Under consideration for publication in Math. Struct. in Comp. Science

JNI Light: An Operational Model for the Core JNI

Gang Tan

Department of Computer Science and Engineering, Lehigh University

gtan@cse.lehigh.edu

Received 10 January 2012

Through foreign function interfaces (FFIs), software components in different programming languages interact with each other in the same address space. Recent years have witnessed a number of systems that analyze FFIs for safety and reliability. However, lack of formal specifications of FFIs hampers progress in this endeavor. We present a formal operational model, JNI Light (JNIL), for a subset of a widely used FFI—the Java Native Interface (JNI). JNIL focuses on the core issues when a high-level garbage-collected language interacts with a low-level language. It proposes abstractions for handling a shared heap, cross-language method calls, cross-language exception handling, and garbage collection. JNIL can directly serve as a formal basis for JNI tools and systems. We demonstrate its utility by proving soundness of a system that checks native code in JNI programs for type-unsafe use of JNI functions. The abstractions in JNIL are also useful when modeling other FFIs, such as the Python/C interface and the OCaml/C interface.

1. Motivation

Most modern programming languages support foreign function interfaces (FFIs) for interoperating with program modules developed in other programming languages. Recent years have witnessed a string of systems that analyze and improve FFIs for safety and reliability [Furr and Foster, 2006, Tan et al., 2006, Hirzel and Grimm, 2007, Tan and Morrisett, 2007, Tan and Croft, 2008, Kondoh and Onodera, 2008, Li and Tan, 2009, Lee et al., 2010]. However, lack of formal semantics of FFIs hampers progress in this domain. The available specifications of FFIs are in prose. Relying on prose specifications has at least two unpleasant consequences. First, prose specifications are often ambiguous and sometimes incomplete. The situation is especially acute for an FFI, whose two sides involve different programming models and language features. For instance, Lee *et al.* reported that Sun's HotSpot and IBM's J9 behave differently for four out of ten JNI test cases [Lee et al., 2010, Table 1]. In such situations, the best an FFI user can do is to perform experiments on particular implementations and make an educated guess. This may cause inconsistencies and unsoundness. Second, without formal semantics, tools and analyzers cannot provide rigorous claims about their strength. As a result, previous systems that target FFIs have to argue their hypotheses and claims informally. This leaves their strength in doubt.

While there have been many efforts in formalizing the semantics of programming languages, almost all have ignored the FFI aspect. The work by Matthews and Findler [2007] formalizes the interoperation between two high-level functional languages, one typed and the other untyped. While this formalism represents significant progress in modeling language interoperation, it does not apply to FFIs. Most FFIs are about the interaction between a high-level language and a low-level language (assembly languages, C, and C++) in a shared memory.

This paper presents the first formal operational model, named JNI Light (JNIL), for a subset of a shared-memory foreign function interface—the JNI interface. The major challenge for the modeling effort is to have the right abstractions to accommodate differences between the programming models of Java and native code, without unduly complicating the model. This is challenging because Java is a high-level OO language with a managed runtime and provides automatic garbage collection and exception handling. Native code, on the other hand, operates at a much lower level. It manually manages the heap and has no built-in exception-handling mechanism. JNIL proposes a set of abstractions to handle these differences. The abstractions make the JNIL model concise and largely straightforward.

We proceed as follows. We highlight key issues and abstractions in JNIL in Sec. 2. The formal semantics of JNIL is presented in Sec. 3. Java bytecode checking and Java safety theorems are in Sec. 4. In Sec. 5, we discuss applications of the JNIL model; we also present and prove soundness of a system that performs extended safety checking of native code. We sketch extensions of JNIL in Sec. 6 and future work in Sec. 7. We present related work in Sec. 8 and conclude in Sec. 9.

A preliminary version of this article was published in the Proceedings of the Eighth Asian Symposium on Programming Languages and Systems (APLAS 2010) [Tan, 2010]. The differences between the conference version and this article are described as follows. First, to demonstrate how JNIL can be used as a foundation to provide rigorous claims of JNI tools and systems, we have added the formalization and proof of soundness of a system that performs extended static checking of native code to catch errors of incorrect JNI function calls (in Sec. 5). Second, due to space limitation, the conference version does not include the full semantics and proofs of the safety theorems. In this version, we have added the full semantics and major lemmas used in the proofs.

2. Informal Discussion of JNIL

In this section, we informally discuss major challenges of modeling the JNI and highlight JNIL's solutions; formal treatment is left to Sec. 3. We also present examples that help understand the key aspects.

Background. The JNI [Liang, 1999] is Java's mechanism for interfacing with native code. A native method is declared in a Java class by adding the `native` modifier. Fig. 1 presents an `Item` class that contains a native `double` method, which doubles the `quantity`

Java code

```
class Item {
   private int quantity = 17;
   private native int double();
   public int quadruple() {int old = double(); double(); return old;}
   static {System.loadLibrary("Item");}
}
```

Native code

```
      // a reference to an Item object is at the top of the operand stack
   SLd r1, sp[0]                       // load the reference to r1
   GetField ⟨"Item", "quantity", Int⟩  // Get the value of the quantity field
   Pop r2                              // pop the quantity value to r2
   Add r3, r2, r2
   Push r1
   Push r3                             // set up the stack for SetField
   SetField ⟨"Item", "quantity", Int⟩
   Push r2
   Ret
```

Fig. 1. A Java class with a native method and an implementation of the native method in JNIL; it assumes arguments and results are passed on the stack.

field and returns the old value. Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. In the example, the `quadruple` Java method invokes the `double` method.

A native method is implemented in a low-level language such as C, C++, or an assembly language. Native code can use all the features provided by the native language. In addition, native code can interact with Java through a set of JNI interface functions (called JNI functions hereafter). For instance, the implementation of `double` can invoke `GetField` to get the value of the `quantity` field, and `SetField` to set the field to double the old value. Through JNI functions, native methods can inspect, modify, and create Java objects, invoke Java methods, catch and throw Java exceptions, and so on.

Two sides of JNIL. A model of the JNI needs both a Java-side language and a native-side language. The Java-side language of JNIL is a subset of the Java Virtual Machine Language (bytecode [Lindholm and Yellin, 1999]). The native-side language is a RISC-style assembly language augmented with a set of JNI functions (such as `GetField`/`SetField`). We choose to model an assembly language because native methods in C or C++ are compiled before loaded and linked into the JVM. Furthermore, there is less modeling overhead for an assembly language, allowing JNIL to concentrate on the interaction between Java and native code.

Many bytecode and JNI functions in JNIL work with *field IDs* and *method IDs*. For example, "GetField *fd*" gets the value of the field represented by *fd*. A field ID identifies a field by specifying three elements: a class name that the field belongs to, a field name, and its type. For example, the ID for the `quantity` field is $\langle$"Item", "quantity", $\mathsf{Int}\rangle$. A method ID has similar information as a field ID. A method ID may identify either a Java method (implemented in bytecode) or a native method (implemented in native code).

Fig. 1 presents an implementation of the native `double` method in the example `Item` class, where both `GetField` and `SetField` use the field ID of `quantity` to access the field.

Heap model. In the JNI, Java and native code reside in the same address space to avoid costly context switches. Consequently, JNIL needs to model a shared heap. However, modeling the shared heap poses challenges because Java's and native code's views of the heap are at different levels.

Being a high-level language, Java takes a high-level view: a heap is mathematically a map from labels to objects. The use of abstract labels hides many complexities of memory management. If a heap is rearranged and labels are renamed, the new heap is considered to be equivalent to the old one as long as the "graph" of the heap is preserved. Furthermore, in the high-level view, objects are storable values. There is no need to consider how objects are represented in memory. Previous Java models [Drossopoulou and Eisenbach, 1999, Flatt et al., 1999, Freund and Mitchell, 2003, Klein and Nipkow, 2006] adopt the high-level view. By contrast, native code takes a low level view: a heap is mathematically a map from addresses to primitive values. An object is represented in memory as a sequence of primitive values according to an object-layout strategy. Native code can perform address arithmetic, for example, to access elements of a Java array.

JNIL adopts an unusual *block model*: (1) a heap is a map from labels to blocks; (2) a block is a map from addresses (natural numbers) to primitive values. A block may hold the representation of a Java object, or may be a memory region allocated and owned by native code.

$$\begin{aligned} \mathit{Heap} &::= \mathit{Label} \rightharpoonup \left\langle\ \mathsf{blk} : \mathit{Block},\ \mathsf{own} : \mathit{Owner}\ \right\rangle \\ \mathit{Block} &::= \mathbb{N} \rightharpoonup \mathit{Value} \end{aligned}$$

A reference value, written as $\ell[i]$, identifies a location in block ℓ with offset i.

There are two major benefits of the block model. First, using abstract labels instead of addresses in the heap preserves the major benefit of the high-level heap model. It simplifies the specification of garbage collection (GC). In particular, there is no need to worry about whether GC moves objects because the resulting heap after moving is equivalent to the previous heap.† The second benefit of the block model is that it also accommodates the low-level view of native code. Values stored in blocks are primitive values. Address arithmetic is allowed within one block. Suppose a block with label ℓ holds the representation of a Java integer array, then Java may pass to native code a reference $\ell[i]$ that identifies where array elements are stored. Adding an offset n to $\ell[i]$ results in a new reference $\ell[i+n]$, which native code can use to access the n-th element of the array.

† We can imagine that there is a flatten function that maps a heap in the block model to a flat heap. A flat heap is just a map from addresses to values. Then a moving GC will change only the flatten function.

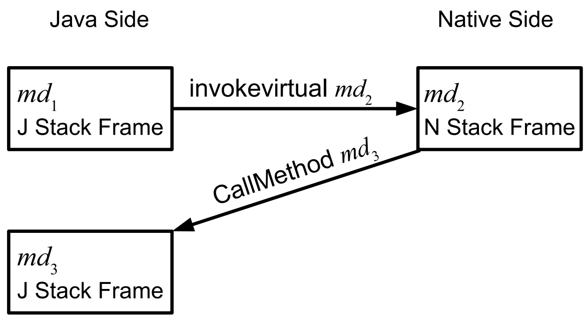

Fig. 2. An example of ping-pong behavior.

Object representation and ownership. Since JNIL's heap holds only primitive values, it is necessary to represent Java objects in the heap. JNIL is parametrized by a representation function, Rep : *Object* $\rightarrow$ *Block*, for the desire of not committing to any particular object-representation strategy. The representation function maps a Java object to a block. For instance, one representation can represent Java class instances and arrays in the following way:

$$\mathrm{Rep}(\langle\!\langle fd_1 = v_1, \ldots, fd_n = v_n \rangle\!\rangle_\phi) = \{0 \mapsto \mathrm{TypeRep}(\phi), 1 \mapsto v_1, \ldots, n \mapsto v_n\}$$
$$\mathrm{Rep}([\![v_0, \ldots, v_{n-1}]\!]_{\tau[n]}) = \{0 \mapsto \mathrm{TypeRep}(\tau), 1 \mapsto n, 2 \mapsto v_0, \ldots, n+1 \mapsto v_{n-1}\}$$

In the above, $\langle\!\langle fd_1 = v_1, \ldots, fd_n = v_n \rangle\!\rangle_\phi$ is a Java instance of class ϕ with fields fd_1 to fd_n; $[\![v_0, \ldots, v_{n-1}]\!]_{\tau[n]}$ is a Java array of size n with element type τ; TypeRep$(-)$ is a function for representing types as primitive values.

Each block in the heap has an owner: $\omega \in \{\mathrm{J}, \mathrm{N}\}$. A heap H is conceptually divided into a subheap owned by Java (J), written as $H|_\mathrm{J}$, and a subheap owned by native code (N), written as $H|_\mathrm{N}$. The reason for adding ownership is twofold. First, it helps specify Java's GC, which recollects locations only in the Java heap. Second, ownership information could be used to define a safety policy. For instance, if the policy is that native code should not access the Java heap, then the semantics of native load/store instructions could have the ownership checking built-in.

Cross-language method calls. Java and native code may engage in the so-called "ping-pong" behavior. Fig. 2 shows a graphical depiction of a sequence of method calls: a Java method with ID md_1 may invoke a native method with ID md_2, which in turn calls back another Java method with ID md_3. It is possible that md_3 invokes a second native method (not shown in the figure) and therefore the control can bounce back and forth between the Java and native sides.

To model cross-language method calls, we introduce in JNIL a *multi-language method-call stack* whose frames are either Java frames or native frames:

$$F \in \mathit{Frame} ::= \langle md, pc, s, a\rangle_\mathrm{J} \mid \langle md, pc, s, v_x, L\rangle_\mathrm{N}$$

A Java frame holds information for a Java-method execution, and a native frame for a native-method execution. Both kinds of frames include a method ID (md), a program counter (pc), and an operand stack (s). The operand stack is used for storing intermediate results and possibly for passing arguments and results of function calls. A Java frame also includes a local variable map (a), which holds values of local variables. A native frame also includes an exception reference (v_x) and a root set (L); we will discuss their uses shortly.

For the example in Fig 2, the shape of the method-call stack when the control is in md_3 is represented as follows (only method IDs are shown).

$$\langle md_3, \ldots \rangle_{\mathrm{J}} \cdot \langle md_2, \ldots \rangle_{\mathrm{N}} \cdot \langle md_1, \ldots \rangle_{\mathrm{J}} \cdot \epsilon \tag{1}$$

The top of the stack is on the left. We treat a stack as a list of frames and use "$F \cdot S$" for the concatenation of frame F and stack S and ϵ for the empty stack.

Cross-language exception handling. The JVM has a built-in mechanism for exception handling. We define *Java exceptions* to be those that are pending in a Java method. For a Java exception, the JVM checks if there is an enclosing try/catch statement that matches the exception type in the method. If not, it pops the method off the method-call stack and checks the next method.

An exception may also be pending on the native side; we call such exceptions *JNI exceptions*. For example, if the Java method md_3 in Fig. 2 throws an exception that is not handled by md_3, then it is a JNI exception pending in native method md_2. Native code itself may also throw exceptions by calling JNI functions such as `JNIThrow`. Furthermore, many JNI functions throw exceptions to indicate failures.

In contrast to how an exception is handled in a Java method, a JNI exception does not immediately disrupt the native method execution. The exception is recorded in the JVM, but the native method will keep executing. After the native method finishes execution and returns to a Java method, the exception becomes pending in the Java method and then the JVM mechanism for exceptions starts to take over.

Given this difference, the question is how to model the operational semantics when an exception becomes pending in a method-call stack that contains mixed Java and native frames. JNIL handles this issue by having different modes for indicating the presence of Java and JNI exceptions. A Java exception is indicated by a special *exception frame* $\langle \ell \rangle_{\mathrm{X}}$ at the top of the method-call stack, where ℓ is a reference to a `Throwable` object. A JNI exception is recorded in a native frame $\langle md, pc, s, v_x, L \rangle_{\mathrm{N}}$: the value v_x is `null` when no exception is pending and is ℓ with a pending JNI exception ℓ. JNIL's abstract machine proceeds differently for the two modes. Briefly, JNIL unwinds the stack for a Java exception and continues the execution of a native method for a JNI exception; we will discuss the details in the next section.

Registration of references. Java's GC is aware of only those references on the Java side. When native code retains references to Java objects, it has to register those references so that the GC will not collect the underlying objects. JNIL records the set of Java references available to a native method in a root set L. A root set is associated with a native frame so that its references are automatically "freed" when the native method finishes its execution. This semantics effectively models the so-called local references in the JNI.[‡]

3. Formal Operational Semantics of JNIL

We next present the core calculus of JNIL. A few simplifications are made to the model. First, arrays are not included. Second, it assumes a calling convention where arguments and results are passed on the operand stack when Java invokes native methods. Sec. 6 briefly discusses how to generalize the model to add arrays and to parametrize over calling conventions. The bytecode language is also simplified. Following Featherweight Java [Igarashi et al., 2001], we avoid the object initialization problem by having a single instruction for creating and initializing an object. There is also no modeling of interfaces, subroutine calls and returns, and various other Java features. They are orthogonal to the multilingual issues we are concerned with in FFIs. A notable missing feature in JNIL is concurrency. We believe it should be straightforward to formulate an interleaving semantics for multithreaded JNI programs based on a model of concurrent bytecode (e.g., [Petri and Huisman, 2008]).

Notation conventions. We write $\overline{e}$ for a list (or sequence) of elements e. The empty list is ϵ, and $e \cdot s$ is the concatenation of e with list s. Appending two lists is written as $s_1 \bullet s_2$. We write $[e_1, \ldots, e_n]$ for a finite list.

Given a function f, we write $f[x \mapsto v]$ for an updated function that agrees with f except that x is mapped to v. We write $f[\overline{x} \mapsto \overline{v}]$ for a function after a sequence of updates from $\overline{x}$ to $\overline{v}$. We write "*X Option*" for an option domain of X (analogous to ML's option types). We write None for the none value, and $\lfloor x \rfloor$ for some x. We use $\top$ for an arbitrary value.

3.1. JNIL *Programs*

A JNIL program is modeled as an environment that records information for classes and methods (Fig. 3). A program P includes maps from class names and method IDs to their respective definitions. In particular, $P(\phi).\mathsf{super}$ is the superclass of class ϕ, or None; $P(\phi).\mathsf{fields}$ is the list of fields declared in ϕ. We write $\mathrm{Fields}(P, \phi)$ for the list of all fields of ϕ, including the ones of its superclasses.

Java method and native method information are separated into two maps: P_{JM} for Java methods and P_{NM} for native methods. We write $\mathrm{JavaMD}(P)$ for the set of Java method IDs in P, and $\mathrm{NativeMD}(P)$ for the set of native method IDs. $P_{\mathrm{JM}}(md)$ contains a list of Java instructions (the code field), a list of exception handlers, and also type

[‡] The JNI also provides global and weak-global references. Global references are valid across multiple invocations of native methods and multiple threads. Weak global references are similar to global references except that the underlying objects can be garbage collected. These references are straightforward to model. Global references can be modeled as a global set of labels. Weak-global references have no impact on GC, although a JNI function for testing the validity of references needs to be exposed to native code. We omit their modeling in JNIL for brevity.

$$
\begin{array}{l}
P = P_{\mathrm{JC}} \cup P_{\mathrm{JM}} \cup P_{\mathrm{NM}} \\
P_{\mathrm{JC}} : \mathit{ClassName} \rightharpoonup \left\langle \ \mathsf{super} : \mathit{ClassName\ Option},\ \mathsf{fields} : \mathit{FID\ List} \ \right\rangle \\
P_{\mathrm{JM}} : \mathit{MID} \rightharpoonup \left\langle \begin{array}{l} \mathsf{code} : \mathit{JInstr\ List},\ \mathsf{handlers} : \mathit{Handler\ List}, \\ \mathsf{stype} : \mathit{CodeAddr} \rightharpoonup \mathit{Type\ List},\ \mathsf{vtype} : \mathit{CodeAddr} \rightharpoonup \mathit{JVarID} \rightarrow \mathit{Type} \end{array} \right\rangle \\
P_{\mathrm{NM}} : \mathit{MID} \rightharpoonup \langle \mathsf{code} : \mathit{NInstr\ List} \rangle
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{fd} \in \mathit{FID} & ::= & \langle \phi, \alpha, \tau \rangle & \mathit{md} \in \mathit{MID} & ::= & \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r \rangle \\
\tau \in \mathit{Type} & ::= & \mathsf{Int} \mid \mathsf{Cls}\ \phi \mid \mathsf{Top} & \eta \in \mathit{Handler} & ::= & \langle n_b, n_e, n_t, \phi \rangle
\end{array}
$$

$$
\phi \in \mathit{ClassName} = \mathit{String} \qquad \alpha \in \mathit{String} \qquad n \in \mathit{CodeAddr} = \mathbb{N} \qquad d \in \mathit{JVarID} = \mathbb{N}
$$

Fig. 3. JNIL programs.

$$
\begin{array}{rcl}
I \in \mathit{JInstr} & ::= & \mathit{arith} \mid \mathit{cond}\ n \mid \mathsf{push}\ v \mid \mathsf{pop} \mid \mathsf{localload}\ d \mid \mathsf{localstore}\ d \mid \mathsf{goto}\ n \\
 & \mid & \mathsf{getfield}\ \mathit{fd} \mid \mathsf{putfield}\ \mathit{fd} \mid \mathsf{new}\ \phi \mid \mathsf{invokevirtual}\ \mathit{md} \mid \mathsf{returnval} \mid \mathsf{throw} \\
\mathit{arith} \in \mathit{JArith} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \ldots \qquad \mathit{cond} \in \mathit{JCond} ::= \mathsf{ifeq} \mid \mathsf{ifne} \mid \mathsf{ifgt} \mid \ldots \\
 & & \\
\iota \in \mathit{NInstr} & ::= & \mathit{jfun} \mid \mathit{aop}\ r_d, r_s, \mathit{op} \mid \mathit{bop}\ r_s, r_t, \mathit{op} \mid \mathsf{Mov}\ r_d, \mathit{op} \mid \mathsf{Jmp}\ \mathit{op} \\
 & \mid & \mathsf{Ld}\ r_d, r_s[r_t] \mid \mathsf{St}\ r_d[r_t], r_s \mid \mathsf{Alloc}\ r_d, n \mid \mathsf{Free}\ r_s[n] \\
 & \mid & \mathsf{SLd}\ r_d, \mathsf{sp}[n] \mid \mathsf{SSt}\ \mathsf{sp}[n], r_s \mid \mathsf{SAlloc}\ n \mid \mathsf{SFree}\ n \mid \mathsf{Ret} \\
\mathit{jfun} \in \mathit{JNIFun} & ::= & \mathsf{GetField}\ \mathit{fd} \mid \mathsf{SetField}\ \mathit{fd} \mid \mathsf{NewObject}\ \phi \mid \mathsf{CallMethod}\ \mathit{md} \\
 & \mid & \mathsf{IsInstanceOf}\ \tau \mid \mathsf{JNIThrow} \mid \mathsf{ExnClear} \mid \mathsf{ExnOccurred} \\
\mathit{aop} \in \mathit{NArith} & ::= & \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Mul} \mid \ldots \qquad \mathit{bop} \in \mathit{NCond} ::= \mathsf{Beq} \mid \mathsf{Bneq} \mid \mathsf{Bgt} \mid \ldots \\
\mathit{op} \in \mathit{Operand} & ::= & r \mid n \qquad\qquad r \in \mathit{Register} ::= \mathsf{r1} \mid \mathsf{r2} \mid \ldots \mid \mathsf{r32}
\end{array}
$$

Fig. 4. Bytecode and native instruction sets.

information (stype and vtype). The type information is used when type checking Java methods and is irrelevant for operational semantics. $P_{\mathrm{NM}}(md)$ simply contains a list of native instructions. We abbreviate $P(md).\mathsf{code}[pc]$ to $P(md)@pc$, the instruction at pc in md.

Java types include Int type, class type ($\mathsf{Cls}\ \phi$), and Top type. For simplicity, JNIL omits types such as void and float. Two special class names, `object` and `throwable`, are assumed. We write Object and Throwable for "Cls `object`" and "Cls `throwable`", respectively. An exception handler, $\langle n_b, n_e, n_t, \phi \rangle$, catches exceptions of class ϕ by transferring the control to address n_t, if the program counter is in the range $[n_b, n_e - 1]$.

Fig. 4 presents the syntax of bytecode and native instructions. The bytecode instruction set is modeled after the Java Virtual Machine Language (JVML [Lindholm and Yellin, 1999]); we refer readers to the specification for a detailed discussion. The native instruction set includes instructions for manipulating the heap (load, store, allocation, and deallocation), a set of instructions for manipulating the operand stack (those instructions whose operators begin with S), a Ret instruction for returning, and a set of JNI functions. We use r for a register and op for an operand, which is either a register or a constant. Finally, we note that instructions for pushing to and popping from the

$$\begin{array}{rcl} S \in \mathit{Stack} & ::= & \overline{F} \mid \langle \ell \rangle_{\mathrm{X}} \cdot \overline{F} \\ F \in \mathit{Frame} & ::= & \langle md, pc, s, a \rangle_{\mathrm{J}} \\ & \mid & \langle md, pc, s, v_x, L \rangle_{\mathrm{N}} \\ s \in \mathit{OpStack} & ::= & \overline{v} \\ a \in \mathit{JVarMap} & ::= & \{0 \mapsto v_0, 1 \mapsto v_1, \ldots\} \\ L \in \mathit{RootSet} & ::= & \{\ell_1, \ldots, \ell_n\} \end{array} \qquad \begin{array}{rcl} H \in \mathit{Heap} & ::= & \mathit{Label} \rightharpoonup \left\langle \begin{array}{l} \mathsf{blk} : \mathit{Block}, \\ \mathsf{own} : \mathit{Owner} \end{array} \right\rangle \\ b \in \mathit{Block} & ::= & \mathbb{N} \rightharpoonup \mathit{Value} \\ v \in \mathit{Value} & ::= & n \mid \mathsf{null} \mid \ell[i] \\ \omega \in \mathit{Owner} & ::= & \mathrm{J} \mid \mathrm{N} \\ o \in \mathit{Object} & ::= & \langle\!\langle fd_1 = v_1, \ldots, fd_n = v_n \rangle\!\rangle_{\phi} \end{array}$$

$$R \in \mathit{RegFile} ::= \{\mathsf{r1} \mapsto v_1, \ldots, \mathsf{r32} \mapsto v_{32}\}$$

Fig. 5. JNIL runtime states $(S; H; R)$.

operand stack can be synthesized: "Push op" is "SAlloc 1; SSt sp[0], op" and "Pop r" is "SLd r, sp[0]; SFree 1".

Fig. 4 also includes a set of common JNI functions. Note that `GetField`, `SetField`, and `CallMethod` take field and method IDs as arguments. The JNI interface actually uses a two-step process to access a field (or call a method): first convert a string that represents the field (or method) to a field (or method) ID; the resulting ID is then used in operations such as `GetField`. JNIL omits the first step to avoid the need to axiomatize the conversion from strings to IDs.

Both the bytecode and native instruction sets include arithmetic and conditional branching instructions. Since their semantics is uninteresting, we will ignore those instructions hereafter. But we will feel free to include them in examples.

3.2. *Runtime states*

A runtime state is a triple $(S; H; R)$, where S is a method-call stack, H a shared heap, and R a register file. Its format is shown in Fig. 5. We have discussed the format of the method-call stack and the heap in the previous section. Recall that the heap holds only primitive values; objects are mapped to primitive values and stored in blocks. A value is either an integer n, a null value, or a reference value $\ell[i]$. We abbreviate $\ell[0]$ to ℓ.

3.3. *Operational semantics*

We will discuss only a subset of the rules to highlight JNIL's features; the full operational semantics is included in Appendix A. Overall, the operational semantics is modeled as a transition relation:

$$P \vdash (S; H; R) \longmapsto (S'; H'; R').$$

Fig. 6 presents evaluation rules at the top level. A state steps forward because of a Java step, a native step, or a GC step.

Fig. 7 presents a few Java heap operations that are used in the operational semantics. ReadFd, UpdFd, and AllocInst read a field, update a field, and allocate a new class instance, respectively. Blank(P, ϕ) returns an instance of class ϕ with its fields initialized to default values. Tag(H, ℓ) returns the runtime tag of a Java object at ℓ in H. If τ is a reference type, IsRefType(τ) holds.

$$\frac{P \vdash (S; H; R) \overset{\mathrm{J}}{\longmapsto} (S'; H'; R')}{P \vdash (S; H; R) \longmapsto (S'; H'; R')} \qquad \frac{P \vdash (S; H; R) \overset{\mathrm{N}}{\longmapsto} (S'; H'; R')}{P \vdash (S; H; R) \longmapsto (S'; H'; R')} \qquad \frac{(S; H) \overset{\mathrm{GC}}{\longmapsto} (S'; H')}{P \vdash (S; H; R) \longmapsto (S'; H'; R)}$$

Fig. 6. JNIL's top evaluation rules.

$$\mathrm{ReadFd}(H, \ell, fd) = \begin{cases} o(fd) & \text{if } H(\ell) = \langle \mathrm{Rep}(o), \mathrm{J} \rangle, \\ & \text{and } o = \langle\!\langle \ldots \rangle\!\rangle_\phi, \text{ and } fd \in \mathrm{dom}(o) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathrm{UpdFd}(H, \ell, fd, v) = \begin{cases} H[\ell \mapsto \langle \mathrm{Rep}(o[fd \mapsto v]), \mathrm{J} \rangle] \\ \quad \text{if } H(\ell) = \langle \mathrm{Rep}(o), \mathrm{J} \rangle, \text{ and } o = \langle\!\langle \ldots \rangle\!\rangle_\phi, \text{ and } fd \in \mathrm{dom}(o) \\ \text{undefined} \qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

$$\mathrm{AllocInst}(H, P, \phi) = (H \uplus \{\ell \mapsto \langle \mathrm{Rep}(o), \mathrm{J} \rangle\}, \ell)$$
$$\quad \text{where } \ell \notin \mathrm{dom}(H) \text{ and } o = \mathrm{Blank}(P, \phi)$$

$$\mathrm{Blank}(P, \phi) = \langle\!\langle fd_1 = \mathrm{Zero}(\tau_1), \ldots, fd_n = \mathrm{Zero}(\tau_n) \rangle\!\rangle_\phi,$$
$$\quad \text{where } \mathrm{Fields}(P, \phi) = [fd_1, \ldots, fd_n], \text{ and } fd_i = \langle \phi_i, \alpha_i, \tau_i \rangle, i \in [1..n]$$

$$\mathrm{Zero}(\mathsf{Int}) = 0 \qquad \mathrm{Zero}(\mathsf{Cls}\ \phi) = \mathsf{null}$$

$$\mathrm{Tag}(H, \ell) = \begin{cases} \phi & \text{if } H(\ell) = \langle \mathrm{Rep}(\langle\!\langle \ldots \rangle\!\rangle_\phi), \mathrm{J} \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathrm{IsRefType}(\tau) = \exists \phi.\ \tau = \mathsf{Cls}\ \phi$$

Fig. 7. Java heap operations.

The semantics of all instructions are included in Appendix A. We reproduce some rules in the main text to illustrate typical cases. Fig. 8 reproduces the rules for "getfield *fd*" (a bytecode instruction) and its counterpart JNI function "GetField *fd*" (used in native code). The semantics of "getfield *fd*" is deliberately partial. The abstract machine does not have a next state (that is, "getting stuck"), if block ℓ in H is not owned by Java, does not hold an object representation, or field *fd* is not in the domain of the representation (in these cases, $\mathrm{ReadFd}(H, \ell, fd)$ is undefined). The bytecode type system ensures such cases will not happen for well-typed bytecode programs when running in well-typed states.

The semantics of "GetField *fd*" is similar to "getfield *fd*", except for a couple of differences. First, no JNI exceptions should be pending. Recall that in a native stack frame $\langle md, pc, s, v_x, L \rangle_{\mathrm{N}}$ the value v_x records a pending JNI exception. The JNI manual specifies that "calling most JNI functions with a pending exception may lead to unexpected results". Consequently, most JNI functions requires v_x be null as a precondition. On this aspect, JNIL follows the specification of the JNI standard. JVM implementations, however, may implement different semantics. The experiments by Lee et al. [2010] showed that in such cases Sun's HotSpot continues running, while IBM's J9 crashes.

Second, some JNI functions may give native code extra references to Java objects. Since these references need to be registered with Java's GC, they are recorded in the root set of a native frame. The semantics of "GetField *fd*" adds the value of the field into the root set L, if that value is a reference value.

Cross-language method calls. The "invokevirtual *md*" instruction may invoke a Java

$P \vdash (\langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H; R) \overset{\mathrm{J}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and conditions hold,	then $S'; H'=$
getfield fd	$fd = \langle \phi, \alpha, \tau\rangle \quad s = \ell \cdot s_1$ $\mathrm{ReadFd}(H, \ell, fd) = v$	$\langle md, pc+1, v \cdot s_1, a\rangle_{\mathrm{J}} \cdot S; H$

$P \vdash (\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R) \overset{\mathrm{N}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and conditions hold,	then $S'; H' =$
GetField fd	$fd = \langle \phi, \alpha, \tau\rangle \quad s = \ell \cdot s_1$ $\mathrm{ReadFd}(H, \ell, fd) = v \quad v_x = \mathsf{null}$	$\langle md, pc+1, v \cdot s_1, \mathsf{null}, L'\rangle_{\mathrm{N}} \cdot S; H,$ where $L' = L \cup \mathrm{Roots}(v)$

Fig. 8. Operational semantics of "getfield *fd*" and "GetField *fd*".

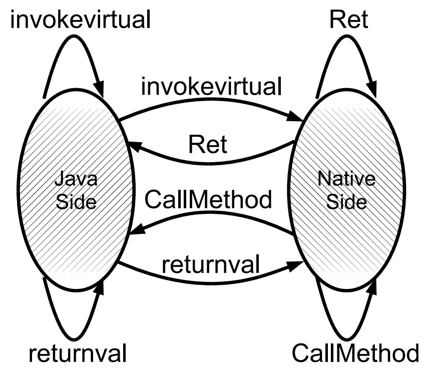

Fig. 9. Boundary-crossing instructions.

or a native method, depending on what kind of method *md* represents. If it invokes a native method, the execution context switches to the native side. returnval may return to a Java, or a native method. JNI function "CallMethod *md*" and native Ret are analogous, except they appear in native code. Fig. 9 presents a diagram depicting how contexts may switch as a result of method-call and return instructions.

Fig. 10 includes rules related to method calls and returns. If "invokevirtual *md*" invokes a Java method, a new Java frame is constructed and parameters are copied to the local variable map of the new frame (following the JVML specification). If it invokes a native method, a native frame is constructed and arguments are put in its operand stack (recall the calling convention). The auxiliary function NewFrame constructs either a Java frame

$P \vdash (\langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H; R) \overset{\mathrm{J}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and cond. hold,	then $S', H'=$
invokevirtual md_1	$md_1 = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $s = v_n \cdot \ldots \cdot v_1 \cdot \ell \cdot s_1 \quad \mathrm{Tag}(H, \ell) = \phi'$ $md' = \langle \phi', \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$	$\mathrm{NewFrame}(P, md', [\ell, v_1, \ldots, v_n])\cdot$ $\langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H$
returnval	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot \ell \cdot s', a'\rangle_{\mathrm{J}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \quad s = v_r \cdot s_1$	$\langle md', pc' + 1, v_r \cdot s', a'\rangle_{\mathrm{J}} \cdot S_1; H$
returnval	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot v \cdot s', v_x, L\rangle_{\mathrm{N}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \quad s = v_r \cdot s_1$	$\langle md', pc' + 1, v_r \cdot s', v_x, L'\rangle_{\mathrm{N}} \cdot S_1;$ $H, \quad$ where $L' = L \cup \mathrm{Roots}(v_r)$

$P \vdash (\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R) \overset{\mathrm{N}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and conditions hold,	then $S', H' =$
CallMethod md_1	$md_1 = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $s = v_n \cdot \ldots v_1 \cdot \ell \cdot s_1$ $\mathrm{Tag}(H, \ell) = \phi' \quad v_x = \mathsf{null}$ $md' = \langle \phi', \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$	$\mathrm{NewFrame}(P, md', [\ell, v_1, \ldots, v_n])\cdot$ $\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H$
Ret	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot v \cdot s', a'\rangle_{\mathrm{J}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \quad s = v_r \cdot s_1 \quad v_x = \mathsf{null}$	$\langle md', pc' + 1, v_r \cdot s', a'\rangle_{\mathrm{J}} \cdot S_1; H$
Ret	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot v \cdot s', v'_x, L\rangle_{\mathrm{N}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \quad s = v_r \cdot s_1 \quad v_x = \mathsf{null}$	$\langle md', pc' + 1, v_r \cdot s', v'_x, L\rangle_{\mathrm{N}} \cdot S_1; H,$ where $L' = L \cup \mathrm{Roots}(v_r)$

Fig. 10. Operational semantics of method calls and returns.

or a native frame:

$$\mathrm{NewFrame}(P, md, [v_1, \ldots, v_n]) = \begin{cases} \langle md, 1, \epsilon, a_{\top}[0 \mapsto v_1, \ldots, n-1 \mapsto v_n]\rangle_{\mathrm{J}}, \\ \quad \text{if } md \in \mathrm{JavaMD}(P), \\ \langle md, 1, [v_n, \ldots, v_1], \mathsf{null}, \mathrm{Roots}([v_1, \ldots, v_n])\rangle_{\mathrm{N}}, \\ \quad \text{if } md \in \mathrm{NativeMD}(P) \end{cases}$$

The semantics of returnval has two cases: returning to a Java method call or a native method call. Similar to “invokevirtual *md*”, “CallMethod *md*” may invoke either a Java or a native method. The JNI manual does not make it clear whether a native method is allowed to invoke another native method through “CallMethod *md*”. Our experiments confirmed that JVM implementations allow this behavior. Both rules for Ret are for the case of no pending exceptions; a different rule for Ret with a pending exception will be presented.

$P \vdash (\langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H; R) \overset{\mathrm{J}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and conditions hold,	then $S'; H' =$
throw	$s = \ell \cdot s_1$	$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H$

$P \vdash S; H; R \overset{\mathrm{J}}{\longmapsto} S'; H'; R$, if

S=	and conditions hold,	then $S', H' =$
$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S_1$	$\mathrm{Tag}(H, \ell) = \phi$ $P(md).\mathsf{handlers} = \overline{\eta}$ $\mathit{CorrectHandler}(\overline{\eta}, P, pc, \phi) = \mathsf{None}$	$\langle \ell\rangle_{\mathrm{X}} \cdot S_1; H$
$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S_1$	$\mathrm{Tag}(H, \ell) = \phi$ $P(md).\mathsf{handlers} = \overline{\eta}$ $\mathit{CorrectHandler}(\overline{\eta}, P, pc, \phi) = \lfloor n_t \rfloor$	$\langle md, n_t, \ell \cdot \epsilon, a\rangle_{\mathrm{J}} \cdot S_1; H$
$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S_1$		$\langle md, pc+1, s, \ell, L\rangle_{\mathrm{N}} \cdot S_1; H$

$$\mathit{CorrectHandler}(\epsilon, P, pc, \phi) = \mathsf{None}$$
$$\mathit{CorrectHandler}(\langle n_b, n_e, n_t, \phi'\rangle \cdot \overline{\eta}, P, pc, \phi) = \begin{cases} \lfloor n_t \rfloor & \text{if } n_b \leq pc < n_e \text{ and } P \vdash \mathsf{Cls}\ \phi <: \mathsf{Cls}\ \phi' \\ \mathit{CorrectHandler}(\overline{\eta}, P, pc, \phi) & \text{otherwise} \end{cases}$$

$P \vdash (\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R) \overset{\mathrm{N}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and conditions hold,		then $S', H' =$
JNIThrow	$s = \ell \cdot s_1$	$v_x = \mathsf{null}$	$\langle md, pc+1, s_1, \ell, L\rangle_{\mathrm{N}} \cdot S; H$
ExnClear			$\langle md, pc+1, s, \mathsf{null}, L\rangle_{\mathrm{N}} \cdot S; H$
ExnOccurred			$\langle md, pc+1, v \cdot s, v_x, L\rangle_{\mathrm{N}} \cdot S; H$ where $v = 0$ if $v_x = \mathsf{null}$, or 1 if $v_x = \ell$
Ret	$v_x = \ell$		$\langle \ell\rangle_{\mathrm{X}} \cdot S; H$

Fig. 11. Raising exceptions and exception handling in JNIL.

Exception handling. Fig. 11 shows rules that are related to exceptions. The throw instruction pushes an exception frame onto the method-call stack. Other bytecode instructions may also generate a Java exception. For instance, "getfield *fd*" generates an exception when the object reference on the operand stack is null. When such cases happen, a `Throwable` object is allocated and an exception frame is placed onto the stack. The formal definition of these cases are listed in Fig. 20 of the appendix. Real implementations create `Throwable` objects of different classes to indicate different kinds of exceptions. Our model simplifies this aspect by always allocating a `Throwable` object; this does not fundamentally affect program behavior.

When a Java exception is pending, JNIL unwinds the stack as shown in the second table of Fig. 11. There are three cases. If the next frame is a Java frame and there is no matched handler for the exception, the Java frame is removed. If the Java frame has a matched handler, then the control transfers to the handler. If the next frame is a native frame, the Java exception is recorded in the native frame (i.e., conceptually converted into a JNI exception) and the execution continues as normal from the next instruction in native code.

The last table in Fig. 11 shows how JNI exceptions are generated and handled. A JNI exception thrown by JNIThrow is recorded in the current native frame. Other JNI functions may also generate JNI exceptions and these cases are in Fig. 20 of the appendix. Native code can either clear the exception by ExnClear or return with the exception pending, in which case an exception frame is pushed onto the stack.

We present an example below showing how the method-call stack unwinds assuming 1) Java method md_1 calls native method md_2, which calls Java method md_3; 2) md_3 throws an exception; 3) md_3 and md_2 do not handle the exception, but md_1 handles the exception. Notice how md_3 and md_2 treat the exception differently.

$$\begin{array}{ll}
 & \langle \ell \rangle_{\mathrm{X}} \cdot \langle md_3, \ldots \rangle_{\mathrm{J}} \cdot \langle md_2, \ldots, \mathsf{null}, \ldots \rangle_{\mathrm{N}} \cdot \langle md_1, \ldots \rangle_{\mathrm{J}} \cdot \epsilon \quad \textit{//}md_3 \textit{ throws an exception} \\
\rightarrow & \langle \ell \rangle_{\mathrm{X}} \cdot \langle md_2, \ldots, \mathsf{null}, \ldots \rangle_{\mathrm{N}} \cdot \langle md_1, \ldots \rangle_{\mathrm{J}} \cdot \epsilon \quad \textit{//}md_3 \textit{ does not handle the exception} \\
\rightarrow & \langle md_2, \ldots, \ell, \ldots \rangle_{\mathrm{N}} \cdot \langle md_1, \ldots \rangle_{\mathrm{J}} \cdot \epsilon \quad \textit{//}md_2 \textit{ records } \ell \textit{ and continues execution} \\
\rightarrow & \langle \ell \rangle_{\mathrm{X}} \cdot \langle md_1, \ldots \rangle_{\mathrm{J}} \cdot \epsilon \quad \textit{//}md_2 \textit{ returns with a pending exception} \\
\rightarrow & \langle md_1, \ldots \rangle_{\mathrm{J}} \cdot \epsilon \quad \textit{//}md_1 \textit{ handles the exception}
\end{array}$$

GC Step. The GC rule is presented below. A set of blocks can be removed from the heap if they are part of the Java heap, their labels are disjoint from the roots of the stack, and they are unreachable from the rest of the Java heap.

$$\frac{L \subseteq \mathrm{dom}(H|_{\mathrm{J}}) \qquad L \cap \mathrm{Roots}(S) = \emptyset \qquad L \cap \mathrm{Reachable}((H|_{\mathrm{J}}) \setminus L) = \emptyset}{(S; H) \overset{\mathrm{GC}}{\longmapsto} (S; H \setminus L)}$$

Definition 1. The reachable set of labels from H is defined as all labels stored in the heap:

$$\mathrm{Reachable}(H) = \{\ell \mid \exists \ell_1, i, j.\ H(\ell_1).\mathsf{blk}(i) = \ell[j]\}$$

Definition 2. (Computing roots)

$$\text{Roots}(S) = \begin{cases} \emptyset & \text{if } S = \epsilon \\ \{\ell\} \cup \text{Roots}(S') & \text{if } S = \langle \ell \rangle_{\text{X}} \cdot S' \\ \text{Roots}(F) \cup \text{Roots}(S') & \text{if } S = F \cdot S' \end{cases}$$

$$\text{Roots}(\langle md, pc, s, a\rangle_{\text{J}}) = (\bigcup_{v \in s} \text{Roots}(v)) \cup (\bigcup_{d} \text{Roots}(a(d)))$$

$$\text{Roots}(\langle md, pc, s, v_x, L\rangle_{\text{N}}) = \text{Roots}(v_x) \cup L$$

$$\text{Roots}(v) = \begin{cases} \emptyset & \text{if } v = n \text{ or } \mathsf{null} \\ \{\ell\} & \text{if } v = \ell[i] \end{cases}$$

Notice that when computing the roots of a native frame the set of labels registered with GC (i.e., L) is included in the set. This is to ensure that GC will not recollect references registered by native code.

Note that the rule is nondeterministic and L can be as small as the empty set. It is also abstract and hides the implementation details of GCs. In fact, it accommodates all garbage collectors that are based on tracing, reference counting, or combinations of both; any such garbage collector computes a set of unreachable locations [Bacon et al., 2004]. Finally, recall that JNIL's heap model allows the rule to ignore the moving aspect of garbage collection.

4. Bytecode safety and GC safety

The JVM always performs bytecode verification before running a bytecode program. Therefore, type checking of bytecode can be considered an essential part of the JNI. The JNIL model also performs type checking of bytecode. The process largely follows a previous JVML model by Freund and Mitchell [2003]; we will highlight its main judgments and the safety theorems, but leave details to Appendix B.

Type checking JNIL programs. Judgment $\vdash P$ prog checks if a JNIL program P is well-typed. It ensures all classes and methods in P are well typed. Fig. 12 lists all judgments that are used in checking the well-typedness of programs (their rules are in Fig. 22 and 23 of the appendix).

We abuse the notation for subtyping and type checking values and will write $P \vdash \overline{\tau_1} <: \overline{\tau_2}$ for the subtyping between sequences of types, and $P, H \vdash \overline{v} : \overline{\tau}$ for checking sequences of values. Their rules are straightforward and therefore omitted.

The judgment for checking a Java method, "$P \vdash md$ $\mathsf{jmethod}$", utilizes the type information associated with the method; recall a Java method is associated with type information for the operand stack and local variables (see the fields `stype` and `vtype` in Fig. 3). We note that bytecode type checking does not infer these type information, but use them to check type consistency.

Suppose $P_{\text{JM}}(md) = \langle \overline{I}, \overline{\eta}, T_s, T_a\rangle$. Then $T_s(i)$ is the operand-stack type at address i and $T_a(i)$ is the type information for local variables at i. An operand-stack type is a list

$\vdash P$ prog	P is a well-typed JNIL program
$P \vdash \phi$ class	ϕ is a well-typed Java class
$P \vdash md$ $\mathsf{jmethod}$	md is a well-typed Java method
$P \vdash md$ $\mathsf{nmethod}$	md is a well-typed native method
$P, md, T_s, T_a \vdash \overline{I}@i$	The i-th instruction in $\overline{I}$ is well typed under typing annotation T_s and T_a
$P \vdash \tau_1 <: \tau_2$	τ_1 is a subtype of τ_2
$P, H \vdash v : \tau$	v has type τ
$P \vdash fd$ fid	fd is a well-formed field ID
$P \vdash md$ mid	md is a well-formed method ID
$P, T_s, T_a \vdash \eta$ $\mathsf{handles}$ $\overline{I}$	η is a valid handler in $\overline{I}$
$P \vdash \tau$ ty	τ is a valid type that can appear in a JNIL program (excluding the top type)

Fig. 12. Judgments used in type checking JNIL programs.

of types for values in the current operand stack. A local-variable type is a map from local variable IDs to types.

A well-typed Java method requires each bytecode instruction in the method be well typed; this is checked through the judgment "$P, md, T_s, T_a \vdash \overline{I}@i$". The following rule for getfield $\langle \phi, \alpha, \tau_1 \rangle$ is a typical case. It requires a reference of type $\mathsf{Cls}\ \phi$ at the top of the stack; after the instruction, the top of the stack is replaced by a value of the field's type. Types of local variables remain unchanged.

if $\overline{I}[i] =$	Conditions on T_s	Conditions on T_a	Other conditions
getfield $\langle \phi, \alpha, \tau_1 \rangle$	$P \vdash T_s(i) <: \mathsf{Cls}\ \phi \cdot \overline{\tau}$ $P \vdash \tau_1 \cdot \overline{\tau} <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$ $\langle \phi, \alpha, \tau_1 \rangle \in \mathrm{Fields}(P, \phi)$

Type checking a runtime state. Judgment "$P \vdash (S; H; R)$ state" checks if runtime state $(S; H; R)$ is well typed. It checks if (1) $H|_{\mathrm{J}}$ is a well-typed Java heap, and (2) S is a well-typed stack under P and the Java heap. The following table lists all judgments that are used to check the well-typedness of a runtime state (their rules are in Fig. 24 of the appendix). Checking well-typed Java heaps requires each heap object be well typed according to its runtime tag, as customary in such kind of type systems. Checking well-typed stacks not only requires every frame be well typed, but also requires the chain of frames be a well-typed call chain—each frame is the result of a call instruction in the caller method.

$P \vdash (S; H; R)$ state	$(S; H; R)$ is a well-typed state
$P \vdash H$ jheap	H is a well-typed Java heap
$P, H \vdash o : g$	Object o in H is consistent with runtime tag g
$P, H_J, S \vdash (R; H_N)$ nstate	$(R; H_N)$ is a well-formed native state
$P, H \vdash S$ stack	S is a well-typed method-call stack
$P, H \vdash F$ frame	F is a well-typed frame
$P \vdash S$ callchain	S is a stack with a valid call chain

Safety theorems. Type soundness of bytecode is expressed in the standard form of progress and preservation theorems.

Definition 3. $(S; H; R)$ is a terminal state if

(1) either $S = \langle md, pc, v_r \cdot s, a\rangle_{\mathrm{J}} \cdot \epsilon$ and $P(md)@pc = \mathsf{returnval}$,
(2) or $S = \langle md, pc, v_r \cdot s, \mathsf{null}, L\rangle_{\mathrm{N}} \cdot \epsilon$ and $P(md)@pc = \mathsf{Ret}$,
(3) or $S = \langle \ell \rangle_{\mathrm{X}} \cdot \epsilon$.

Theorem 1 (Java Progress). If $\vdash P$ prog, and $P \vdash (S_1; H_1; R_1)$ state, then

(1) either $(S_1; H_1; R_1)$ is a terminal state,
(2) or $\exists S_2, H_2, R_2.\ P \vdash S_1; H_1; R_1 \xmapsto{\mathrm{J}} S_2; H_2; R_2$,
(3) or $S_1 = \langle \ldots \rangle_{\mathrm{N}} \cdot S_1'$.

Theorem 2 (Java Preservation). If $\vdash P$ prog, and $P \vdash (S_1; H_1; R_1)$ state, and $P \vdash S_1; H_1; R_1 \xmapsto{\mathrm{J}} S_2; H_2; R_2$, then $P \vdash (S_2; H_2; R_2)$ state.

By the progress theorem, a well-typed state will be either a terminal state, a state that can take a Java step, or a state where native code is in control. It will never get stuck when bytecode is in control. By the preservation theorem, a well-typed state steps to another well-typed state when taking Java steps.

The proofs of Java progress and preservation are mostly standard. The first is by case analysis over the derivation of $P \vdash (S_1; H_1; R_1)$ state, and the second by case analysis over the Java step relation. Appendix C lists the major lemmas used in the proofs.

A GC step does not affect the type safety of bytecode, as the following theorem asserts:

Theorem 3 (GC Safety). If $\vdash P$ prog, $P \vdash (S; H; R)$ state, and $(S; H) \xmapsto{\mathrm{GC}} (S'; H')$, then $P \vdash (S'; H'; R)$ state.

As a final note, these safety theorems make no guarantee when a state takes a native step, reflecting the fact that native code is not checked by Java's type system and can cause havoc. Our formalization does include judgments for native code. However, their rules are vacuous in the sense that they allow any native code. For instance, the rule for "$P \vdash md$ nmethod" accepts any native method.

5. Applications of the JNIL model

The JNI specification does not mandate any checking of native methods. Native methods are notoriously unsafe and a rich source of software errors. Recent studies have reported hundreds of interface bugs in JNI programs [Furr and Foster, 2006, Tan and Croft, 2008, Kondoh and Onodera, 2008]. We list the most common kinds of pitfalls as follows:

— Violations of Java's type safety. Native code may pass parameters of wrong types when invoking JNI functions. For instance, GetField *fd* expects a Java object reference that contains a field corresponding to *fd*; but native code may pass in an incompatible reference. Another case of violating Java's type safety is that native code may perform direct reads and writes on memory that is part of the Java heap, destroying its invariants.

— Mishandling exceptions. When an exception is pending in native code, calling most JNI functions may lead to unexpected results.

— Mishandling JNI resources. The JNI interface resorts to manual management of certain resources (in the malloc/free style). One such example arises when managing pointers to primitive arrays. The scheme of manual mangement of resources has well-known problems such as double frees and using already released resources.

A number of systems have been designed and implemented to improve and find misuses of the JNI interface. They have overall improved the JNI's safety and security. We classify them into three broad categories:

— *New interface languages.* Jeannie [Hirzel and Grimm, 2007] is a language design that allows programmers to mix Java with C code using quasi-quoting. A Jeannie program is then compiled into JNI code by the Jeannie compiler. Jeannie helps programmers reduce errors. For instance, programmers can raise Java exceptions directly in Jeannie, avoiding the error-prone process of exception handling in native code.

— *Static checking.* Several recent systems employ static analysis to identify specific classes of errors in JNI code [Furr and Foster, 2006, Tan and Morrisett, 2007, Kondoh and Onodera, 2008, Li and Tan, 2009]. These bug finders have found hundreds of errors in real JNI programs.

— *Dynamic checking.* SafeJNI [Tan et al., 2006] combines Java with CCured [Necula et al., 2002] and inserts dynamic tests that check for safety violations. Going one step further, Jinn [Lee et al., 2010] automatically generates dynamic checks based on safety specifications in terms of finite-state machines.

We argue that it would be valuable to formalize the claims of these systems in JNIL and thus provide a rigorous foundation for their strength. We envision JNIL would be useful in the following ways:

— *Formal semantics of Jeannie.* We discussed Jeannie, a language that mixes Java with C code and is translated to JNI code. Jeannie does not come with formal semantics. An interesting way of defining Jeannie's semantics would be to map Jeannie programs to JNIL programs.

— *Soundness of JNI static checking.* JNIL can serve as a basis for proving that a JNI bug finder does not miss any errors of a certain kind. One way to show the soundness is to structure the system into two components: inference and verification. The first part infers annotations (e.g., in the form of types) and the second part performs verification with annotations as hints. Then the soundness theorem is to show that programs (with annotations) that pass the verification do not incur the kind of errors in question.

— *Soundness of JNI dynamic checking.* JNIL can also serve as a basis for showing the soundness of systems that insert dynamic checks for safety (e.g., SafeJNI [Tan et al., 2006]). One way to proceed is to have an "instrumented" semantics of JNIL in which dynamic checks are embedded into its transition rules. If a dynamic check fails, the system transits to an error state. The soundness theorem expresses that a state is either a terminal state, an error state, or a state that can progress. A more ambitious attempt to formalize dynamic checking is to treat the insertion of dynamic checks as a source-to-source rewriting system. The safety theorem would then show the resulting program is safe according to the vanilla semantics of JNIL.

In the above examples, JNIL alone would not be sufficient; we would also need formal models of other parts (e.g., a model of static checking). But JNIL provides a common foundation for such formal development to proceed. With additional constraints on the native code, JNIL makes it possible to prove properties of a multilingual system.

5.1. *Extended checking of native code*

As a concrete example demonstrating JNIL's utility, we next formalize a static checking system in JNIL that checks native code for violations of Java's type safety due to incorrect invocation of JNI functions. Also included is a soundness theorem showing that such errors will not occur in JNIL programs that pass the extended checking.

As we have discussed, native code mainly interacts with Java through JNI functions. These JNI functions require well-typed arguments for their correct functioning. For instance, if native code calls back a Java method through CallMethod, then Java expects the number of arguments and the types of arguments to match the method's type signature. A mismatch will likely crash the JVM and more severely result in security vulnerabilities; previous work [McGraw and Felten, 1999] demonstrated such kind of *type confusion* may allow attackers to control the JVM completely.

To prevent type confusion due to incorrect JNI function calls, the extended-checking system statically tracks Java types of object references in native code and ensures arguments of JNI function calls are of correct types. The system starts by augmenting native methods with extra type annotations:

$$P_{\mathrm{NM}} : \mathit{MID} \rightharpoonup \left\langle \begin{array}{l} \mathsf{code} : \mathit{NInstr\ List}, \\ \mathsf{stype} : \mathit{CodeAddr} \rightharpoonup \mathit{Type\ List}, \\ \mathsf{rtype} : \mathit{CodeAddr} \rightharpoonup \mathit{Register} \rightarrow \mathit{Type} \end{array} \right\rangle$$

If $P_{\mathrm{NM}}(md) = \langle \bar{\iota}, T_s, T_R \rangle$, type $T_s(i)$ is the type of the operand stack at address i and $T_R(i)$ is the type of registers at i. Similar to bytecode checking, the extended checking of native code takes type annotations as input to check type consistency, but does not per-

$\langle[\mathsf{Cls}\ \text{“Item”}], \{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Top}\}\rangle$
$\mathsf{SLd}\ \mathsf{r1}, \mathsf{sp}[0]$
$\langle[\mathsf{Cls}\ \text{“Item”}], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Top}\}\rangle$
$\mathsf{GetField}\ \langle\text{“Item”}, \text{“quantity”}, \mathsf{Int}\rangle$
$\langle[\mathsf{Int}], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Top}\}\rangle$
$\mathsf{Pop}\ \mathsf{r2}$
$\langle[], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Top}\}\rangle$
$\mathsf{Add}\ \mathsf{r3}, \mathsf{r2}, \mathsf{r2}$
$\langle[], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}\}\rangle$
$\mathsf{Push}\ \mathsf{r1}$
$\langle[\mathsf{Cls}\ \text{“Item”}], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}\}\rangle$
$\mathsf{Push}\ \mathsf{r3}$
$\langle[\mathsf{Int}, \mathsf{Cls}\ \text{“Item”}], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}\}\rangle$
$\mathsf{SetField}\ \langle\text{“Item”}, \text{“quantity”}, \mathsf{Int}\rangle$
$\langle[], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}\}\rangle$
$\mathsf{Push}\ \mathsf{r2}$
$\langle[\mathsf{Int}], \{\mathsf{r1} : \mathsf{Cls}\ \text{“Item”}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}\}\rangle$
Ret

Fig. 13. The native-method example in Fig. 1 with type annotations.

form type inference. Other systems such as J-Saffire [Furr and Foster, 2006] can perform type inference in native code.

Before presenting the checking rules, we use an example to demonstrate how the extended checking tracks Java types in native code's operand stack and registers. Fig. 13 presents type annotations for the example native method in Sec 2. Recall that it is an implementation of the native `double` method, which doubles the `quantity` field of the `Item` class. At each address i, the figure includes both the stack type $T_s(i)$ and the register-file type $T_R(i)$ in the format of $\langle T_s(i), T_R(i)\rangle$. Initially, the stack contains only one item of type Cls “Item”. After loading from the stack (i.e., $\mathsf{SLd}\ \mathsf{r1}, \mathsf{sp}[0]$), register `r1` gets type Cls “Item”. Next operation is a JNI function call ($\mathsf{GetField}$); as a result, the top of the stack type is removed and the type of the field is pushed onto the stack type. The effects of other instructions on types are also straightforward.

Rules for checking native methods and states. In the extended checking system, we change the rules of those judgments that are related to native methods and native states. These judgments include:

(1) $P \vdash md$ $\mathsf{nmethod}$, which checks that md is a well-formed native method.
(2) $P, H_J, S \vdash (R; H_N)$ nstate, which checks $(R; H_N)$ is a well-formed native state.
(3) $P, H \vdash \langle md, pc, s, v_x, L\rangle_{\mathrm{N}}$ frame, which checks $\langle md, pc, s, v_x, L\rangle_{\mathrm{N}}$ is a well-formed native frame.

The new rules are presented as follows. To distinguish between the system of extended checking and the basic bytecode checking, all judgments in this section will use $\vdash_*$ to replace $\vdash$.

$$\frac{\begin{array}{c} md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_{n+1} \rangle \qquad P(md) = \langle \bar{\iota}, T_s, T_R \rangle \\ T_s(1) = [\tau_n, \ldots, \tau_1, \mathsf{Cls}\ \phi] \qquad T_R(1) = \{\mathsf{r1} \mapsto \mathsf{Top}, \ldots, \mathsf{r32} \mapsto \mathsf{Top}\} \\ \forall i \in \mathrm{dom}(\bar{\iota}).\ P, md, T_s, T_R \vdash_* \bar{\iota}@i \end{array}}{P \vdash_* md\ \mathsf{nmethod}}$$

$$\frac{\begin{array}{c} \mathrm{TopFrame}(S) = \langle md, pc, s, v_x, L \rangle_{\mathrm{N}} \\ P(md) = \langle \bar{\iota}, T_s, T_R \rangle \\ P, H_J \vdash R : T_R(pc) \end{array}}{P, H_J, S \vdash_* (R; H_N)\ \mathsf{nstate}} \qquad \frac{\mathrm{TopFrame}(S) = \langle md, pc, s, a \rangle_{\mathrm{J}} \text{ or } \langle \ell \rangle_{\mathrm{X}}}{P, H_J, S \vdash_* (R; H_N)\ \mathsf{nstate}}$$

$$\frac{\begin{array}{c} md \in \mathrm{NativeMD}(P) \qquad pc \in \mathrm{dom}(\bar{\iota}) \\ P(md) = \langle \bar{\iota}, T_s, T_R \rangle \qquad P, H \vdash s : T_s(pc) \qquad P, H \vdash v_x : \mathsf{Throwable} \end{array}}{P, H \vdash_* \langle md, pc, s, v_x, L \rangle_{\mathrm{N}}\ \mathsf{frame}}$$

The rule for $P \vdash_* md$ nmethod sets up the initial stack type according to the type signature of the method, sets the types of all registers to be Top, and checks each instruction (its rules will be presented shortly). The rules for $P, H_J, S \vdash_* (R; H_N)$ nstate check that registers are of the specified types in the current register-file type and the rule for checking native frames ensures that the operand stack is of the specified stack type.

Fig. 14 and 15 present rules for checking native instructions. These rules are straightforward. For instance, the rule for "GetField *fd*" checks that there is a Java object reference at the top of the stack and the class of the reference must be a subtype of the one specified in the field ID. The new stack type after the instruction has the field's type at the top. The register-file type is unchanged as "GetField *fd*" does not modify registers.

Native-code safety theorem. To characterize what kind of errors the extended checking can capture, we add a distinguishing error state JTypeError (JNI type errors) to JNIL's operational semantics. We also add rules that specify when the abstract machine steps to the error state; these rules are in Fig. 16. For instance, GetField $\langle \phi, \alpha, \tau \rangle$ steps to the error state (1) when the operand stack is empty, (2) or when the top of the stack is an integer value, (3) or when the Java reference at the top of the stack is not of the class specified in the field ID, (4) or when the read-field operation fails (happens when, e.g., the field is not in the object being accessed).

The safety theorem expresses that a JNIL program that passes the extended checking will not result in a JNI type error. The proof of the theorem is by a straightforward case analysis over the instruction at the current program counter.

Theorem 4. If $\vdash_* P$ wf, and $P \vdash_* (S_1; H_1; R_1)$ state, then $\neg(P \vdash S_1; H_1; R_1 \longmapsto \mathsf{JTypeError})$.

We stress that the extended checking is meant to demonstrate the utility of the JNIL model and does not eliminate every possible JNI error. For instance, a native memory-store instruction can still change the Java state and cause havoc. In the presence of

$P, md, T_s, T_R \vdash_* \bar{\iota}@i$, if

$\bar{\iota}[i] =$	and the following conditions hold
GetField fd	$fd = \langle\phi, \alpha, \tau_1\rangle \in \text{Fields}(P, \phi)$ $\quad i+1 \in \text{dom}(\bar{\iota})$ $P \vdash T_s(i) <: \mathsf{Cls}\ \phi \cdot \overline{\tau}$ $\quad P \vdash \tau_1 \cdot \overline{\tau} <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
SetField fd	$fd = \langle\phi, \alpha, \tau_1\rangle \in \text{Fields}(P, \phi)$ $\quad i+1 \in \text{dom}(\bar{\iota})$ $P \vdash T_s(i) <: \tau_1 \cdot \mathsf{Cls}\ \phi \cdot T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
NewObject ϕ	$\text{Fields}(P, \phi) = [\langle\phi_1, \alpha_1, \tau_1\rangle, \ldots, \langle\phi_n, \alpha_n, \tau_n\rangle]$ $\quad i+1 \in \text{dom}(\bar{\iota})$ $P \vdash T_s(i) <: \tau_n \cdot \ldots \cdot \tau_1 \cdot \overline{\tau}$ $\quad P \vdash \mathsf{Cls}\ \phi \cdot \overline{\tau} <: T_s(i+1)$ $P \vdash T_R(i) <: T_R(i+1)$
CallMethod md_1	$md_1 = \langle\phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$ $\quad i+1 \in \text{dom}(\bar{\iota})$ $P \vdash T_s(i) <: \tau_n \cdot \ldots \cdot \tau_1 \cdot \mathsf{Cls}\ \phi \cdot \overline{\tau}$ $\quad P \vdash \tau_r \cdot \overline{\tau} <: T_s(i+1)$ $T_R(i+1) = \{\mathsf{r1} \mapsto \mathsf{Top}, \ldots, \mathsf{r32} \mapsto \mathsf{Top}\}$
IsInstanceOf τ	$\text{IsRefType}(\tau)$ $\quad i+1 \in \text{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: \tau_1 \cdot \overline{\tau}$ $\quad \text{IsRefType}(\tau_1)$ $P \vdash \mathsf{Int} \cdot \overline{\tau} <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
JNIThrow	$i+1 \in \text{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: \mathsf{Throwable} \cdot T_s(i+1)$ $P \vdash T_R(i) <: T_R(i+1)$
ExnClear	$i+1 \in \text{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
ExnOccurred	$i+1 \in \text{dom}(\bar{\iota})$ $\quad P \vdash \mathsf{Int} \cdot T_s(i) <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$

Fig. 14. Extended checking of JNI functions.

native code, a preservation theorem for well-typedness of the Java heap can only be proved for a comprehensive protection system such as the Robusta JVM [Siefers et al., 2010]; formalization of such systems is left for future work.

6. Extensions

Th extension of JNIL to support Java arrays is mostly standard and we omit its formal presentation. The only complication in the extension is that the JNI treats arrays with primitive types differently from arrays of reference types. For instance, the GetIntArray-Elements function returns a pointer to the first element of the array and native code can then perform address arithmetic to access array elements. JNIL can accommodate direct pointers to Java arrays since its heap model allows address arithmetic within blocks.

Another simplification in JNIL is that it assumes a calling convention that passes arguments and results through the operand stack when Java interfaces with native code. However, the calling convention varies greatly in reality, depending on compilers and architectures. We next sketch how to extend JNIL to parametrize over calling conventions.

Data for native method calls are passed through *machine resources*, which are either registers or slots on the operand stack.

$$rd ::= r \mid \texttt{sp}[n]$$

$P, md, T_s, T_R \vdash_* \bar{\iota}@i$

when $\bar{\iota}[i] =$	and the following conditions hold
$\mathsf{Mov}\ r_d, op$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $\quad P \vdash T_R(i)[r_d \mapsto \tau] <: T_R(i+1)$ where $\tau = \mathsf{Int}$ if $op = n$, and $\tau = T_R(i)(r)$ if $op = r$
$\mathsf{Jmp}\ n$	$n \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(n)$ $\quad P \vdash T_R(i) <: T_R(n)$
$\mathsf{Ld}\ r_d, r_s[r_t]$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $P \vdash T_R(i)[r_d \mapsto \mathsf{Top}] <: T_R(i+1)$
$\mathsf{St}\ r_d[r_t], r_s$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
$\mathsf{Alloc}\ r_d, n$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $P \vdash T_R(i)[r_d \mapsto \mathsf{Top}] <: T_R(i+1)$
$\mathsf{Free}\ r_s[n]$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
$\mathsf{SLd}\ r_d, \mathsf{sp}[n]$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: \tau_0 \cdot \ldots \cdot \tau_n \cdot \overline{\tau}$ $\quad P \vdash T_s(i) <: T_s(i+1)$ $P \vdash T_R(i)[r_d \mapsto \tau_n] <: T_R(i+1)$
$\mathsf{SSt}\ \mathsf{sp}[n], r_s$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: \tau_0 \cdot \ldots \cdot \tau_n \cdot \overline{\tau}$ $P \vdash \tau_0 \cdot \ldots \cdot (T_R(i)(r_s)) \cdot \overline{\tau} <: T_s(i+1)$ $\quad P \vdash T_R(i) <: T_R(i+1)$
$\mathsf{SAlloc}\ n$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash \underbrace{\mathsf{Top} \cdot \ldots \cdot \mathsf{Top}}_{n} \cdot T_s(i) <: T_s(i+1)$ $P \vdash T_R(i) <: T_R(i+1)$
$\mathsf{SFree}\ n$	$i+1 \in \mathrm{dom}(\bar{\iota})$ $\quad P \vdash T_s(i) <: \tau_1 \cdot \ldots \cdot \tau_n \cdot T_s(i+1)$ $P \vdash T_R(i) <: T_R(i+1)$
Ret	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r \rangle$ $\quad P \vdash T_s(i) <: \tau_r \cdot \overline{\tau}$

Fig. 15. Extended checking of native instructions.

We use $\mathsf{sp}[n]$ for the stack slot with offset n from the top of the operand stack.

We write $\mathrm{GetR}(s, R, rd)$ for a getter function that retrieves the value of resource rd from (s, R). We write $\mathrm{UpdR}(s, R, rd, v)$ for the setter function; it returns the new state (s', R'). We abuse the notation so that the getter and setter functions also work for a list of resources.

A calling convention is specified by two functions: (1) $\mathcal{P}_a([\tau_1, \ldots, \tau_n])$ tells what machine resources are used to pass n arguments that are of types τ_1 to τ_n; (2) $\mathcal{P}_r(\tau_r)$ tells what machine resources are used to pass a result of type τ_r. These functions take types as arguments because some calling conventions use types to decide what resources to use in function calls and returns.

Suppose $\mathcal{P}_a([\tau_1, \ldots, \tau_n]) = ([rd_1, \ldots, rd_n], k_a)$. Then it specifies a convention where the i-th argument is passed in resource rd_i; in addition, k_a tells the size of the extra stack frame for holding arguments. There are two validity requirements for this function. First, the resources should be disjoint. Second, if $rd_i = \mathsf{sp}[o]$, then $0 \le o < k_a$. Function

$P \vdash (\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R) \overset{\mathrm{N}}{\longmapsto} \mathsf{JTypeError}$, if

$P(md)@pc =$	and the following holds
GetField $\langle\phi,\alpha,\tau\rangle$	(1) either $\lvert s\rvert = 0$; (2) or $s = v \cdot s_1$ but $v = n$ for some integer n; (3) or $s = \ell \cdot s_1$ but $P, H \not\vdash \ell : \mathsf{Cls}\ \phi$; (4) or $s = \ell \cdot s_1$ but $\mathrm{ReadFd}(H, \ell, \langle\phi,\alpha,\tau\rangle)$ is undefined.
SetField $\langle\phi,\alpha,\tau\rangle$	(1) either $\lvert s\rvert < 2$; (2) or $s = v \cdot v' \cdot s_1$ but $v' = n$ for some integer n; (3) or $s = v \cdot \ell \cdot s_1$, but $P, H \not\vdash v : \tau$; (4) or $s = v \cdot \ell \cdot s_1$, but $P, H \not\vdash \ell : \mathsf{Cls}\ \phi$; (5) or $s = v \cdot \ell \cdot s_1$, but $\mathrm{UpdFd}(H, \ell, \langle\phi,\alpha,\tau\rangle, v)$ is undefined.
NewObject ϕ	$\mathrm{Fields}(P,\phi) = [\langle\phi_1,\alpha_1,\tau_1\rangle, \ldots, \langle\phi_n,\alpha_n,\tau_n\rangle]$ (1) either $\lvert s\rvert < n$; (2) or $s = v_n \cdot \ldots \cdot v_1 \cdot s_1$, but $P, H \not\vdash v_i : \tau_i$ for some $i \in [1..n]$;
CallMethod md_1	$md_1 = \langle\phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ and (1) either $\lvert s\rvert < n+1$; (2) or $s = v_n \cdot \ldots \cdot v_1 \cdot v' \cdot s_1$, but $P, H \not\vdash v_i : \tau_i$ for some $i \in [1..n]$; (3) or $s = v_n \cdot \ldots \cdot v_1 \cdot v' \cdot s_1$ but $v' = n$ for some integer n; (4) or $s = v_n \cdot \ldots \cdot v_1 \cdot \ell \cdot s_1, \mathrm{Tag}(H,\ell) = \phi'$ but $P \not\vdash \mathsf{Cls}\ \phi' <: \mathsf{Cls}\ \phi$.
IsInstanceOf τ	(1) either $\lvert s\rvert = 0$; (2) or $s = v \cdot s_1$ but $v = n$ for some integer n.
JNIThrow	(1) either $\lvert s\rvert = 0$; (2) or $s = v \cdot s_1$ but $v = n$ for some integer n; (3) or $s = \ell \cdot s_1$, but $P, H \not\vdash \ell : \mathsf{Throwable}$
Ret	$v_x = \mathsf{null}$ and $md = \langle\phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ (1) either $\lvert s\rvert = 0$; (2) or $s = v \cdot s'$, but $P, H \not\vdash v : \tau_r$; (3) or $S = \langle md', pc', s', a'\rangle_{\mathrm{J}} \cdot S_1$ and $\lvert s'\rvert < n+1$; (4) or $S = \langle md', pc', s', v_x', L\rangle_{\mathrm{N}} \cdot S_1$ and $\lvert s'\rvert < n+1$
Ret	$v_x = \ell$ and $P, H \not\vdash \ell : \mathsf{Throwable}$

Fig. 16. JNI type errors.

$\mathcal{P}_r(\tau_r)$ is similar. Suppose $\mathcal{P}_r(\tau_r) = (rd_r, k_r)$. It specifies a convention where the result is in rd_r and the size of the extra stack frame for holding the result is k_r.

Since JNIL passes all arguments and results on the stack in a left-to-right order, it effectively uses the following calling convention:

$$\mathcal{P}_a([\tau_1, \ldots, \tau_n]) = ([\mathtt{sp}[n-1], \ldots, \mathtt{sp}[0]], n)$$
$$\mathcal{P}_r(\tau_r) = (\mathtt{sp}[0], 1)$$

As another example, the `cdecl` convention passes arguments on the stack in a right-to-left order and the return value in `r1`. It can be specified by the following:

$$\mathcal{P}_a([\tau_1, \ldots, \tau_n]) = ([\mathtt{sp}[0], \ldots, \mathtt{sp}[n-1]], n)$$
$$\mathcal{P}_r(\tau_r) = (\mathtt{r1}, 0)$$

With this calling-convention specification, JNIL's operational semantics can be modified to parametrize over the calling convention. For instance, the following rule is for the case when a native method is invoked through $\mathsf{invokevirtual}$. The calling convention is used to put the arguments at the appropriate places. Note the notation $(\top)^{\mathsf{sz}}$ stands for an operand stack with sz number of uninitialized values.

$$\frac{\begin{array}{c} P(md)@pc = \mathsf{invokevirtual}\ md_1 \qquad md_1 = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r \rangle \in \mathrm{NativeMD}(P) \\ s = v_n \cdot \ldots \cdot v_1 \cdot \ell \cdot s_1 \qquad \mathrm{Tag}(H, \ell) = \phi' \qquad md' = \langle \phi', \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r \rangle \\ \mathcal{P}_a([\mathsf{Cls}\ \phi', \tau_1, \ldots, \tau_n]) = ([rd_0, rd_1, \ldots, rd_n], \mathsf{sz}) \\ V = [\ell, v_1, \ldots, v_n] \qquad \mathrm{UpdR}((\top)^{\mathsf{sz}}, R, [rd_0, rd_1, \ldots, rd_n], V) = (s', R') \end{array}}{P \vdash (\langle md, pc, s, a \rangle_{\mathrm{J}} \cdot S; H; R) \xrightarrow{\mathrm{J}} (\langle md', 1, s', \mathsf{null}, \mathrm{Roots}(V) \rangle_{\mathrm{N}} \cdot \langle md, pc, s, a \rangle_{\mathrm{J}} \cdot S; H; R')}$$

7. Discussions and future work

JNIL is designed to be a minimal formalism to capture the core language-interoperation issues in the JNI. The relationship between JNIL and the JNI is similar to that between Featherweight Java [Igarashi et al., 2001] and Java. Consequently, JNIL's design aims to follow the JNI standard, not a specific implementation. For instance, representation of Java objects is abstract in the semantics. In the same vein, Java's GC is specified abstractly using reachability. On the other hand, there are places where the JNI standard is unclear or ambiguous. For instance, it is unclear what happens if a native method invokes another native method through the JNI call-back functions. Such cases were resolved by a careful consideration of the semantics and also experiments in real implementations.

To stay minimal, it is necessary for JNIL to make simplifications. We believe most of these do not affect the claims that are made about the semantics. Nevertheless, it is important to list the major simplifications:

- JNI uses specific functions to construct field and method IDs from strings and class objects, while JNIL uses field and method IDs directly. Related is the issue of class objects. JNI provides `FindClass` for converting a class name to an object that represents the class. By contrast, JNIL uses class names directly in functions such as IsInstanceOf.
- JNI provides different methods for processing data of different types. For instance, `GetIntField` accesses an integer field and `GetFloatField` accesses a float field. This is the case for many other operations, including Java method invocation and array processing. Therefore, one type of mistakes in JNI programming is calling wrong methods; for instance, it is wrong to call `GetFloatField` with a field ID that represents an integer field. JNIL hides this problem by using polymorphic operators; for example, "GetField *fd*" takes a field ID of any type.
- There is only one exception class in JNIL, while real JNI implementations creates objects of different classes to indicate different kinds of exceptions.
- JNIL models only local references. JNI also provides global and weak-global references.

Another notable missing feature in JNIL is concurrency. There have been several attempts at modeling Java concurrency at the bytecode level (see BicolanoMT [Petri and Huisman, 2008] for a recent attempt). Based on a model of concurrent bytecode, it should be straightforward to formulate an interleaving semantics for multithreaded, mixed bytecode and native code. A more ambitious attempt is to consider the effect of memory models on the semantics. It is unclear how to reconcile differences between the Java Memory Model [Manson et al., 2005] and the memory model of a native architecture. Related to concurrency is the use of `JNIEnv` pointers. JNI functions are invoked indirectly through a `JNIEnv` pointer, which is thread local. Since JNIL includes only sequential semantics, it omits the `JNIEnv` pointer.

One future work is to develop methodology to evaluate JNIL. We plan to develop machine-checked semantics of JNIL in Coq. The native-side language of JNIL will use our recently built Coq model of the Intel x86-32 machine code [Morrisett et al., 2011]. In this model, the semantics of x86 instructions is defined by a translation to a small RTL (register transfer list) intermediate language. It has an operational, small-step semantics based on which we extracted an executable OCaml emulator. Using the emulator, we have performed extensive model validation by comparing it against real x86 processors; over 10 million instruction instances have been tested and verified in about 60 hours. The same methodology for model construction and validation will be used when constructing the JNIL Coq model. Building on top of the high-fidelity native x86 language, it will need to add machine-checked semantics of Java bytecode and JNI functions. Several projects have developed machine-checked semantics of Java bytecode [Moore and Porter, 2002, Klein and Nipkow, 2006, Pichardie, 2006]. We plan to build upon Bicolano [Pichardie, 2006], a recent formalization of Java bytecode semantics in Coq. Bicolano builds on an extensible framework [Czarnik and Schubert, 2007], which will make our development of JNIL modular by reusing much of the sequential semantics of bytecode. Same as our x86 model, the formalized JNIL model will be executable so that it will be possible to run benchmark programs to compare against implementations. This will serve as an important step to validate the JNIL model.

Although this paper targets the JNI, the abstractions in JNIL apply broadly when modeling other foreign function interfaces, including the CLR, the Python/C interface, and the OCaml/C interface. All these interfaces share the same core issues as the JNI: a shared heap, cross-language method calls, cross-language exceptions, and others.

8. Related work

The block heap model in JNIL takes inspiration from Leroy and Blazy's block memory model in the CompCert project [Leroy and Blazy, 2008]. They use the block memory model to specify the semantics of C-like languages and verify correctness of program transformations. We use the block model to reconcile differences between a high-level, garbage-collected OO language and a low-level language. The bytecode language in JNIL bears many similarities to the JVML$_f$ model by Freund and Mitchell [2003]; the native language is similar to Morrisett *et al.*'s stack-based typed assembly language [Morrisett et al., 2002]. JNIL's emphasis is on proposing abstractions for modeling language-interoperation issues in FFIs.

Previous work proposed preliminary formalisms that capture certain aspects of the JNI. Furr and Foster justified J-Saffire's soundness on a formalization of a subset of the JNI [Furr and Foster, 2008]. It models only the native side, and treats Java objects opaquely. Jinn [Lee et al., 2010] describes safety constraints of the JNI using finite-state machines. JNIL models both sides of the interface and proposes abstractions that address issues including a shared heap, cross-language method calls, exception handling, and the impact of garbage collection; these issues have not been addressed by previous efforts.

There have been a few systems for modeling various aspects of the interoperation of two safe high-level languages. Already mentioned in the introduction, the work by Matthews and Findler [2007] formalizes the interoperation between simply typed lambda calculus (as a stand-in for ML) and untyped lambda calculus (as a stand-in for Scheme). Their formalization focuses on high-level interoperation issues such as value conversion and abstracts away low-level details. The work by Trifonov and Shao [1999] presents a type and effect system for reasoning about the interoperation of two safe languages when they have different systems of computational effects. Compared to these models, JNIL is at a much lower level and exposes details including stack frame layout and garbage collection. These low-level details cannot be ignored when modeling the interaction between high-level and low-level languages.

More remotely related is the work of modeling general multi-language systems. This includes the formalization of COM [Pucella, 2002], a language-neutral binary standard for the interaction of component-based software, and the formalization of a subset of the intermediate language of .NET [Gordon and Syme, 2001], which is specially designed to be compatible with multiple languages.

9. Conclusions

Most real software systems are multilingual. A safe software system depends on its building blocks and their interoperation. Even if each building block is safe in some language model with respect to some safety policy, without safe interoperation between languages there would be no safety guarantee on the whole system. Therefore, modeling and reasoning about language interoperation is critical to the safety and security of software systems. JNIL is a formal model that covers the core JNI. Its abstractions elegantly reconcile the differences between a high-level OO language and a low-level language. It can directly be used to provide a formal foundation for systems that analyze the JNI. We believe its concepts can be generalized to model other FFIs.

Acknowledgments

We would like to thank Andrew W. Appel and Greg Morrisett for their comments on ideas in this paper. We would also like to thank anonymous reviewers for their constructive comments on earlier versions of this paper. This research is supported in part by US NSF grant CCF-0915157 and a research grant from Google.

References

David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 50–68, New York, 2004. ACM Press.

Patryk Czarnik and Aleksy Schubert. Extending operational semantics of the Java bytecode. In *Trustworth Global Computing 2007*, pages 57–72, 2007.

Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. In *Formal Syntax and Semantics of Java*, pages 41–82, London, UK, 1999. Springer-Verlag.

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, 1999. Springer-Verlag.

Stephen N. Freund and John Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, 2003.

Michael Furr and Jeffrey Foster. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)*, pages 309–324, 2006.

Michael Furr and Jeffrey Foster. Checking type safety of foreign function calls. *ACM Transactions on Programming Languages and Systems*, 30(4):1–63, 2008.

Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages (POPL)*, pages 248–260, 2001.

Martin Hirzel and Robert Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, 2007.

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.

Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 109–118, New York, NY, USA, 2008. ACM.

Byeongcheol Lee, Martin Hirzel, Robert Grimm, Ben Wiedermann, and Kathryn S. McKinley. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 36–49, 2010.

Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Autom. Reasoning*, 41(1):1–31, 2008.

Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *16th ACM Conference on Computer and Communications Security (CCS)*, pages 442–452, 2009.

Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201325772.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison Wesley, 1999. ISBN 0201432943.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 378–391, New York, NY, USA, 2005. ACM.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *34th ACM Symposium on Principles of Programming Languages (POPL)*, pages 3–10, 2007.

Gary McGraw and Edward W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

J. Strother Moore and George Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems*, 24(3):193–216, 2002.

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. submitted for conference publication. Technical report, Harvard University, November 2011.

George Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, 2002.

Gustavo Petri and Marieke Huisman. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *Bytecode 2008*, 2008.

David Pichardie. Bicolano—byte code language in Coq. `http://mobius.inria.fr/bicolano`, 2006.

Riccardo Pucella. Towards a formalization for COM, part I: the primitive calculus. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 331–342, New York, 2002. ACM Press.

Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the JVM. In *17th ACM Conference on Computer and Communications Security (CCS)*, pages 201–211, 2010.

Gang Tan. JNI Light: An operational model for the core JNI. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 114–130, 2010.

Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *17th Usenix Security Symposium*, pages 365–377, 2008.

Gang Tan and Greg Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 39–56, 2007.

Gang Tan, Andrew Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In *8th European Symposium on Programming (ESOP)*, pages 128–146, 1999.

Appendix A. JNIL operational semantics

$$\frac{P \vdash (S;H;R) \overset{\text{J}}{\longmapsto} (S';H';R')}{P \vdash (S;H;R) \longmapsto (S';H';R')} \qquad \frac{P \vdash (S;H;R) \overset{\text{N}}{\longmapsto} (S';H';R')}{P \vdash (S;H;R) \longmapsto (S';H';R')} \qquad \frac{(S;H) \overset{\text{GC}}{\longmapsto} (S';H')}{P \vdash (S;H;R) \longmapsto (S';H';R)}$$

$$\frac{L \subseteq \text{dom}(H|_{\text{J}}) \qquad L \cap \text{Roots}(S) = \emptyset \qquad L \cap \text{Reachable}((H|_{\text{J}}) \setminus L) = \emptyset}{(S;H) \overset{\text{GC}}{\longmapsto} (S;H \setminus L)}$$

$P \vdash (\langle md, pc, s, a\rangle_{\text{J}} \cdot S; H; R) \overset{\text{J}}{\longmapsto} (S';H';R)$, if

$P(md)@pc =$	and conditions hold,	then $S';H'=$
push v	$v = n$ or null	$\langle md, pc+1, v\cdot s, a\rangle_{\text{J}} \cdot S; H$
pop	$s = v \cdot s_1$	$\langle md, pc+1, s_1, a\rangle_{\text{J}} \cdot S; H$
localload d		$\langle md, pc+1, a(d)\cdot s, a\rangle_{\text{J}} \cdot S; H$
localstore d	$s = v \cdot s_1$	$\langle md, pc+1, s_1, a[d \mapsto v]\rangle_{\text{J}} \cdot S; H$
goto n		$\langle md, n, s, a\rangle_{\text{J}} \cdot S; H$
getfield fd	$fd = \langle \phi, \alpha, \tau\rangle \qquad s = \ell \cdot s_1$ $\text{ReadFd}(H, \ell, fd) = v$	$\langle md, pc+1, v\cdot s_1, a\rangle_{\text{J}} \cdot S; H$
putfield fd	$fd = \langle \phi, \alpha, \tau\rangle \qquad s = v \cdot \ell \cdot s_1$ $\text{UpdFd}(H, \ell, fd, v) = H_1$	$\langle md, pc+1, s_1, a\rangle_{\text{J}} \cdot S; H_1$
new ϕ	$\text{Fields}(P, \phi) = [fd_1, \ldots, fd_n]$ $s = v_n \cdot \ldots \cdot v_1 \cdot s_1$ $\text{AllocInst}(H, P, \phi) = (H_1, \ell)$ $\text{UpdFd}(H_1, \ell, [fd_1, \ldots, fd_n], [v_1, \ldots, v_n])$ $= H_2$	$\langle md, pc+1, \ell\cdot s_1, a\rangle_{\text{J}} \cdot S; H_2$
invokevirtual md_1	$md_1 = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ $s = v_n \cdot \ldots \cdot v_1 \cdot \ell \cdot s_1 \qquad \text{Tag}(H, \ell) = \phi'$ $md' = \langle \phi', \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$	$\text{NewFrame}(P, md', [\ell, v_1, \ldots, v_n])\cdot$ $\langle md, pc, s, a\rangle_{\text{J}} \cdot S; H$
returnval	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot \ell \cdot s', a'\rangle_{\text{J}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \qquad s = v_r \cdot s_1$	$\langle md', pc'+1, v_r \cdot s', a'\rangle_{\text{J}} \cdot S_1; H$
returnval	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot v \cdot s', v_x, L\rangle_{\text{N}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \qquad s = v_r \cdot s_1$	$\langle md', pc'+1, v_r \cdot s', v_x, L'\rangle_{\text{N}} \cdot S_1;$ $H, \qquad$ where $L' = L \cup \text{Roots}(v_r)$
throw	$s = \ell \cdot s_1$	$\langle \ell\rangle_{\text{X}} \cdot \langle md, pc, s, a\rangle_{\text{J}} \cdot S; H$

Fig. 17. JNIL operational semantics: GC and Java steps.

$P \vdash \langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R \xmapsto{\mathrm{N}} \langle md, pc', s, v_x, L\rangle_{\mathrm{N}} \cdot S; H'; R'$, if

$P(md)@pc =$	then $pc'; H'; R' =$
$\mathsf{Mov}\ r_d, op$	$pc + 1; H; R[r_d \mapsto \hat{R}(op)]$
$\mathsf{Jmp}\ op$	$\hat{R}(op); H; R$
$\mathsf{Ld}\ r_d, r_s[r_t]$	$pc + 1; H; R[r_d \mapsto H(\ell).\mathsf{blk}(n + n')]$, if $R(r_s) = \ell[n]$, and $R(r_t) = n'$
$\mathsf{St}\ r_d[r_t], r_s$	$pc + 1; H[\ell \mapsto \langle b', \omega\rangle]; R$ if $R(r_d) = \ell[n], R(r_t) = n', H(\ell) = \langle b, \omega\rangle$, and $b' = b[n + n' \mapsto R(r_s)]$
$\mathsf{Alloc}\ r_d, n$	$pc + 1; H \uplus H'; R[r_d \mapsto \ell]$, where $b = \{0 \mapsto \top, \ldots, n - 1 \mapsto \top\}$, and $H' = \{\ell \mapsto \langle b, \mathrm{N}\rangle\}$
$\mathsf{Free}\ r_s[n]$	$pc + 1; H \setminus \ell[n' + n]; R$, if $R(r_s) = \ell[n']$ and $n' + n \in \mathrm{dom}(H(\ell).\mathsf{blk})$

$P \vdash \langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R \xmapsto{\mathrm{N}} \langle md, pc + 1, s', v_x, L\rangle_{\mathrm{N}} \cdot S; H; R'$, if

$P(md)@pc =$	then $s'; R' =$
$\mathsf{SLd}\ r_d, \mathsf{sp}[n]$	$s; R[r_d \mapsto v_n]$, if $s = v_0 \cdot v_1 \cdot \ldots \cdot v_n \cdot s_1$
$\mathsf{SSt}\ \mathsf{sp}[n], r_s$	$v_1 \cdot \ldots \cdot R(r_s) \cdot s_1; R$, if $s = v_0 \cdot v_1 \cdot \ldots \cdot v_n \cdot s_1$
$\mathsf{SAlloc}\ n$	$\underbrace{\top \cdot \ldots \cdot \top}_{n} \cdot s; R$
$\mathsf{SFree}\ n$	$s_1; R$, if $s = v_0 \cdot \ldots \cdot v_{n-1} \cdot s_1$

where $\hat{R}(r) = R(r)$ and $\hat{R}(n) = n$

Fig. 18. JNIL operational semantics: native instructions (part 1).

$P \vdash (\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H; R) \overset{\mathrm{N}}{\longmapsto} (S'; H'; R)$, if

$P(md)@pc =$	and conditions hold,	then $S'; H' =$
$\mathsf{GetField}\ fd$	$fd = \langle \phi, \alpha, \tau\rangle \quad s = \ell \cdot s_1$ $\mathrm{ReadFd}(H, \ell, fd) = v \quad v_x = \mathsf{null}$	$\langle md, pc+1, v \cdot s_1, \mathsf{null}, L'\rangle_{\mathrm{N}} \cdot S; H$, where $L' = L \cup \mathrm{Roots}(v)$
$\mathsf{SetField}\ fd$	$fd = \langle \phi, \alpha, \tau\rangle \quad s = v \cdot \ell \cdot s_1$ $\mathrm{UpdFd}(H, \ell, fd, v) = H_1 \quad v_x = \mathsf{null}$	$\langle md, pc+1, s_1, \mathsf{null}, L\rangle_{\mathrm{N}} \cdot S; H_1$
$\mathsf{NewObject}\ \phi$	$\mathrm{Fields}(P, \phi) = [fd_1, \ldots, fd_n]$ $s = v_n \cdot \ldots v_1 \cdot s_1$ $\mathrm{AllocInst}(H, P, \phi) = (H_1, \ell)$ $\mathrm{UpdFd}(H_1, \ell, [fd_1, \ldots, fd_n],$ $[v_1, \ldots, v_n] = H_2$ $v_x = \mathsf{null}$	$\langle md, pc+1, \ell \cdot s_1, \mathsf{null}, L \cup \{\ell\}\rangle_{\mathrm{N}} \cdot$ $S; H_2$
$\mathsf{CallMethod}$ md_1	$md_1 = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ $s = v_n \cdot \ldots v_1 \cdot \ell \cdot s_1$ $\mathrm{Tag}(H, \ell) = \phi' \quad v_x = \mathsf{null}$ $md' = \langle \phi', \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$	$\mathrm{NewFrame}(P, md', [\ell, v_1, \ldots, v_n])\cdot$ $\langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S; H$
$\mathsf{IsInstanceOf}$ τ	$s = \ell \cdot s_1 \quad \mathrm{Tag}(H, \ell) = \phi'$ $v_x = \mathsf{null}$	$\langle md, pc+1, v \cdot s_1, \mathsf{null}, L\rangle_{\mathrm{N}} \cdot S; H$, where $v = 1$ if $P \vdash \mathsf{Cls}\ \phi' <: \tau$ or 0 otherwise.
$\mathsf{JNIThrow}$	$s = \ell \cdot s_1 \quad v_x = \mathsf{null}$	$\langle md, pc+1, s_1, \ell, L\rangle_{\mathrm{N}} \cdot S; H$
$\mathsf{ExnClear}$		$\langle md, pc+1, s, \mathsf{null}, L\rangle_{\mathrm{N}} \cdot S; H$
$\mathsf{ExnOccurred}$		$\langle md, pc+1, v \cdot s, v_x, L\rangle_{\mathrm{N}} \cdot S; H$ where $v = 0$ if $v_x = \mathsf{null}$ or 1 if $v_x = \ell$
Ret	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot v \cdot s', a'\rangle_{\mathrm{J}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \quad s = v_r \cdot s_1 \quad v_x = \mathsf{null}$	$\langle md', pc'+1, v_r \cdot s', a'\rangle_{\mathrm{J}} \cdot S_1; H$
Ret	$md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle$ $S = \langle md', pc', \overline{v_p} \cdot v \cdot s', v'_x, L\rangle_{\mathrm{N}} \cdot S_1$ $\lvert\overline{v_p}\rvert = n \quad s = v_r \cdot s_1 \quad v_x = \mathsf{null}$	$\langle md', pc'+1, v_r \cdot s', v'_x, L\rangle_{\mathrm{N}} \cdot S_1; H$, where $L' = L \cup \mathrm{Roots}(v_r)$
Ret	$v_x = \ell$	$\langle \ell\rangle_{\mathrm{X}} \cdot S; H$

Fig. 19. JNIL operational semantics: native instructions (part 2).

$P \vdash \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H; R \xrightarrow{\mathrm{J}} \langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S; H'; R$
if $o = \mathrm{Blank}(P, \texttt{throwable})$, $\ell \notin \mathrm{dom}(H)$, $H' = H \uplus \{\ell \mapsto (o, \mathrm{J})\}$, and one of the following holds:
— $P(md)@pc = \mathsf{getfield}\ fd$, and $s = \mathsf{null} \cdot s_1$;
— $P(md)@pc = \mathsf{putfield}\ fd$, and $s = v \cdot \mathsf{null} \cdot s_1$;
— $P(md)@pc = \mathsf{invokevirtual}\ \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$, and $s = v_n \cdot \ldots \cdot v_1 \cdot \mathsf{null} \cdot s_1$;
— $P(md)@pc = \mathsf{throw}$, and $s = \mathsf{null} \cdot s_1$.

$P \vdash \langle md, pc, s, \mathsf{null}, L\rangle_{\mathrm{N}} \cdot S; H; R \xrightarrow{\mathrm{N}} \langle md, pc+1, s, \ell, L\rangle_{\mathrm{N}} \cdot S; H'; R$
if $o = \mathrm{Blank}(P, \texttt{throwable})$, $\ell \notin \mathrm{dom}(H)$, $H' = H \uplus \{\ell \mapsto (o, \mathrm{J})\}$, and one of the following holds:
— $P(md)@pc = \mathsf{GetField}\ fd$, and $s = \mathsf{null} \cdot s_1$.
— $P(md)@pc = \mathsf{SetField}\ fd$, and $s = v \cdot \mathsf{null} \cdot s_1$.
— $P(md)@pc = \mathsf{CallMethod}\ \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \rightarrow \tau_r\rangle$, and $s = v_n \cdot \ldots \cdot v_1 \cdot \mathsf{null} \cdot s_1$.
— $P(md)@pc = \mathsf{IsInstanceOf}\ \tau$, and $s = \mathsf{null} \cdot s_1$.
— $P(md)@pc = \mathsf{JNIThrow}$, and $s = \mathsf{null} \cdot s_1$.

Fig. 20. JNIL operational semantics: raising exceptions.

$P \vdash S; H; R \xrightarrow{\mathrm{J}} S'; H'; R$, if

S=	and conditions hold,	then S', H'=
$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S_1$	$\mathrm{Tag}(H, \ell) = \phi \quad P(md).\mathsf{handlers} = \overline{\eta}$ $\mathit{CorrectHandler}(\overline{\eta}, P, pc, \phi) = \mathsf{None}$	$\langle \ell\rangle_{\mathrm{X}} \cdot S_1; H$
$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, a\rangle_{\mathrm{J}} \cdot S_1$	$\mathrm{Tag}(H, \ell) = \phi \quad P(md).\mathsf{handlers} = \overline{\eta}$ $\mathit{CorrectHandler}(\overline{\eta}, P, pc, \phi) = \lfloor n_t \rfloor$	$\langle md, n_t, \ell \cdot \epsilon, a\rangle_{\mathrm{J}} \cdot S_1; H$
$\langle \ell\rangle_{\mathrm{X}} \cdot \langle md, pc, s, v_x, L\rangle_{\mathrm{N}} \cdot S_1$		$\langle md, pc+1, s, \ell, L\rangle_{\mathrm{N}} \cdot S_1; H$

$$\mathit{CorrectHandler}(\epsilon, P, pc, \phi) = \mathsf{None}$$
$$\mathit{CorrectHandler}(\langle n_b, n_e, n_t, \phi'\rangle \cdot \overline{\eta}, P, pc, \phi) = \begin{cases} \lfloor n_t \rfloor & \text{if } n_b \le pc < n_e \text{ and } P \vdash \mathsf{Cls}\ \phi <: \mathsf{Cls}\ \phi' \\ \mathit{CorrectHandler}(\overline{\eta}, P, pc, \phi) & \text{otherwise} \end{cases}$$

Fig. 21. JNIL operational semantics: exception handling.

Appendix B. Bytecode type checking and safety

$$\boxed{\vdash P\ \mathsf{prog}}$$

$$\frac{\begin{array}{c}\texttt{object},\texttt{throwable}\in\mathrm{dom}(P)\qquad \forall\phi\in\mathrm{dom}(P).\ P\vdash\phi\ \mathsf{class}\\ \forall md\in\mathrm{JavaMD}(P)\cup\mathrm{NativeMD}(P).\ P\vdash md\ \mathsf{mid}\\ \forall md\in\mathrm{JavaMD}(P).\ P\vdash md\ \mathsf{jmethod}\qquad \forall md\in\mathrm{NativeMD}(P).\ P\vdash md\ \mathsf{nmethod}\end{array}}{\vdash P\ \mathsf{prog}}$$

$$\boxed{P\vdash\phi\ \mathsf{class}}\qquad \frac{P(\texttt{object})=\langle\mathsf{None},\emptyset\rangle}{P\vdash\texttt{object}\ \mathsf{class}}\qquad \frac{P(\texttt{throwable})=\langle\lfloor\texttt{object}\rfloor,\emptyset\rangle}{P\vdash\texttt{throwable}\ \mathsf{class}}$$

$$\frac{\begin{array}{cl}P(\phi)=\langle\lfloor\phi'\rfloor,[\langle\phi,\alpha_1,\tau_1\rangle,\ldots,\langle\phi,\alpha_n,\tau_n\rangle]\rangle & \\ \phi'\in\mathrm{dom}(P)\qquad\neg(P\vdash\mathsf{Cls}\ \phi'<:\mathsf{Cls}\ \phi) & \textit{no cycles}\\ \forall j\in[1..n].\ P\vdash\langle\phi,\alpha_j,\tau_j\rangle\ \mathsf{fid} & \\ \forall\alpha,\overline{\tau},\tau_r.\ \langle\phi',\alpha,\overline{\tau}\to\tau_r\rangle\in\mathrm{dom}(P)\ \Rightarrow\ \langle\phi,\alpha,\overline{\tau}\to\tau_r\rangle\in\mathrm{dom}(P) & \textit{inherit/override all methods}\end{array}}{P\vdash\phi\ \mathsf{class}}$$

$$\boxed{P\vdash md\ \mathsf{jmethod}}$$

$$\frac{\begin{array}{c}md=\langle\phi,\alpha,[\tau_1,\ldots,\tau_n]\to\tau_{n+1}\rangle\qquad P(md)=\langle\overline{I},\overline{\eta},T_s,T_a\rangle\qquad|\overline{I}|\geq 1\\ T_s(1)=\epsilon\qquad P\vdash\{0\mapsto\mathsf{Cls}\ \phi,1\mapsto\tau_1,\ldots,n\mapsto\tau_n,n+1\mapsto\mathsf{Top},\ldots\}<:T_a(1)\\ \forall i\in\mathrm{dom}(\overline{I}).\ P,md,T_s,T_a\vdash\overline{I}@i\qquad\forall\eta\in\overline{\eta}.\ P,T_s,T_a\vdash\eta\ \mathsf{handles}\ \overline{I}\end{array}}{P\vdash md\ \mathsf{jmethod}}$$

$$\boxed{P\vdash md\ \mathsf{nmethod}}\qquad\frac{}{P\vdash md\ \mathsf{nmethod}}$$

$$\boxed{P\vdash\tau_1<:\tau_2}$$

$$\frac{}{P\vdash\tau<:\mathsf{Top}}\qquad\frac{}{P\vdash\mathsf{Int}<:\mathsf{Int}}\qquad\frac{}{P\vdash\mathsf{Cls}\ \phi<:\mathsf{Cls}\ \phi}\qquad\frac{\begin{array}{c}P\vdash\mathsf{Cls}\ \phi_1<:\mathsf{Cls}\ \phi_2\\ P(\phi_2).\mathsf{super}=\lfloor\phi_3\rfloor\end{array}}{P\vdash\mathsf{Cls}\ \phi_1<:\mathsf{Cls}\ \phi_3}$$

$$\boxed{P,H\vdash v:\tau}\qquad\frac{P,H\vdash v:\tau\quad P\vdash\tau<:\tau'}{P,H\vdash v:\tau'}\qquad\frac{}{P,H\vdash v:\mathsf{Top}}$$

$$\frac{}{P,H\vdash n:\mathsf{Int}}\qquad\frac{\mathrm{Tag}(H,\ell)=\phi}{P,H\vdash\ell:\mathsf{Cls}\ \phi}\qquad\frac{}{P,H\vdash\mathsf{null}:\mathsf{Cls}\ \phi}$$

$$\boxed{P\vdash fd\ \mathsf{fid}}\qquad\frac{\phi\in\mathrm{dom}(P)\quad P\vdash\tau\ \mathsf{ty}}{P\vdash\langle\phi,\alpha,\tau\rangle\ \mathsf{fid}}$$

$$\boxed{P\vdash md\ \mathsf{mid}}\qquad\frac{\phi\in\mathrm{dom}(P)\quad\phi\neq\texttt{object}\quad\forall i\in[1..n+1].\ P\vdash\tau_i\ \mathsf{ty}}{P\vdash\langle\phi,\alpha,[\tau_1,\ldots,\tau_n]\to\tau_{n+1}\rangle\ \mathsf{mid}}$$

$$\boxed{P,T_s,T_a\vdash\eta\ \mathsf{handles}\ \overline{I}}$$

$$\frac{\begin{array}{c}P\vdash\mathsf{Cls}\ \phi<:\mathsf{Throwable}\qquad 1\leq n_b<n_e\qquad n_b,n_e-1,n_t\in\mathrm{dom}(\overline{I})\\ P\vdash[\mathsf{Cls}\ \phi]<:T_s(n_t)\qquad\forall i\in[n_b,n_e-1].\ P\vdash T_a(i)<:T_a(n_t)\end{array}}{P,T_s,T_a\vdash\langle n_b,n_e,n_t,\phi\rangle\ \mathsf{handles}\ \overline{I}}$$

$$\boxed{P\vdash\tau\ \mathsf{ty}}\qquad\frac{}{P\vdash\mathsf{Int}\ \mathsf{ty}}\qquad\frac{\phi\in\mathrm{dom}(P)}{P\vdash(\mathsf{Cls}\ \phi)\ \mathsf{ty}}$$

Fig. 22. Bytecode verification, part 1.

$\boxed{P, md, T_s, T_a \vdash \overline{I}@i}$

if $\overline{I}[i] =$	Conditions on T_s	Conditions on T_a	Other conditions
push n	$P \vdash \mathsf{Int} \cdot T_s(i) <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
push null	$P \vdash \tau \cdot T_s(i) <: T_s(i+1)$ $\mathrm{IsRefType}(\tau)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
pop	$T_s(i) = \tau_1 \cdot \overline{\tau}$ $P \vdash \overline{\tau} <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
localload d	$P \vdash T_a(i)(d) \cdot T_s(i) <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
localstore d	$T_s(i) = \tau_1 \cdot \overline{\tau}$ $P \vdash \overline{\tau} <: T_s(i+1)$	$P \vdash T_a(i)[d \mapsto \tau_1] <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
goto n	$P \vdash T_s(i) <: T_s(n)$	$P \vdash T_a(i) <: T_a(n)$	$n \in \mathrm{dom}(\overline{I})$
getfield $\langle \phi, \alpha, \tau_1 \rangle$	$P \vdash T_s(i) <: \mathsf{Cls}\ \phi \cdot \overline{\tau}$ $P \vdash \tau_1 \cdot \overline{\tau} <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$ $\langle \phi, \alpha, \tau_1 \rangle \in \mathrm{Fields}(P, \phi)$
putfield $\langle \phi, \alpha, \tau_1 \rangle$	$P \vdash T_s(i) <: \tau_1 \cdot \mathsf{Cls}\ \phi \cdot T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$ $\langle \phi, \alpha, \tau_1 \rangle \in \mathrm{Fields}(P, \phi)$
new ϕ	$\mathrm{Fields}(P, \phi) =$ $[\langle \phi_1, \alpha_1, \tau_1 \rangle, \ldots, \langle \phi_n, \alpha_n, \tau_n \rangle]$ $P \vdash T_s(i) <: \tau_1 \cdot \ldots \cdot \tau_n \cdot \overline{\tau}$ $P \vdash \mathsf{Cls}\ \phi \cdot \overline{\tau} <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
invokevirtual md_1	$md_1 = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r \rangle$ $P \vdash T_s(i) <: \tau_n \cdot \ldots \cdot \tau_1 \cdot \mathsf{Cls}\ \phi \cdot \overline{\tau}$ $P \vdash \tau_r \cdot \overline{\tau} <: T_s(i+1)$	$P \vdash T_a(i) <: T_a(i+1)$	$i+1 \in \mathrm{dom}(\overline{I})$
returnval	$md = \langle \phi, \alpha, \overline{\tau_1} \to \tau_r \rangle$ $P \vdash T_s(i) <: \tau_r \cdot \overline{\tau}$		
throw	$P \vdash T_s(i) <: \mathsf{Throwable} \cdot \overline{\tau}$		

Fig. 23. Bytecode verification, part 2: checking instructions.

$$\boxed{P \vdash (S; H; R)\ \mathsf{state}}$$

$$\frac{P \vdash (H|_J)\ \mathsf{jheap} \qquad P, H|_J \vdash S\ \mathsf{stack} \qquad P, H|_J, S \vdash (R; H|_N)\ \mathsf{nstate}}{P \vdash (S; H; R)\ \mathsf{state}}$$

$$\boxed{P \vdash H\ \mathsf{jheap}} \quad \frac{\forall \ell \in \mathrm{dom}(H).\ \exists o.\ H(\ell) = \langle \mathrm{Rep}(o), \mathrm{J}\rangle \ \wedge\ P, H \vdash o : \mathrm{Tag}(H, \ell)}{P \vdash H\ \mathsf{jheap}}$$

$$\boxed{P, H \vdash o : g} \quad \frac{\begin{array}{c}\mathrm{Fields}(P, \phi) = [fd_1, \ldots, fd_n] \\ \forall i \in [1..n].\ fd_i = \langle -, -, \tau_i\rangle \ \Rightarrow\ P, H \vdash v_i : \tau_i\end{array}}{P, H \vdash \langle\!\langle fd_1 = v_1, \ldots, fd_n = v_n\rangle\!\rangle_\phi : \phi}$$

$$\boxed{P, H_J, S \vdash (R; H_N)\ \mathsf{nstate}} \quad \frac{}{P, H_J, S \vdash (R; H_N)\ \mathsf{nstate}}$$

$$\boxed{P, H \vdash S\ \mathsf{stack}}$$

$$\frac{\begin{array}{c}P, H \vdash \ell : \mathsf{Throwable} \\ P \vdash (\langle \ell\rangle_{\mathrm{X}} \cdot S)\ \mathsf{callchain} \qquad P, H \vdash S\ \mathsf{stack}\end{array}}{P, H \vdash (\langle \ell\rangle_{\mathrm{X}} \cdot S)\ \mathsf{stack}} \qquad \frac{\begin{array}{c}P, H \vdash F\ \mathsf{frame} \qquad P, H \vdash S\ \mathsf{stack} \\ P \vdash (F \cdot S)\ \mathsf{callchain}\end{array}}{P, H \vdash F \cdot S\ \mathsf{stack}}$$

$$\boxed{P, H \vdash F\ \mathsf{frame}}$$

$$\frac{\begin{array}{c}md \in \mathrm{JavaMD}(P) \qquad P(md) = \langle \overline{I}, \overline{\eta}, T_s, T_a\rangle \\ pc \in \mathrm{dom}(\overline{I}) \qquad P, H \vdash s : T_s(pc) \qquad P, H \vdash a : T_a(pc)\end{array}}{P, H \vdash \langle md, pc, s, a\rangle_{\mathrm{J}}\ \mathsf{frame}} \qquad \frac{md \in \mathrm{NativeMD}(P)}{P, H \vdash \langle md, pc, s, v_x, L\rangle_{\mathrm{N}}\ \mathsf{frame}}$$

$$\boxed{P \vdash S\ \mathsf{callchain}}$$

$$\frac{F = \langle \ldots\rangle_{\mathrm{J}} \text{ or } \langle \ldots\rangle_{\mathrm{N}}}{P \vdash (F \cdot \epsilon)\ \mathsf{callchain}} \qquad \frac{\mathrm{TopFrame}(S) = \langle \ldots\rangle_{\mathrm{J}}}{P \vdash (\langle \ell\rangle_{\mathrm{X}} \cdot S)\ \mathsf{callchain}} \qquad \frac{\begin{array}{c}\mathrm{TopFrame}(S) = \langle md', pc', -, -, -\rangle_{\mathrm{N}} \\ P(md')@pc' = \mathsf{CallMethod}\ \langle \phi_1, \alpha, \overline{\tau} \to \tau_r\rangle\end{array}}{P \vdash (\langle \ell\rangle_{\mathrm{X}} \cdot S)\ \mathsf{callchain}}$$

$$\frac{\begin{array}{c}F = \langle md, pc, -, -\rangle_{\mathrm{J}} \ \vee\ \langle md, pc, -, -, -\rangle_{\mathrm{N}} \qquad md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle \\ P(md')@pc' = \mathsf{invokevirtual}\ \langle \phi_1, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle \qquad P \vdash \mathsf{Cls}\ \phi <: \mathsf{Cls}\ \phi_1\end{array}}{P \vdash (F \cdot \langle md', pc', s', a'\rangle_{\mathrm{J}} \cdot S)\ \mathsf{callchain}}$$

$$\frac{\begin{array}{c}F = \langle md, pc, -, -\rangle_{\mathrm{J}} \ \vee\ \langle md, pc, -, -, -\rangle_{\mathrm{N}} \qquad md = \langle \phi, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle \\ |s'| \geq n + 1 \qquad P(md')@pc' = \mathsf{CallMethod}\ \langle \phi_1, \alpha, [\tau_1, \ldots, \tau_n] \to \tau_r\rangle \qquad P \vdash \mathsf{Cls}\ \phi <: \mathsf{Cls}\ \phi_1\end{array}}{P \vdash (F \cdot \langle md', pc', s', v_x', L'\rangle_{\mathrm{N}} \cdot S)\ \mathsf{callchain}}$$

Fig. 24. Well-typed Java states.

Appendix C. Lemmas used in proofs of safety theorems

Lemma 1 (Canonical forms).

(1) If $P, H \vdash v : \mathsf{Int}$, then $v = n$.
(2) If $P, H \vdash v : \mathsf{Cls}\ \phi$, then one of the following cases is true:

(i) $v = \mathsf{null}$;

(ii) $\exists \ell, \phi'$. so that $v = \ell$, $\mathrm{Tag}(H, \ell) = \phi'$, and $P \vdash \mathsf{Cls}\ \phi' <: \mathsf{Cls}\ \phi$.

Lemma 2 (Laws of subtyping).

(1) $P \vdash \tau <: \tau$.
(2) If $P \vdash \tau_1 <: \tau_2$, and $P \vdash \tau_2 <: \tau_3$, then $P \vdash \tau_1 <: \tau_3$.

The first is proved by induction over τ, and the second by induction over $P \vdash \tau_2 <: \tau_3$.

Lemma 3. If $P, H \vdash v : \tau$, then $P, H|_{\mathrm{J}} \vdash v : \tau$.

A necessary notion when proving Java preservation is a definition of Java heap extensions.

Definition 4 (Java heap extensions).

$$H \leq H' \triangleq \forall \ell \in \mathrm{dom}(H).\ \ell \in \mathrm{dom}(H') \ \wedge\ \mathrm{Tag}(H, \ell) = \mathrm{Tag}(H', \ell)$$

Note that the above definition concerns only runtime tags; H can be extended to H' even if there is some mutation to heap objects.

The following lemmas show that typings are not affected when extending heaps:

Lemma 4 (Monotonicity). Assume $H \leq H'$.

(1) If $P, H \vdash v : \tau$, then $P, H' \vdash v : \tau$.
(2) If $P, H \vdash o : g$, then $P, H' \vdash o : g$.
(3) If $P, H \vdash F\ \mathsf{frame}$, then $P, H' \vdash F\ \mathsf{frame}$.
(4) If $P, H \vdash S\ \mathsf{stack}$, then $P, H' \vdash S\ \mathsf{stack}$.

The proof of the GC-safety theorem uses the following lemma.

Lemma 5. Assume $H' = H \setminus L$.

(1) If $P, H \vdash v : \tau$, and $L \cap \mathrm{Roots}(v) = \emptyset$, then $P, H' \vdash v : \tau$.
(2) If $P, H \vdash F\ \mathsf{frame}$, and $L \cap \mathrm{Roots}(F) = \emptyset$, then $P, H' \vdash F\ \mathsf{frame}$.
(3) If $P, H \vdash S\ \mathsf{stack}$, and $L \cap \mathrm{Roots}(S) = \emptyset$, then $P, H' \vdash S\ \mathsf{stack}$.