk`, the local variables `loc`, the current class `C_0`, the current method `M_0`, the program counter `pc`, and the frame stack `frs`. It returns a triple made up of an optional exception, the updated heap, and the updated frame stack. In Jinja, this triple makes up a *program state*.

RedNew:
$$\frac{\texttt{new\_Addr}\ h = \texttt{Some}\ a \qquad P \vdash C\ \texttt{has\_fields}\ FDTs \qquad h' = h(a \mapsto (C, \texttt{init\_fields}\ FDTs))}{P \vdash \langle \texttt{new}\ C, (h, l)\rangle \rightarrow \langle \texttt{addr}\ a, (h', l)\rangle}$$

RedNewFail:
$$\frac{\texttt{new\_Addr}\ h = \texttt{None}}{P \vdash \langle \texttt{new}\ C, (h, l)\rangle \rightarrow \langle \texttt{THROW OutOfMemory}, (h, l)\rangle}$$

FAccRed:
$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e \bullet F\{D\}, s\rangle \rightarrow \langle e' \bullet F\{D\}, s'\rangle}$$

RedFAcc:
$$\frac{h\ a = \texttt{Some}(C, fs) \qquad fs(F, D) = \texttt{Some}\ v}{P \vdash \langle \texttt{addr}\ a \bullet F\{D\}, (h, l)\rangle \rightarrow \langle \texttt{Val}\ v, (h, l)\rangle}$$

RedFAccNull:
$$\frac{}{P \vdash \langle \texttt{null} \bullet F\{D\}, s\rangle \rightarrow \langle \texttt{THROW NullPointer}, s\rangle}$$

RedFAccThrow:
$$\frac{}{P \vdash \langle \texttt{throw}\ e \bullet F\{D\}, s\rangle \rightarrow \langle \texttt{throw}\ e, s\rangle}$$

Figure 2. Example rules from Jinja's Java-level small-step semantics

```
"exec_instr (New C) P h stk loc C0 M pc frs =
  (case new_Addr h of
    None =>   (Some (sys_xcpt OutOfMemory), h,
                    (stk, loc, C0, M, pc)#frs)
  | Some a => (None, h(a |-> blank P C),
                (Addr a#stk,loc,C0,M,pc+1)#frs))"

"exec_instr (Getfield F C) P h stk loc C0 M pc frs =
  (let v      = hd stk;
       xp'    = if v=Null
                  then Some (sys_xcpt NullPointer)
                   else None;
       (D,fs) = the(h(the_Addr v))
   in (xp', h,
 (the(fs(F,C))#(tl stk),loc,C0,M,pc+1)#frs))"
```

Figure 3. Example rules from Jinja's JVM instruction execution function

The rules for new and getfield can be seen in Figure 3.

In Jinja's JVM the instruction new is written as New and as in J takes one argument: the name of the class to be instantiated. The rule has the same two cases as in J: the heap is out of memory or it is not. This is done using a branching case statement.

```
"exec (P, None, h, (stk,loc,C,M,pc)#frs) =
  (let  i = instrs_of P C M ! pc;
 (xp',h',frs') = exec_instr i P h stk loc C M pc frs
   in Some(case xp' of None => (None,h',frs')
           | Some a =>
     find_handler P a h ((stk,loc,C,M,pc)#frs)))"

| "exec _ = None"
```

Figure 4. Single step execution function for JVM level

The instruction getfield is written as Getfield and takes two arguments: the name of the field whose value is being fetched, and the name of the class that defines that field in the object being passed. The field's value is fetched from the object referenced by the pointer on top of the stack. This rule also has two cases: the pointer on top of the stack is Null or it is not. The former results in a NullPointer exception; the latter results in a completed operation. Like with FAcc at the Java level, this instruction looks up the field F in a class instance, using a pointer from the top of the stack.

The function exec_instr defined partially above is then wrapped inside a function exec shown in Figure 4, which takes a program and a program state and optionally returns the next program state. A program state is made up of an optional exception, a heap, and a frame stack. exec uses the top frame in the frame stack to determine the current instruction, which it then passes to exec_instr along with the other pieces of the frame. It gives the next state given by exec_instr if there is no exception; otherwise, it attempts to handle the exception instead by checking the exception table of each frame until a handler is found (and placed on top of the frame stack) or the frame stack is empty.

2.3 Extending Jinja

While Jinja represents a massive and impressive piece of work, there are still many features of Java and the JVM that it does not support. Our motivating example, in particular, involves proving that the behavior of a test does not change if the classes it uses ("touches") are unchanged. More accurately, we wish to prove that the algorithm for collecting "touched" classes given by [Gligoric et al. 2015] collects a set of classes large enough to have this property. Static instructions and dynamic class initialization are cornerstone features for class use, and are thus a place where this collection might easily be subtly incorrect. Therefore, we required a semantics that included both of these features, and chose to extend Jinja as Isabelle provides an especially good framework for proofs about algorithms of this sort. The latter in particular was not a straightforward extension.

We have also updated several of the large results of the Jinja framework to include statics and dynamic class initialization: type safety, equivalence of the two Java-level semantics, and the correctness of a compiler from the Java level to the JVM level. Due to the more complicated nature of dynamic class initialization, these updates required updated proof statements in addition to updated proofs. We will detail these changes in Sections 7 and 8.

It should be noted that Lochbihler [Lochbihler 2007] wrote an extension of Jinja, JinjaThreads, which adds threads to both Jinja and Jinja's JVM. We are very interested in the implications threads have on dynamic class initialization, but threads are largely orthogonal to our current proof goals, so we have left this combination as future work.

3 Static Instructions

In Java, classes can have static fields and methods: these fields and methods belong to the class rather than to any instantiation of it. A static instruction is one that uses or manipulates static fields or methods. The static instructions used in JVM byte code are as follows:

`getstatic` C F D - fetches the value of a static field F belonging to class C, defined by class D

`putstatic` C F D - assigns a value to a static field F belonging to class C, defined by class D

`invokestatic` M C n - calls the static method M of class C which takes n arguments

These instructions correspond to with the Java-level expressions `SFAcc` ($C \bullet_s F\{D\}$), `SFAss` ($C \bullet_s F\{D\} := e$), and `SCall` ($C \bullet_s M(es)$), respectively.

Static instructions, methods, and fields are a core feature of Java that see a lot of use. Because of this, it makes sense for a reasonable semantics for Java to include them. However, supporting static instructions requires deciding how and when the values of static fields are first set. In the semantics of Java, this is done during class preparation and initialization, so this is the approach we have chosen.

4 Dynamic Class Initialization

In the JVM, class initialization methods are called dynamically. Rather than initializing classes up front, Java waits until the the class is actually used. Because of this, compilers can make the decision to postpone the loading and linking of classes. Loading is the process of finding a binary representation of a class and using a class loader on that representation to create the class. This process is followed by linking, which includes verification (checking that the code is structurally correct), preparation (setting static fields to default values based on type), and resolution of symbolic references. A class must be loaded and linked before it is initialized. In our semantics, we assume that classes have been loaded, verified, and resolved ahead of time, but not necessarily prepared. (Unprepared classes are prepared when initialization is first called on them.)

The process of initialization can result in a number of errors. As a result, it can cause different behaviors depending on when it occurs in a program. For example, if initialization is only attempted inside of blocks with the proper error handlers, then the program may exit or continue gracefully in a way that would not be possible if the process were run prematurely.

Initialization checks are triggered by the use of a class or one of its subclasses. In particular, in the supported subset of JVM instructions, if a class C is not initialized, class initialization occurs when:

- an instruction among `new`, `getstatic`, `putstatic`, or `invokestatic` references C,
- one of C's subclasses is being initialized, or
- at startup of the JVM, if C is the designated initial class.

At the Java level, initialization is called by the same or corresponding events. If triggered by the evaluation of an expression, initialization is not called until the expression's subexpressions are completely evaluated.

Introducing dynamic class initialization into Jinja requires that the initialization procedure be called at every point that triggers it. The direct effect of this on the semantics is the addition of rules describing the procedure and calls to it within the rules that trigger it.

4.1 The Initialization Procedure

After it has been loaded and linked, a class C can be in one of four states:

1. `Prepared`: Loaded and linked, but not initialized
2. `Processing`[1]: Currently being initialized
3. `Done`: Fully initialized
4. `Error`: Initialization is in an erroneous state, perhaps due to an error in a previous initialization attempt (or in another thread in a multithreaded program)

The current state of a class C affects how the initialization of that class proceeds.

Once one of the initialization triggers listed above occurs, the following procedure is performed[2]:

1. Check current state.
   - If class C is currently being initialized (i.e., `Processing`), that means that this call is recursive,

[1] This case would be seen by the initialization procedure whenever a class's initialization method makes a static instruction call to one of its own static fields or methods, including instructions for getting and putting values from and to static fields. In this case, we would not want to re-call the initialization method.

[2] This procedure is simplified to reflect a world without threads, assertions, interfaces, or exceptions that take arguments, as we do not currently support these features.

i.e., inside of the initialization procedure; return without error[3].
- If class C is already fully initialized (i.e., `Done`), nothing else is required; return without error.
- If class C's initialization is in an erroneous state (i.e., `Error`), throw `NoClassDefFoundError`.
- Otherwise, continue to next step.

2. Mark C's initialization procedure as `Processing`.
3. If C has a superclass S, initialize S.
   - If this results in throwing an exception, mark C's initialization as being in an erroneous state. Then throw the same exception that S's initialization threw.
   - Otherwise, continue to next step.
4. Execute the class initialization method of C.
   - If execution completes normally, mark C's initialization as `Done` and return without error.
   - Otherwise, an exception E was thrown. Mark C's initialization as being in an erroneous state, then throw E.

Formalizing this procedure is the core of our extension to the semantics.

5 Impact on the Formal Semantics

Adding static instructions is a fairly straightforward extension to the existing semantics. However, dynamic class initialization is much less straightforward. Initialization is a procedure that is written into the JVM byte code specification as if it is meant to be done all at once inside the computation of a single instruction, and so it is most naturally represented in big-step style. As with anything in big-step, it is possible to transform this process into one done over small steps, and this is what we have chosen to do. However, as we went on to prove that the Java-level big-step implementation is both equivalent to the Java-level small-step approach, and is correctly simulated by the JVM-level small-step semantics, we are confident in our transformation.

In this approach, it is necessary to find a way to make sure that the results of step 4 of the initialization procedure are properly propagated, that control is returned to the calling location upon completion, and that the procedure is not called again by the same expression (or instruction). This means that the semantics must recognize and handle the returning of an initialization method, and that the semantic context must be aware of a just-run procedure. As the Java-level and JVM-level semantics have different structures, we have two separate but similar solutions to these problems.

At the Java-level, we added two runtime-only expressions. The first of these, `INIT`, is used to run steps 1 through 3 of the initialization procedure, and calls step 4. The second, `RI`, acts as a container for the body of the initialization method during step 4, so that it is kept in context to allow for post-processing upon completion.

At the JVM level, the scope of any method in the JVM is its corresponding frame - created when the method is invoked, and permanently removed upon its return or failure to catch an exception passed to it. We have adding a flag to this structure that can take one of several values used to signal the current role of a frame in an initialization procedure. This flag is used to guarantee that the post-processing behavior of the initialization procedure is followed, that the procedure is only called once per instruction, and that thrown errors are passed properly. We will go into more detail about this flag in Section 6.2.2. We also added helper functions that are called directly by `exec` in lieu of `exec_instr` when necessary. These functions handle the creation of initialization frames and the marking of classes as being in the `Error` state, and are called based on a frame's initialization flag.

These changes also have an impact on the proofs of type safety, semantic equivalence, and compiler correctness. In addition to creating new cases for each, the initialization procedure complicates what it means for a state to be correct.

6 Semantic Extensions

6.1 Updated Structure

In order to support statics and the initialization procedure, we first needed to add extra arguments to various constructors and functions. First, we added a flag to the field and method types to mark whether they are static. Second, we added a "static heap" (sh) to the program state for storing the static state of each prepared class, including its static fields and its initialization state flag.

In the Java-level small-step semantics we further added an "indicator boolean" to each side of the relation, which indicates whether the need for initialization has been checked for the current subexpression. The stepping relation is now written as $P \vdash \langle e, s, b\rangle \rightarrow \langle e', s', b'\rangle$.

At the JVM level, we additionally updated the frame type to include an initialization call status flag. A corresponding argument (ics) has been added to the `exec_instr` function.

6.2 Semantics of the Initialization Procedure

In this section, we translate the steps of the initialization procedure from Section 4.1 into formal operational semantics.

6.2.1 Java Level

The initialization procedure is performed at the Java level by two runtime-only expressions added for this purpose. To keep track of the context of a call to the initialization procedure, the below expressions have an "expression on hold" to return to upon completion:

- `INIT` C' $(Cs, b) \curvearrowleft e'$, which handles the initialization of class C'; Cs is the list of classes to be initialized as a consequence (with C' at the end), built up by a

[3]Note that if this was the result of a `getstatic` instruction call inside the initialization method, this means that the value fetched would be the default for the field, unless a `putstatic` instruction has already been performed.

NONE:

$$\frac{sh\ C = \texttt{None}}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh(C \mapsto (\texttt{sblank}\ P\ C, \texttt{Prepared}))), b\rangle}$$

PROCESSING:

$$\frac{sh\ C = \texttt{Some}(obj, \texttt{Processing})}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{INIT}\ C'\ (Cs, \texttt{True}) \curvearrowleft e', (h, l, sh), b\rangle}$$

DONE:

$$\frac{sh\ C = \texttt{Some}(obj, \texttt{Done})}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{INIT}\ C'\ (Cs, \texttt{True}) \curvearrowleft e', (h, l, sh), b\rangle}$$

ERROR:

$$\frac{sh\ C = \texttt{Some}(obj, \texttt{Error})}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{RI}(C, \texttt{THROW NoClassDefFoundError}); Cs \curvearrowleft e', (h, l, sh), b\rangle}$$

Figure 5. small-step rules for INIT, non-Prepared

sequence of step 3s, then removed as completed; b is a boolean indicating whether the class on the top of this list (if it exists) has performed steps 1 through 3 of its initialization procedure; e' is the expression on hold

- RI(C, e); $Cs \curvearrowleft e'$, which is a container for running the class initialization method of C and marking its initialization state; e is either an exception thrown to C's procedure during step 3, or the expression being run by step 4; Cs is the list of classes still to be initialized; e' is the expression on hold

The small-step rules for INIT are given in Figures 5, 6, and 7. Note that in the rules described in Figure 5 and in the first two rules in Figure 6, INIT's boolean is set to False, indicating that the head of the class list (C) has not yet had steps 1 through 3 performed. The big-step rules for RI are given in Figure 8.

The first rule for INIT, None, describes the case where class C has not yet been linked; it creates a blank instance of the class with default values for the static fields, inserts this instance into the static heap as the object associated with class C, sets C's initialization state to Prepared, then steps again to the same INIT expression with the new static heap.

The next three rules, Processing, Done, and Error, describe the first three branches in step 1 of the initialization procedure. In these cases, class C does not need to be initialized, and there are no further classes to check. For the first two, this means that the list of classes (Cs) is complete and ready to be initialized. They therefore step to INIT over

INITNONOBJECT:

$$\frac{sh\ C = \texttt{Some}(obj, \texttt{Prepared}) \quad C \neq \texttt{Object} \quad \texttt{class}\ P\ C = \texttt{Some}(D, r) \quad sh' = sh(C \mapsto (obj, \texttt{Processing}))}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{INIT}\ C'\ (D\#C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh'), b\rangle}$$

INITOBJECT:

$$\frac{sh\ C = \texttt{Some}(obj, \texttt{Prepared}) \quad C = \texttt{Object} \quad sh' = sh(C \mapsto (obj, \texttt{Processing}))}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{False}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{True}) \curvearrowleft e', (h, l, sh'), b\rangle}$$

INITRINIT:

$$\frac{}{P \vdash \langle \texttt{INIT}\ C'\ (C\#Cs, \texttt{True}) \curvearrowleft e', (h, l, sh), b\rangle \rightarrow \langle \texttt{RI}(C, C \bullet_s \texttt{clinit}([])); Cs \curvearrowleft e', (h, l, sh), b\rangle}$$

Figure 6. small-step rules for INIT, when C is Prepared

Cs, with the boolean set to True. In the Error case, however, C's initialization procedure is ended abruptly due to the Error state. The result is the same as if C's class initialization method itself had resulted in an uncaught exception: every class in Cs will receive an error in step 3, meaning their initializations also result in an error. This is handled by passing the list of classes to RI with the appropriate exception expression (THROW NoClassDefFoundError) as if produced by C's initialization method.

The first rule in Figure 6, InitNonObject, describes the case where C's state is Prepared and C has a superclass, so steps 2 and 3 are both performed. The hypothesis class $P\ C$ = Some(D, r) indicates C's direct superclass in P is D. Thus D is added to the list of classes to initialize, and C's initialization flag is set to Processing in the meantime. Note that after this step, steps 1 and 2 have been performed for class C, and that step 3 will be complete once D's initialization procedure completes.

The next rule, InitObject, describes the case where step 3 of the above procedure is skipped because the class being initialized is the class Object, which does not have a superclass. Thus Object's initialization flag is set to Processing as in step 2, and then INIT's boolean is set to True to indicate it is safe to proceed to step 4.

The next rule, InitRInit, describes the case where steps 1 through 3 of C's initialization procedure are complete. The next step is the execution of C's class initialization method clinit (a static method taking no arguments), which is carried out by wrapping a call to this method in the RI constructor, keeping C, Cs, and e'.

The final rule for INIT, given in both big- and small-step in Figure 7, describes how INIT is finally discharged once its list is completely initialized (and therefore empty). In

$$\frac{P \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle}{P \vdash \langle \texttt{INIT}\ C\ (\texttt{Nil}, b) \curvearrowleft e, s\rangle \Rightarrow \langle e', s'\rangle}\ \textsc{Final}$$

RedInit:

$$\frac{\neg \texttt{sub\_RI}\ e'}{P \vdash \langle \texttt{INIT}\ C\ (\texttt{Nil}, b) \curvearrowleft e', s, b'\rangle \rightarrow \langle e', s, \texttt{icheck}\ P\ C\ e'\rangle}$$

Figure 7. big-step rule and small-step rules for returning from INIT

RInit:

$$\frac{P \vdash \langle e, s\rangle \Rightarrow \langle \texttt{Val}\ v, (h', l', sh')\rangle \qquad sh'\ C = \texttt{Some}(sfs, i) \qquad sh'' = sh'(C \mapsto (sfs, \texttt{Done})) \qquad C' = \texttt{last}(C\#Cs) \qquad P \vdash \langle \texttt{INIT}\ C'\ (Cs, \texttt{True}) \curvearrowleft \texttt{unit}, (h', l', sh'')\rangle \Rightarrow \langle e_1, s_1\rangle}{P \vdash \langle \texttt{RI}\ (C, e); Cs \curvearrowleft e', s\rangle \Rightarrow \langle e_1, s_1\rangle}$$

RInitFail:

$$\frac{P \vdash \langle e, s\rangle \Rightarrow \langle \texttt{throw}\ a, (h', l', sh')\rangle \qquad sh'\ C = \texttt{Some}(sfs, i) \qquad sh'' = sh'(C \mapsto (sfs, \texttt{Error})) \qquad P \vdash \langle \texttt{RI}(D, \texttt{throw}\ a); Cs \curvearrowleft e, (h', l', sh'')\rangle \Rightarrow \langle e_1, s_1\rangle}{P \vdash \langle \texttt{RI}(C, e); D\#Cs \curvearrowleft e', s\rangle \Rightarrow \langle e_1, s_1\rangle}$$

RInitFailFinal:

$$\frac{P \vdash \langle e, s\rangle \Rightarrow \langle \texttt{throw}\ a, (h', l', sh')\rangle \qquad sh'\ C = \texttt{Some}(sfs, i) \qquad sh'' = sh'(C \mapsto (sfs, \texttt{Error}))}{P \vdash \langle \texttt{RI}(C, e); Nil \curvearrowleft e', s\rangle \Rightarrow \langle \texttt{throw}\ a, (h', l', sh')\rangle}$$

Figure 8. big-step rules for RI

big-step, the expression on hold is evaluated and the result returned. Similarly, in small-step control is returned to the expression on hold, e'. It is confirmed (for type safety reasons) that the held-over expression (e') does not contain any initialization-related subexpressions (INIT, RI, or a call to the method clinit). Further, the indicator boolean is set to true - as long as e' is one of the expressions described as triggering class initialization in Section 4 that could trigger C's initialization. This check will pass if the initialization expression was originally introduced by another of the small-step rules. These restrictions are here in order to facilitate type safety and equivalence in behavior between the big- and small-step semantics, as we will describe further in Section 7.2.

The first rule for RI, RInit, describes the non-error case: the expression e contained by RI evaluates to a value. Since this means C's initialization method has returned without error, C's initialization state is set to Done and the result is passed back to INIT along with the stack of classes still to be initialized. INIT's boolean is set to True. The class-being-initialized (C') is set as the last class in the combined list $C\#Cs$.

The other two rules describe what happens when e evaluates to an uncaught exception. In both cases, C's initialization state is set to Error. The first, RInitFail, handles the case when the list of classes left to be initialized is non-empty: the exception is passed down to the next class on the list, with the rest of the list still on hold. The second, RInitFailFinal, handles the case when the class list is finally depleted: the RI, including its expression on hold, are thrown away entirely, and the thrown exception is returned.

6.2.2 JVM Level

At the JVM level, the initialization procedure is controlled by an initialization call status flag *ics* added to each frame. Instead of exec_instr, the execution function exec calls a new helper function exec_step which uses the flag *ics* to determine the next step of execution. The flag's type has four constructors:

- Calling C Cs is a signal to exec to perform the initialization procedure on C, where Cs is the list of classes already collected during all step 3s so far (with C being the most recent)
- Called Cs indicates that the classes Cs are ready to have their initialization methods run (in order), as per step 4; if Cs is empty, the procedure is complete
- Throwing Cs a is a signal to exec to process the throwing of error a to the classes Cs as per the exception case of step 3
- No_ics is for when none of the above apply; i.e., there is no current initialization procedure

To begin this initialization procedure, when an instruction triggers the procedure for class C, it sets the current frame's flag to Calling C [] to begin the procedure, indicating that C is being initialized, and no other classes have been collected yet. At a high level, when an initialization procedure is triggered by an instruction, it will result in the collection of a list of superclasses to be initialized inside a Calling flag. When this list is completed, the flag becomes Called over the list, and the initialization methods for the classes on the list will be run in order until the list is empty or an error is thrown. If an uncaught exception is thrown by any of the initialization methods, the flag is set to Throwing over the remaining list and the thrown error. The remaining classes in the list are set to an erroneous initialization state (in order), then the exception is thrown from the original position of the initialization procedure call.

The above is achieved by then function exec_step behaving differently based on the current frame's *ics*. exec_step's definition is given in Figure 9, and its behavior can be summarized as follows:

```
"exec_step (Calling C' Cs) ...
 = exec_Calling C' Cs P h stk loc C M pc frs sh" |
"exec_step (Called (C'#Cs)) ...
 = (None, h, init_frame P C'#
   (stk,loc,C,M,pc,Called Cs)#frs, sh)" |
"exec_step (Throwing (C'#Cs) a) ...
 = (None, h, (stk,loc,C,M,pc,Throwing Cs a)#frs,
    sh(C' |-> fst(the(sh C')), Error))" |
"exec_step (Throwing [] a) ...
 = (Some a, h, (stk,loc,C,M,pc,No_ics)#frs, sh)" |
"exec_step ics ...
 = exec_instr (instrs_of P C M ! pc) P h stk loc
    C M pc ics frs sh"
```

Figure 9. Helper function for JVM-level single step execution (... indicates the omission of the remaining arguments: `P`, `h`, `stk`, `loc`, `C`, `M`, `pc`, `frs`, and `sh`)

- `Calling` *C Cs*: calls helper function `exec_Calling`, described below
- `Called` *C#Cs*: changes *ics* to `Called` *Cs* and places an initialization frame for *C* on top of the frame stack
- `Throwing` *C#Cs a*: sets *C*'s initialization flag to `Error` and changes *ics* to `Throwing` *Cs a*
- `Throwing` [] *a*: changes *ics* to `No_ics` and throws *a*
- Otherwise (`No_ics` or `Called` []), calls the current instruction; in the latter case, this instruction will know from this flag that it just called and completed the appropriate initialization procedure

The behavior of `exec_Calling` called on *C* is determined by the initialization status of *C*.

`exec_Calling` *C Cs* first checks that *C* has a static object in the static heap: if it does not, `sblank` is called to create this object, and the initialization flag remains unchanged to signal to `exec` to call this function again.

If *C* has a static object, then its associated initialization state is checked. In the cases of `Done` and `Processing`, *C* does not need to be initialized, so no initialization frame is created; the given arguments are returned as is.

In the case of `Error`, *ics* is set to `Throwing` *Cs* over the appropriate error in order to start the process of passing it down through the initialization procedure stack.

In the case of `Prepared`, *C*'s initialization procedure is allowed to proceed to its next steps. *C*'s initialization state is set to `Processing` (step 2). If *C* is `Object`, *ics* is set to `Called` *C#Cs*, to start the process of running the initialization methods of the classes collected. If it is not, then *ics* is modified to be `Calling` *D C#Cs*, where *D* is *C*'s direct superclass, signaling to `exec` to call `exec_Calling` on this class. *C* is collected into the list so that its initialization method will be run once *D*'s initialization procedure completes.

RedSFAccNone:

$$\frac{\neg(\exists b\ t.P \vdash C \text{ has } F, b : t \text{ in } D)}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh), b\rangle \rightarrow \langle \texttt{THROW NoSuchFieldError}, s, \texttt{False}\rangle}$$

RedSFAccNonStatic:

$$\frac{P \vdash C \text{ has } F, \texttt{NonStatic} : t \text{ in } D}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh), b\rangle \rightarrow \langle \texttt{THROW ICCError}, s, \texttt{False}\rangle}$$

where `ICCError` = `IncompatibleClassChangeError`

SFAccInitDoneRed:

$$\frac{P \vdash C \text{ has } F, \texttt{Static} : t \text{ in } D \qquad sh\ D = \texttt{Some}(sfs, \texttt{Done})}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh), \texttt{False}\rangle \rightarrow \langle C \bullet_s F\{D\}, (h, l, sh), \texttt{True}\rangle}$$

SFAccInitRed:

$$\frac{P \vdash C \text{ has } F, \texttt{Static} : t \text{ in } D \qquad \nexists sfs.\ sh\ D = \texttt{Some}(sfs, \texttt{Done})}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh), \texttt{False}\rangle \rightarrow \langle \texttt{INIT}\ D\ ([D], \texttt{False}) \curvearrowleft C \bullet_s F\{D\}, (h, l, sh), \texttt{False}\rangle}$$

RedSFAcc:

$$\frac{P \vdash C \text{ has } F, \texttt{Static} : t \text{ in } D \qquad sh\ D = \texttt{Some}(sfs, i) \qquad sfs\ F = \texttt{Some}\ v}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh), \texttt{True}\rangle \rightarrow \langle \texttt{Val}\ v, (h, l, sh), \texttt{False}\rangle}$$

Figure 10. small-step semantics for `SFAcc` expression

6.3 Semantics for New Instructions

Now that we have defined the semantics of initialization, we can give semantics to the JVM instructions that deal directly with static fields and methods.

6.3.1 `Getstatic`.

In Figure 10, we present the Java-level small-step rules defining the behavior of the `SFAcc` expression. Compare with `FAcc` as defined in Figure 2.

The first two rules are the error cases. Note that the preconditions for these cases do not overlap, and that these errors are checked against in all of the remaining rules.

The next two cases perform the initialization check, given that the field exists and is static. The initialization check is performed on *D*, the class defining the referenced field. In the first, the initialization check passes, so the indicator boolean is set to `True`. In the second, initialization is required, so the `SFAcc` expression is put on hold inside of an `INIT` expression set to initialize *D*.

The final case occurs when the boolean indicating all checks through initialization have been completed is `True`; note that this will occur as a result of one of the previous

SFAccInit:

$$\frac{\begin{array}{c}P \vdash C \text{ has } F, \texttt{Static} : t \text{ in } D \\ \nexists sfs.\ sh\ D = \texttt{Some}(sfs, \texttt{Done}) \\ P \vdash \langle \texttt{INIT}\ D\ ([D], \texttt{False}) \curvearrowleft \texttt{unit}, (h, l, sh)\rangle \Rightarrow \langle \texttt{Val}\ v', (h', l', sh')\rangle \\ sh'\ D = \texttt{Some}(sfs, i) \qquad sfs\ F = \texttt{Some}\ v\end{array}}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh)\rangle \Rightarrow \langle \texttt{Val}\ v, (h', l', sh')\rangle}$$

SFAccInitThrow:

$$\frac{\begin{array}{c}P \vdash C \text{ has } F, \texttt{Static} : t \text{ in } D \\ \nexists sfs.\ sh\ D = \texttt{Some}(sfs, \texttt{Done}) \\ P \vdash \langle \texttt{INIT}\ D\ ([D], \texttt{False}) \curvearrowleft \texttt{unit}, (h, l, sh)\rangle \Rightarrow \langle \texttt{throw}\ a, s'\rangle\end{array}}{P \vdash \langle C \bullet_s F\{D\}, (h, l, sh)\rangle \Rightarrow \langle \texttt{throw}\ a, s'\rangle}$$

Figure 11. Rule examples for big-step semantics for SFAcc expression

two rules: the original initialization check passed, or initialization was performed and returned without error. The indicator boolean is also set back to `False`, as the "just checked" promise was only made to this particular expression.[4]

The big-step rules for SFAcc are similar. Other than the lack of an indicator boolean, the main difference is that there are two rules for when the initialization check fails, given in Figure 11. They branch on whether INIT returns a value or an exception. Since the INIT expression returns back to the expression initially calling it (in this case, SFAcc), it is created with `unit` as the expression on hold. If INIT returns a value, the rest of the operation completes immediately (without another initialization check).

In Figure 12, we present the new case of `exec_instr` written for the `getstatic` instruction. In the style of Jinja, the instruction is written as `Getstatic` with arguments as described in section 3. Compare with the old definition given in Figure 3.

The cases here are the same as at the Java level: first, `exec_instr` checks that the field exists and is static. If either check fails, the exception is set accordingly and returned without further changed or checks.

If the field exists and is static, then the initialization status flag is checked: if it is `Called`, then initialization has already been called and returned without error, so the lookup into the static heap is completed and the value of the field is placed on the top of the stack. Note that in this way, `Called` is akin to the indicator boolean used in the Java-level small-step semantics, and that this case is the parallel of the final SFAcc case given in Figure 10.

If the initialization status flag is not `Called`, then an initialization check is performed. If the check passes then lookup

[4]Since Jinja's small-step semantics is written deterministically, with only one subexpression being rewritten at a time, no other subexpression will get the chance to misuse this promise.

```
"exec_instr
 (Getstatic C F D) P h stk loc C0 M0 pc ics frs sh
 = (let (D',b,t) = field P D F;
     xp'   = if ~(exists t b. P |- C has F,b:t in D)
             then Some(sys_xcpt NoSuchFieldError)
             else case b of
                    NonStatic => Some(sys_xcpt ICCError)
                  | Static => None
   in (case (xp', ics, sh D') of
         (Some a, _) =>
          (xp',h,(stk,loc,C0,M0,pc,ics)#frs,sh)
       | (_, Called Cs, _) =>
          let (sfs, i) = the(sh D');
              v = the(sfs F)
          in (xp', h,
          (v#stk,loc,C0,M0,pc+1,No_ics)#frs,sh)
       | (_, _, Some (sfs, Done)) =>
          let v = the (sfs F)
          in (xp', h,
             (v#stk,loc,C0,M0,pc+1,ics)#frs,sh)
       | _ =>
        (xp',h,
        (stk,loc,C0,M0,pc,Calling D' [])#frs,sh)
 ) )"
  where ICCError = IncompatibleClassChangeError
```

Figure 12. Semantics for `getstatic` instruction

is completed as in the previous case. Otherwise, the initialization status flag is set to `Calling` D', signaling to the `exec` function to perform initialization on D' (the class where the field is defined). These cases parallel the initialization check rules at the Java level, except that the first completes the lookup immediately instead of taking the extra step to set the indicator.

6.3.2 Putstatic and Invokestatic

The Java-level small-step rules for SFAss consist of seven cases: five are the same as the cases for SFAcc. The other two cases handle the contained subexpression: the first reduces the subexpression; the second handles this subexpression reducing to an uncaught exception. Both of these cases and the failed-initialization-check case are given in Figure 13. Note that the subexpression is fully reduced before the error and initialization checks are performed.

The rules for SCall consist of the same seven cases as SFAss.

The are six big-step rules each for SFAss and SCall: the same five as for SFAcc, plus a case to handle their respective subexpressions reducing to an uncaught exception.

The JVM-level rules for the `putstatic` and `invokestatic` instructions are written with the same cases as `getstatic`.

SFAssRed:

$$\frac{P \vdash \langle e, s, b\rangle \rightarrow \langle e', s', b'\rangle}{P \vdash \langle C \bullet_s F\{D\} := e, s, b\rangle \rightarrow \langle C \bullet_s F\{D\} := e', s', b'\rangle}$$

SFAssThrow:

$$\frac{}{P \vdash \langle C \bullet_s F\{D\} := (\texttt{throw}\ e), s, b\rangle \rightarrow \langle \texttt{throw}\ e, s, b\rangle}$$

SFAssInitRed:

$$\frac{\begin{array}{c}P \vdash C\ \texttt{has}\ F, \texttt{Static} : t\ \texttt{in}\ D \\ \nexists sfs.\ sh\ D = \texttt{Some}(sfs, \texttt{Done})\end{array}}{\begin{array}{c}P \vdash \langle C \bullet_s F\{D\} := (\texttt{Val}\ v), (h, l, sh), \texttt{False}\rangle \rightarrow \\ \langle \texttt{INIT}\ D\ ([D], \texttt{False}) \curvearrowleft C \bullet_s F\{D\} := (\texttt{Val}\ v), \\ (h, l, sh), \texttt{False}\rangle\end{array}}$$

Figure 13. Rule examples for SFAss

6.4 Updated Rules

Besides the addition of static instructions and the initialization procedure, some of the existing instructions must also have their semantics altered in the presence of these changes.

6.4.1 Return

The JVM-level `return` instruction (which has no Java-level expression equivalent) had to be updated to perform initialization post-processing for normally-returning class initialization methods. When the initialization frame for a class C (i.e., a frame with current method clinit and current class C) performs a `return` instruction, C's initialization state is set to `Done`. Furthermore, unlike in the case of a regular return, the frame's method was not invoked directly by the frame below it, so this lower frame is not changed in any way. (When a regular frame returns, the arguments passed to that instruction would be removed from the stack and the program counter would be incremented at this time.)

6.4.2 New

The `new` expression and instruction must be modified to call the initialization procedure, since it is one of the triggering instructions. In the original Java-level small-step semantics, recall from Figure 2 that there were two cases: the case where the heap had space, and the case where it did not. In Figure 14, the first two rules are these, except they now require the indicator boolean to be `True` (since initialization is checked before heap space is). The other two rules are the initialization check rules: one where it passes, and the other where it fails and the initialization procedure is begun on C.

The cases for the updated JVM-level `new` are the same as at the Java level; like with SFAcc, the initialization status flag of the frame is checked for `Called`.

RedNew:

$$\frac{\begin{array}{c}\texttt{new\_Addr}\ h = \texttt{Some}\ a \\ P \vdash C\ \texttt{has\_fields}\ FDTs \qquad h' = h(a \mapsto \texttt{blank}\ P\ C)\end{array}}{P \vdash \langle \texttt{new}\ C, (h, l, sh), \texttt{True}\rangle \rightarrow \langle \texttt{addr}\ a, (h', l, sh), \texttt{False}\rangle}$$

RedNewFail:

$$\frac{\texttt{new\_Addr}\ h = \texttt{None}}{\begin{array}{c}P \vdash \langle \texttt{new}\ C, (h, l, sh), \texttt{True}\rangle \rightarrow \\ \langle \texttt{THROW OutOfMemory}, (h, l, sh), \texttt{False}\rangle\end{array}}$$

NewInitDoneRed:

$$\frac{sh\ C = \texttt{Some}(sfs, \texttt{Done})}{P \vdash \langle \texttt{new}\ C, (h, l, sh), \texttt{False}\rangle \rightarrow \langle \texttt{new}\ C, (h, l, sh), \texttt{True}\rangle}$$

NewInitRed:

$$\frac{\nexists sfs.\ sh\ C = \texttt{Some}(sfs, \texttt{Done})}{\begin{array}{c}P \vdash \langle \texttt{new}\ C, (h, l, sh), \texttt{False}\rangle \rightarrow \\ \langle \texttt{INIT}\ C\ ([C], \texttt{False}) \curvearrowleft \texttt{new}\ C, (h, l, sh), \texttt{False}\rangle\end{array}}$$

Figure 14. Modified Java-level small-step rules for new

6.4.3 Getfield, Putfield, and Invoke.

The instructions `getfield`, `putfield`, and `invoke`, and their Java-level expression equivalents are also affected by the changes; the new static flag must be checked before the instruction is performed. This introduces a minor change to the existing rules, plus a new rule for each that handles the case when the field or method is marked as static.

7 Java-Level Type Safety and Semantic Equivalence

7.1 Type Safety

Part of the Java level of the Jinja framework is a collection of proofs leading to a proof that the small-step semantics is type safe: that starting with a well-typed expression guarantees that execution will never get stuck, and the end result it correctly typed. We extended these proofs to the updated semantics. Previously, type safety relied on starting with a well-typed expression in a well-formed program with a "conforming" state: the heap had to have type-correct values for every field of every object it contained, and the local variables had to have types that matched how they were used in well-typing the expression. Updating the type safety proofs required modifying heap conformance to only require type-correct values for *nonstatic* fields. Additionally, it required defining a few more such conformance properties (where an initialization expression is one of `INIT`, `RI`, and a static method call to class initialization method `clinit`):

- static heap conformance (`shconf`; written $P, h \vdash_s sh\surd$): akin to heap conformance; true if every prepared static

object in *sh* has a type-correct value for all of its class-defined static fields.
- expression initialization conformance (`iconf` P sh e): true if e's currently computing subexpression s is the only initialization subexpression, if any, and s is in a valid state in relation to sh
- indicator boolean conformance (`bconf`; written $P, sh \vdash_b (e, b)\surd$): where b is the small-step indicator boolean; requires that if b is `True`, then the currently evaluating subexpression is allowed to have called initialization on some class C whose initialization state in sh is either `Done` or `Processing`.

Each of these properties is preserved under reasonable conditions.

Small-step `TypeSafety` required `shconf`, `iconf`, and `bconf` of the initial state. The equivalence of the big- and small-step semantics explained in section 7.2 means that big-step is also type safe.

7.2 Equivalence of Big-Step and Small-Step

The Jinja framework also included a proof of the equivalence of the Java-level big- and small-step semantics. Extending this result to the updated semantics required a large amount of theory related to initialization expressions especially. Also, due to the addition of initialization and the slight difference between the states used by big- and small-step, the statement of this proof required some additional hypotheses: Small-step simulates big-step if the initial state is `iconf`, the starting indicator boolean is `bconf`, and the ending indicator boolean is `False`. Big-step simulates small-step if the initial state is `iconf` and `bconf`.[5] These conditions are necessary because invalid initialization expressions do not have the same behavior in small-step and big-step, and of course the small-step indicator boolean is meant to preserve context that must be present in order for behavior to be the same.

8 Compilation

Part of the contribution of Jinja was a compiler from the Java-level syntax to the JVM-level syntax, complete with a proof of correctness showing that the JVM code would simulate the Java code. As part of our work, we extended this compiler to include the new static instructions, and updated the proof of correctness to show that the JVM semantics continues to simulate the Java semantics.

8.1 Compiler from J to J1

Jinja does not compile immediately from the Java level (J) to JVM; instead, top-level code is first compiled into the intermediate language J1. J and J1 are nearly identical: the syntax of J1 is the same as J, except that local variable names are numbers instead of strings. Then while in J the local state is a partial function from strings to values, in J1 it is a list of values, referred to by index. The compilation function from J to J1 thus replaces variable names with the numbers in a fashion that maintains this relationship between variables and their values. As none of the expressions we added add or remove any variables to or from scope, extending this compilation to include our new syntax is straightforward.

The compilation of methods does require a small change to reflect that nonstatic methods include the `this` pointer in their list of arguments, whereas static methods do not.

Since J and J1 have nearly identical semantics, extending these was straightforward.

Note that J1 handles local variables in the same way as JVM byte code does. By performing this change first, the compilation to JVM is made much more straightforward.

8.2 Compiler from J1 to JVM

J1 and JVM are naturally more dissimilar than J and J1, but compilation from one to the other is nearly as straightforward: expressions are compiled into lists of instructions in a fairly intuitive manner. For the most part, the subexpressions' compilations come first (in the order of operations), followed by the instruction corresponding to the overall expression. Some of the more complex expressions that branch require some use of the `goto` instruction, and some `push`es and `pop`s are required to maintain the stack, but the static instructions we have added are compiled in much the same way as their nonstatic counterparts. The runtime-only initialization expressions `INIT` and `RI` compile to empty lists: as these expressions are only introduced during execution, and are never inside the bodies of methods, they do not need to be compiled. Furthermore, they do not have JVM-level instruction counterparts to be compiled into.

8.3 Behavioral Correctness

The updates to the proof of the correctness of the compilation from J to J1 follow fairly directly from the existing proof. As there are no behavioral changes between the new and updated rules between J and J1, these updates followed generally the same format as the existing proof.

The proof of the correctness for the J1 to JVM compilation required much larger changes. The original inductive hypothesis needed to be updated to accommodate the behavior of the added initialization expressions, in addition to adding a few new requirements to the old expressions.

Our updated inductive hypothesis is quite complicated, but the changes amount to this: the frames that simulate the behavior of initialization expressions have a different structure than those for all other expressions. They require the *ics* to be set appropriately, and possibly for the addition of an initialization frame. Further, running these frames results in a change to the *ics* rather than to the value stack. Finally, the conditions for these expressions depend on the ics and the formulation of the expression rather than on the current

[5] Extending back one small step only requires `bconf`, but `iconf` is necessary for `bconf` preservation, which is needed when extending to multiple steps.

instruction. We therefore had to define a multi-case inductive hypothesis, dependent on the type of expression.

Ultimately, we were able to show that a well-formed method body that can evaluate in J1 evaluates in the same way in Jinja's JVM (in a frame marked with `No_ics`). The resulting theory file is nearly three times as long as the original, a reflection of both the addition of new expression types, and the fact that the modified inductive hypothesis required some extra lemmas to help existing cases work as before.

8.4 Compiler Type Preservation

In addition to extending the compiler and its proof of behavioral correctness, we extended the proof that the compiler from J to JVM preserves well-typedness. As initialization expressions get compiled away, static instructions were the main addition. This was therefore a largely straightforward extension.

9 Reflections

This work is the product of a great deal of effort and refinement. Dynamic class initialization being both interruptive and involving an external-to-code, multi-step procedure resulted in many iterations between models and proofs before reaching the approach presented above. Some large changes to proof statements were also required - most notably, the overhaul of the inductive hypothesis for the correctness of compilation. Even removing the time spent iterating the model, updating the proofs was the product of many months. The resulting proof statements, however, have proven to be robust enough that model modifications within the extension only take a day or two to propagate.

Overall, the extended definition files tend to be about 1.5 to two times the length of the original files, and the proof files in the development tend to be about two to three times the length of the original files. The latter especially is a reflection of the pervasive nature of the changes made, and the amount of proof effort required to support them.

10 Related Work

Many semantics have been written for Java over the years. We choose to highlight here a couple of these that covered an especially large number of features.

The formal executable semantics of Java given using the Typol logical framework by Attali, Caromel, and Russo [Attali et al. 1998] includes dynamic linking, and claims to cover a large set of features, but they are unfortunately described in a system that seems to no longer be available. Even if it were, it did not include any JVM component.

K-Java [Bogdanas and Roşu 2015] is an impressive and largely complete executable semantics for Java 1.4, but the authors have chosen to consider dynamic class initialization solely a JVM feature. It also has no JVM semantics, which we need for our motivating work. Finally, the K framework does not support the ability to reason about algorithms over programs in the way we need.

ASM-Java [Stärk et al. 2012] presents an executable semantics for both Java and JVM using Abstract State Machines (ASMs) that includes dynamic class loading. Their system includes an executable compiler and a way to test code. Unfortunately, none of this is developed inside of or attached to a theorem prover such as Isabelle; these semantics exist to assist programmers rather than for verification purposes. As far as we are aware, Jinja and JinjaThreads [Lochbihler 2007], written in Isabelle, are the only other unified semantic model that includes both.

Atkey's CoqJVM [Atkey 2008] is an executable specification of JVM in Coq that includes static instructions and dynamically loads classes into a class pool, but does not perform actual class initialization.

Bertelsen [Bertelsen 1997] gives a semantics for Java byte code, in the form of functions and semantic rules, which represents a large subset of the JVM as specified in 1996, including statics and class initialization. This work was a thorough investigation that unearthed some of the corner cases of and errors in the JVM specification. However, it does not appear to have any supporting theory, nor to have been written up in any tool to be used for proofs. It also does not have a supporting model of Java or of its translation into byte code. Belblidia and Debbabi [Belblidia and Debbabi 2007] likewise gives a semantics that includes static instructions and class initialization, but do not include any theory support or related proofs.

M6 [Liu and Moore 2003] is a nearly complete executable model of JVM in ACL2 by Liu and Moore that can be used to run and derive properties of Java programs. Their class initialization is also invoked dynamically and performed small-step style, over multiple steps of computation. However, there is no associated Java semantics.

11 Conclusion and Future Work

In this paper, we presented an extension of the Jinja Java and JVM semantics to include static instructions and dynamic class initialization. We described the initialization procedure and the instructions that call it. We updated the rules of existing expressions and instructions to reflect the addition of static flags and initialization calls. We further extended the Java-level proofs of type safety and big- and small-step equivalence to hold on the updated semantics, including updating well-typing definitions and adding new conformance properties to reflect new requirements. Finally, we extended the compiler from the Java level to the JVM level and updated its proofs of correctness.

This updated semantics can be used to reason about Java and JVM programs with statics and dynamic class initialization. In the future, we intend to use it to prove the correctness of the class-collecting algorithm that motivated our work.

The semantics presented here could be added upon further to support other Java features such as interfaces, assertions, and threads. The last of these could be achieved by combining the work done here with Lochbihler's JinjaThreads [Lochbihler 2007], now that the structure and requirements of this addition are better understood. As many of the proofs we have updated and created are inductive and use the Isar proof style, making them fairly modular, most additions that did not change the overall structures (such as the state) would not require a great deal of overhaul. We have left updating the JVM-level type safety proofs and byte code verifier as future work.

Acknowledgments

This material is based upon work supported in part by the uder Grants CCF-1439957 and CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We would like to thank Milos Gligoric and Darko Marinov for providing the motivating problem and for their insight into the importance of initialization timing, and those who gave us feedback on this work and paper along the way. We are also grateful to the reviewers for their time, suggestions, and comments.

References

Robert Atkey. 2008. CoqJVM: An Executable Specification of the Java Virtual Machine Using Dependent Types. In *Proceedings of the 2007 International Conference on Types for Proofs and Programs (TYPES'07)*. Springer-Verlag, Berlin, Heidelberg, 18–32. http://dl.acm.org/citation.cfm?id=1786134.1786136

Isabelle Attali, Denis Caromel, and Marjorie Russo. 1998. A formal executable semantics for Java.

Nadia Belblidia and Mourad Debbabi. 2007. A Dynamic Operational Semantics for JVML. *Journal of Object Technology* 6, 3 (2007), 71–100. https://doi.org/10.5381/jot.2007.6.3.a2

Peter Bertelsen. 1997. Semantics of Java byte code. (1997).

Denis Bogdanas and Grigore Roşu. 2015. K-Java: a complete semantics of Java. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 445–456.

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 713–716.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java Language Specification: Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/html/index.html

Gerwin Klein and Tobias Nipkow. 2006. A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (July 2006), 619–695. https://doi.org/10.1145/1146809.1146811

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. The Java Virtual Machine Specification: Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

Hanbing Liu and J Strother Moore. 2003. Executable JVM model for analytical reasoning: A study. In *Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators*. ACM, 15–23.

Andreas Lochbihler. 2007. Jinja with threads. *The Archive of Formal Proofs. http://afp. sf. net/entries/JinjaThreads. shtml* (2007).

Robert F Stärk, Joachim Schmid, and Egon Börger. 2012. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media.