Type Safe Dynamic Linking for JVM Access Control

Christian Skalka
University of Vermont
skalka@cs.uvm.edu

Abstract

The Java JDK security model provides an access control mechanism for the JVM based on dynamic stack inspection. Previous results have shown how stack inspection can be enforced at compile time via whole-program type analysis, but features of the JVM present significant remaining technical challenges. For instance, dynamic dispatch at the bytecode level requires special consideration to ensure flexibility in typing. Even more problematic is dynamic class loading and linking, which disallow a purely static analysis in principle, though the intended applications of the JDK exploit these features. We propose an extension to existing bytecode verification, that enforces stack inspection at link time, without imposing new restrictions on the JVM class loading and linking mechanism. Our solution is more flexible than existing type based approaches, and establishes a formal type safety result for bytecode-level access control in the presence of dynamic class linking.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Constructs and Features

General Terms Security, Languages, Theory, Verification.

Keywords Static type analysis, bytecode verification, dynamic linking, language-based security.

1. Introduction

A primary purpose of the JDK security model is to protect against non-local extensions to local code in the Java Virtual Machine (JVM) [9]. *Sandboxing* of dynamically loaded plug-ins and applets provides security in web environments where non-local code authors may not be fully trusted. Sandboxing relies on a combination of class loading discipline, bytecode verification, and dynamic stack inspection to enforce security. An essential point is that the intended applications of the JDK are in program settings where dynamic class loading and linking are used, and possibly abused.

Stack inspection is a language-based security mechanism used in the JVM, which ensures that executing programs can access a protected resource only if they're trusted to do so. The stack inspection algorithm examines the dynamic caller history of the program, as recorded on the call stack, to ensure that particular resources are not accessed by unprivileged code either directly or through man-in-the-middle attacks [9, 24]. A variety of previous results have demonstrated that stack inspection can be enforced statically in a variety of language models (e.g. core ML), using both type theories [20] and control flow graph abstractions [16]. It is argued that static verification of stack inspection promotes program efficiency and understanding, and provides a more eager approach to security. However, despite this body of work, only one group has studied static enforcement of stack inspection for Java bytecodes [13, 14], and none have thoroughly considered how to integrate their analyses with dynamic linking and loading in the JVM. But to be practical, it is obviously necessary for any analysis to be applicable to the language model where stack inspection is employed, so a consideration of the problem of dynamic linking and loading is essential.

In this paper we develop a type analysis for a Java bytecode language, and show how it can be integrated with class loading and linking in the JVM to obtain link time enforcement of stack inspection. An obvious problem is that purely static analysis is literally impossible in this setting, since not all code involved in the program is known at compile-time. However, bytecode verification as it currently exists in the JVM [8] illustrates a compromise solution: even though dynamically loaded code is not known statically, it can be type checked at link time, yielding a form of *link time type safety*. Note that this is different than e.g. soft typing [4] or dynamic typing [1], since link time type safety guarantees that dynamic checks will succeed, whereas those approaches are based on an interaction of type checks and dynamic checks. Link time typing is essentially static typing performed incrementally as new code is available (linked).

Requirements for a solution The addition of dynamic loading and linking to a bytecode level model raises some subtle issues in the implementation. For example, we can imagine a scheme whereby the *entire* program codebase is type checked again upon every new linkage, but while this could obtain type safety it is not a good solution to the problem. Thus, we enumerate four requirements for a solution as follows, with the understanding that this foundational study will focus on a simplified core language model. (1) The analysis must provably eliminate the need for run time checks, or any run time overhead associated with stack inspection. This requirement is in place to preserve a known practical advantage of type analysis of stack inspection, namely that it allows elimination of run time checks and enables compiler optimizations. (2) The analysis must not impose on the flexibility of dynamic linking and loading as it currently exists in the JVM, in the sense that it should not force earlier loading or linking of code. Since dynamic linking and loading provides significant flexibility in JVM execution for web applications, this requirement ensures that flexibility is preserved. (3) The analysis must be modular, in that analysis of dynamically loaded and linked code must not require re-analysis of previously linked code. (4) The analysis must be sufficiently flexible to provide a foundation for realistic program analysis. This last point will be clarified when we examine previous related work that imposes perhaps unrealistic restrictions on code.

PPDP'07 July 14–16, 2007, Wroclaw, Poland.
Copyright © 2007 ACM 978-1-59593-769-8/07/0007…$5.00

$$CT \vdash \texttt{C} <: \texttt{C} \qquad \frac{CT \vdash \texttt{B} <: \texttt{C} \qquad CT \vdash \texttt{C} <: \texttt{D}}{CT \vdash \texttt{B} <: \texttt{D}}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\}^{\pi}}{CT \vdash \texttt{C} <: \texttt{D}}$$

Figure 1. Nominal subtyping for FJ

Our contribution We claim that our system is a solution in the previously defined sense, because: (1) We develop metatheory for a link time type safety result demonstrating that run time stack inspection checks can be eliminated in the presence of dynamic loading and linking (Theorem 4.2). (2) Our type analysis works as an extension to bytecode verification, within the existing mechanism for loading and linking. In particular, the analysis does not require any classes to be loaded or linked earlier than they would be in the existing JVM. (3) A polymorphic constraint type system, combined with a novel technique called *reification*, provides a modular solution, that does not require re-computation of types when new classes are linked. (4) Through the use of parametric polymorphism, our analysis is more complete than existing type systems for JVM access control such as [14].

1.1 Linking and Type Reification

We define the semantics of programs as a single-step operational relation on *configurations* $\mathcal{K}$. Leaving aside details for the moment, any $\mathcal{K}$ is a model of machine and program state. Execution in the model depends on a class table CT, which is a mapping from class names $\texttt{C}$ to class definitions $\texttt{L}$. In a setting with dynamic loading and linking of classes, the given class table grows monotonically during execution. We write $CT[\texttt{C} : \texttt{L}]$ to denote the extension of CT with a mapping from $\texttt{C}$ to $\texttt{L}$, and $CT \subseteq CT'$ means that every mapping in CT is also in CT'.

In the JVM, loading and linking are separate processes, hence we are careful not to conflate the terms. *Linking* of a class $\texttt{C}$ in the JVM implies bytecode verification of $\texttt{C}$, and general preparation of $\texttt{C}$ for execution [14], e.g. resolution of references. Note that bytecode verification of $\texttt{C}$ depends on checking subtyping relations, for example if a method is invoked on some class $\texttt{D}$, then $\texttt{D}$ must be a subclass of the declared method parameter type. Hence, linking of a class $\texttt{C}$ requires that all classes $\texttt{D}$ mentioned in $\texttt{C}$ be *loaded* so that their code is on hand and subtype relations can be checked. But class subtyping in Java is *nominal* (Fig. 1), not structural, so that checking subtype relations involving $\texttt{D}$ does not require $\texttt{D}$ to itself be typechecked. Once the code is on hand, the associated subtyping relation is easily obtained from code annotations.

Our formalization of program execution thus mentions loaded class tables CT_{ld} and linked class tables CT_{lk} as well as configurations $\mathcal{K}$, so that one step reduction is written:

$$CT_{ld}, CT_{lk}, \mathcal{K} \to CT'_{ld}, CT'_{lk}, \mathcal{K}'$$

Linking steps are predicated on verification of the linked class, and verification is predicated on the inheritance relation induced by the loaded class table. Writing "$CT \vdash_{verify} \texttt{C}$" to denote that $\texttt{C}$ is verifiable given the inheritance relation declared in class table CT, the following rule approximates our linkage model:

$$\frac{\texttt{C} \in dom(CT_{ld}) \qquad CT_{ld} \vdash_{verify} \texttt{C}}{CT_{ld}, CT_{lk}, \mathcal{K} \to CT_{ld}, CT_{lk}[\texttt{C} : \texttt{L}], \mathcal{K}}$$

Link time verification enforces safety during run time– that is, given a series of non-linking computation steps, since any execution is based on verified linked code, any resulting machine state is guaranteed not to be ill-formed. In the remainder of the paper, we will fill in the details of verification with an analysis guaranteeing that run time stack inspections and associated machinery can be eliminated in a simplified bytecode language model.

Our analysis is a type and effect system, where effects approximate the security effects of programs (stack inspection checks and privilege activations [9]). In particular, method types are of the form $\texttt{T}_1..\texttt{T}_n \xrightarrow{\texttt{R}} \texttt{T}$, where $\texttt{T}_1..\texttt{T}_n$ and $\texttt{T}$ are the domain and range types and $\texttt{R}$ is the effect of the method. Object types are of the form $[\texttt{T}\ \texttt{C}]$, where $\texttt{C}$ is the object class name and $\texttt{T}$ describes the fields and methods of the object; thus our type system combines nominal and structural features. A constraint system D is a component of any given type, and this, combined with effect polymorphism, allows the necessary flexibility for our analysis in the presence of linking.

For example, consider the following class definition `Foo`, containing a single method that invokes the `run` method of a `Runnable` object, the latter specified as containing a single method `run` that takes and returns an `Object`:

```
class Foo extends Object {
   Object m(Object x, Runnable y){ return y.run(x); }
}
```

Clearly, the statically assigned effect of the method `m` will be the effect of the `run` method of the given `Runnable` object. But any version of the `run` method may be dynamically dispatched at this point, and more than one version may exist. Previous authors [14] have proposed an approach whereby the effect of `m` is approximated as the join of the effects of all known versions of `run`. A scheme for verifying linked classes is also outlined by those authors, whereby the effects of new versions of `run` must be contained in that join. However, this approach has two significant drawbacks. Firstly, in large codebases, where there may exist many `Runnable` classes, the join of effects will be very large and nonspecific. Secondly, linked classes should be able to establish new security contexts– the point of the JDK after all is to allow interaction with unknown code. But in our system, polymorphic effects stand in for unknown effects. Assuming the existence of an `Object` class with no fields or methods, and letting $\texttt{Unit} \triangleq [\ \texttt{Object}]$ be its type in our system, we can assign the following type to `Foo`'s method `m`:

$$(\texttt{Unit}, [\texttt{run} : \texttt{Unit} \xrightarrow{\{\texttt{v}\}} \texttt{Unit Runnable}]) \xrightarrow{\{\texttt{v}\}} \texttt{Unit}$$

Here, `v` is a polymorphic type variable that may be lower bounded by different type constraints at different application points of `m`. This allows us to achieve greater type precision and scalability to large codebases, as we will illustrate in Sect. 3.3 by returning to this example.

More significantly, polymorphic constraints serve as the basis for *reification*, a technique we develop to obtain precise typings for dynamically linked code. In essence, we delay the lower bounding of polymorphic effects until relevant code is linked, by maintaining a closure that is filled in and elaborated by link time verification. Reification has the same computational complexity and precision of purely static type analysis, the latter established in Lemma 4.6. We obtain an appropriately phrased type safety result in Theorem 3.1, based on a subject reduction argument, that also obtains type safety for link time verification in Theorem 4.2. We will revisit the above example in Sect. 2.2, Sect. 3.3, and Sect. 4.5 to illustrate our language, type system, and reification.

2. The Language Model

We begin our development with an overview of our language model. Ohori et al. have already defined a simplified bytecode model with OO features including inheritance and dynamic dispatch [14]. This serves as an appealing core model for development of our type analysis, and using it allows a direct comparison

$\mathtt{r} \in \mathcal{R}, \pi \subseteq \mathcal{R}, R \subseteq \mathcal{R}$			*privileges*
$\mathtt{J}$	$::=$	$\mathtt{C} \mid \mathtt{C}..\mathtt{C} \rightarrow \mathtt{C}$	*JVM types*
$\mathtt{I}$	$::=$	$\mathtt{enable(r)} \mid \mathtt{check(r)} \mid \mathtt{new\,C} \mid \mathtt{invoke(C,m)}$	*instructions*
$\mathtt{B}$	$::=$	$\mathtt{return} \mid \mathtt{goto}(\ell) \mid \mathtt{I} \cdot \mathtt{B}$	*blocks*
$\mathtt{v}$	$::=$	$h \mid \mathtt{new\,C}$	*values*
M	$::=$	$\{\ell_1 = \mathtt{B}_1, \ldots, \ell_n = \mathtt{B}_n\}$	*methods*
$\mathtt{L}$	$::=$	$\mathtt{class\,C\,extends\,D}\,\{\mathtt{m}_1 : \mathtt{J}_1 = M_1, \ldots, \mathtt{m}_n : \mathtt{J}_n = M_n\}^{\pi}$	*class definitions*
ς	$::=$	$\mathbf{nil} \mid \mathtt{v} :: \varsigma$	*stacks*
D	$::=$	$\mathbf{nil} \mid (R, \varsigma, M^{\pi}\{\mathtt{B}\}) :: D$	*dumps*
H	$::=$	$\{h_1 \mapsto \mathtt{new\,C}_1, \ldots, h_n \mapsto \mathtt{new\,C}_n\}$	*heaps*
$\mathcal{K}$	$::=$	$\langle R, \varsigma, M^{\pi}\{\mathtt{B}\}, D, H\rangle$	*configurations*

Figure 2. JVM$_{\text{sec}}$ language syntax

$$\frac{CT(\mathtt{C}) = \mathtt{class\,C\,extends\,D}\,\{\ldots, \mathtt{m} : \mathtt{J} = M, \ldots\}^{\pi}}{mtype_{CT}(\mathtt{m}, \mathtt{C}) = \mathtt{J}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\,C\,extends\,D}\,\{\mathtt{m}_1 : \ldots, \ldots, \mathtt{m}_n : \ldots\}^{\pi} \quad \mathtt{m} \notin \mathtt{m}_1..\mathtt{m}_n}{mtype_{CT}(\mathtt{m}, \mathtt{C}) = mtype_{CT}(\mathtt{m}, \mathtt{D})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\,C\,extends\,D}\,\{\ldots, \mathtt{m} : \mathtt{J} = M, \ldots\}^{\pi}}{mbody_{CT}(\mathtt{m}, \mathtt{C}) = M^{\pi}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\,C\,extends\,D}\,\{\mathtt{m}_1 : \ldots, \ldots, \mathtt{m}_n : \ldots\}^{\pi} \quad \mathtt{m} \notin \mathtt{m}_1..\mathtt{m}_n}{mbody_{CT}(\mathtt{m}, \mathtt{C}) = mbody_{CT}(\mathtt{m}, \mathtt{D})}$$

Figure 3. Auxiliary functions

with their results. Therefore our language model is an extension of Ohori et al.'s with dynamic loading and linking in the JVM style. There are some important nuances to these mechanisms, so we first provide intuitions as to how they work, prior to our formalization.

2.1 Configurations and Bytecode Instructions

We now define the syntax and run time semantics of a simple bytecode language, called JVM$_{\text{sec}}$, assuming given a single linked, unchanging class table. The treatment is cursory since the specification is adapted from Higuchi et al.'s previous work [14], though for simplicity, we leave out native code, and numeric values and operations, as in their model. Also, rather than specifying authorization requirements for methods, we adapt a stack inspection check that is closer in spirit to existing Java. The JVM$_{\text{sec}}$ bytecode language is functional, and omits object fields and mutual recursion between classes to focus our technical treatment (recursion within a class definition is allowed).

The syntax of JVM$_{\text{sec}}$ is given in Fig. 2. We add to a standard bytecode language resources $\mathtt{r}$ taken from a countably infinite set $\mathcal{R}$. Sets of privileges are denoted R, while *principals* π are also defined as sets of privileges, following [20]. Classes contain a supertype declaration, and a collection of labeled method bodies along with their type annotations $\mathtt{J}$, that we assume are function types. In any function type $\mathtt{C}_1..\mathtt{C}_n \rightarrow \mathtt{C}$, the notation $\mathtt{C}_1..\mathtt{C}_n$ represents a possibly empty sequence of JVM types, and we write $\mathtt{C} \in \mathtt{C}_1..\mathtt{C}_n$ iff $\mathtt{C}$ is an element in the sequence. The empty sequence is denoted $\varnothing$. This sequence notation will be reused for other syntactic forms.

Method type and supertype annotations are declared by the programmer, and form an important part of the backbone of type analysis. The $mtype$ function defined in Fig. 3 retrieves the declared types of methods. As in Java, we require that overridden methods do not differ in their type annotations from supertype versions. Formally:

DEFINITION 2.1. *A class table CT is* annotation consistent *iff for all $\mathtt{C} \in dom(CT)$, $CT(\mathtt{C}) = \mathtt{class\,C\,extends\,B}\,\{\ldots\}^{\pi}$, and $mtype_{CT}(\mathtt{m}, \mathtt{C}) = \mathtt{J}$ and $mtype_{CT}(\mathtt{m}, \mathtt{B}) = \mathtt{J}'$ together imply that $\mathtt{J} = \mathtt{J}'$.*

Hereafter we assume that all class tables are annotation consistent.

Method bodies are sets of labeled blocks. All labels ℓ are unique within a particular block. We assume that every method $\mathtt{m}$ contains one block labeled $entry$. Blocks $\mathtt{B}$ are sequences of instructions, which have access to an operand stack ς of values $\mathtt{v}$ which are object references h. The instruction $\mathtt{goto}(\ell)$ is the standard goto instruction, and $\mathtt{return}$ terminates a method. The $\mathtt{new\,C}$ instruction creates a new object of class $\mathtt{C}$. The instructions $\mathtt{enable(r)}$ and $\mathtt{check(r)}$ enable and check for a privilege $\mathtt{r}$ respectively.

Arguments for and return values of instructions reside on the stack ς. When we say that instructions $\mathtt{B}$ "return" a value $\mathtt{v}$, this literally means that $\mathtt{v}$ is on the top of the stack when $\mathtt{B}$ terminates. When a method with n parameters is invoked, it is provided a stack of depth $n+1$; the additional element is a reference to the invoking object at the bottom of the stack, so that $\mathtt{self}$ is always accessible via $\mathtt{acc(0)}$. The result of evaluating a program is the value at the top of the stack when it terminates.

Configurations $\mathcal{K}$ are then constructed from these elements. Each configuration includes a set R of *enabled privileges*, in the security passing style. Each configuration also contains an operand stack ς. The *redex* component $M^{\pi}\{\mathtt{B}\}$ of a configuration represents the block $\mathtt{B}$ being executed, the method M the block was defined in, and the method's defining principal π– that is, the principal that owns the class in which the method is defined. Configurations also contain *heaps* H, which are mappings from locations h to objects. *Dumps* are sequences of continuations; when a method is invoked, the context of invocation is stored on the dump for subsequent execution when the method returns. Selected operational semantics rules for configurations are defined in Fig. 4. We write $\rightarrow_{CT}^{\star}$ to denote the reflexive, transitive closure of $\rightarrow_{CT}$. We assume that every program execution begins in an initial configuration with no privileges enabled, as follows.

DEFINITION 2.2. *We require that any class table CT contains at least a definition for a class* $\mathtt{Main}$ *owned by $\varnothing$ containing only one method* $\mathtt{main}$*. The* initial heap *H_ι contains a single mapping*

R-GOTO

$$\langle R, \varsigma, M^{\pi}\{\mathtt{goto}(\ell)\}, D, H\rangle \rightarrow_{CT} \langle R, \varsigma, M^{\pi}\{M^{\pi}(\ell)\}, D, H\rangle$$

R-RETURN

$$\langle R, \mathtt{v} :: \varsigma, M^{\pi}\{\mathtt{return}\}, (R_0, \varsigma_0, M_0^{\pi_0}\{\mathtt{B}\}) :: D, H\rangle \rightarrow_{CT} \langle R_0, \mathtt{v} :: \varsigma_0, M_0^{\pi_0}\{\mathtt{B}\}, D, H\rangle$$

R-CHECK

$$\frac{\mathtt{r} \in R}{\langle R, \varsigma, M^{\pi}\{\mathtt{check(r)} \cdot \mathtt{B}\}, D, H\rangle \rightarrow_{CT} \langle R, \varsigma, M^{\pi}\{\mathtt{B}\}, D, H\rangle}$$

R-ENABLE

$$\frac{\mathtt{r} \in \pi}{\langle R, \varsigma, M^{\pi}\{\mathtt{enable(r)} \cdot \mathtt{B}\}, D, H\rangle \rightarrow_{CT} \langle R \cup \{\mathtt{r}\}, \varsigma, M^{\pi}\{\mathtt{B}\}, D, H\rangle}$$

R-NEW

$$\langle R, \varsigma, M^{\pi}\{\mathtt{new\ C} \cdot \mathtt{B}\}, D, H\rangle \rightarrow_{CT} \langle R, h :: \varsigma, M^{\pi}\{\mathtt{B}\}, D, H[h \mapsto \mathtt{new\ C}]\rangle$$

R-INVOKE

$$\frac{H(h) = \mathtt{new\ D} \qquad \mathit{mbody}_{CT}(\mathtt{m}, \mathtt{D}) = M_0^{\pi_0} \qquad \mathit{mtype}_{CT}(\mathtt{m}, \mathtt{D}) = \mathtt{C}_1..\mathtt{C}_n \rightarrow \mathtt{E}}{\begin{array}{c}\langle R, \mathtt{v}_1 :: \cdots :: \mathtt{v}_n :: h :: \varsigma, M^{\pi}\{\mathtt{invoke(C,m)} \cdot \mathtt{B}\}, D, H\rangle \\ \rightarrow_{CT} \\ \langle R \cap \pi_0, \mathtt{v}_1 :: \cdots :: \mathtt{v}_n :: h :: \mathbf{nil}, M_0^{\pi_0}\{M_0(\mathit{entry})\}, (R, \varsigma, M^{\pi}\{\mathtt{B}\}) :: D, H\rangle\end{array}}$$

Figure 4. JVM$_{\text{sec}}$ configuration operational semantics

from a distinguished location r_ι to an instance of Main*, and the* initial stack ς_ι *contains just* r_ι*. Formally,* $\varsigma_\iota \triangleq h_\iota :: \mathbf{nil}$ *and* $H_\iota \triangleq \{h_\iota \mapsto \mathtt{new\ Main}\}$ *The* initial configuration $\mathcal{K}_\iota$ *is then defined as follows:*

$$\mathcal{K}_\iota \triangleq \langle \varnothing, \varsigma_\iota, \varnothing^{\varnothing}\{\mathtt{invoke(Main, main)} \cdot \mathtt{return}\}, \mathbf{nil}, H_\iota\rangle$$

2.2 Examples

Here are some examples that illustrate JVM$_{\text{sec}}$ bytecode and extend the running example begun in the introduction. We use syntactic sugar $(\mathtt{m} : \mathtt{J} = \mathtt{B}) \triangleq (\mathtt{m} : \mathtt{J} = \{\mathit{entry} = \mathtt{B}\})$ to abbreviate methods with only one block (necessarily labeled *entry*) in unlabeled form. Let $\pi_1, \pi_2, \pi_3, \pi_4$ be arbitrary principals such that their intersection includes distinct $\mathtt{r}_1$ and $\mathtt{r}_2$. Then the Foo class introduced in Sect. 1.1 can be rewritten in bytecode as follows, assuming that it is owned by π_4:

```
class Foo extends Object {
   m : (Object, Runnable) → Object =
     invoke(Runnable, run) · return
}^π4
```

We further imagine two subclasses of Runnable that we assume are annotation consistent, one that inspects for a privilege $\mathtt{r}_1$ and one that has no security restrictions:

```
class Bar extends Runnable {
   run : Object → Object = check(r1) · return
}^π1

class Baz extends Runnable {
   run : Object → Object = return
}^π2
```

Later in Sect. 3.3 we will show what types our system assigns to these classes statically, and how link time typings are obtained in Sect. 4.5.

3. Type System Specification

Our type theory is based on *polymorphic subtyping constraints*, which are well-known to be useful in OO settings for e.g. typing binary methods and object self-reference [6]. A constraint representation is also advantageous for the implementation of link time type analysis, as we will observe in Sect. 4. We begin with formal details of the type specification, leading up to a type safety result (Theorem 3.1). However, the reader may wish to skip to Sect. 3.3 for examples and discussion, including discussion of how our statement of type safety anticipates dynamic linking in the implementation.

$\mathtt{c}$	::=	$\mathtt{Pre} \mid \mathtt{Abs}$	*presence constructors*
$\mathtt{V}$	::=	$\mathtt{c} \mid \mathtt{v} \mid \mathtt{c}{*} \mid \mathtt{r} : \mathtt{V}; \mathtt{V}$	*rows*
$\mathtt{R}$	::=	$\{\mathtt{V}\}$	*security effects*
$\mathtt{T}$	::=	$\mathtt{R} \mid \mathtt{X} \mid [\mathtt{T}\,\mathtt{C}] \mid (\mathtt{m} : \mathtt{T})..(\mathtt{m} : \mathtt{T}) \mid \mathtt{T}..\mathtt{T} \xrightarrow{\mathtt{R}} \mathtt{T}$	*types*
$\mathtt{X}$	::=	$\mathtt{v} \mid \mathtt{t} \mid \mathtt{s}$	*type variables*
$\mathtt{S}$	::=	$\mathtt{nil} \mid \mathtt{s} \mid \mathtt{T} :: \mathtt{S}$	*stack types*
C	::=	$\mathtt{T} <: \mathtt{T} \mid C \wedge C \mid \mathbf{true}$	*constraints*

Figure 5. JVM$_{\text{sec}}$ type and constraint syntax

3.1 The Type and Effect Language

The type and constraint grammar of JVM$_{\text{sec}}$ is given in Fig. 5. The type language includes method types $\mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{R}} \mathtt{T}$ where $\mathtt{R}$ is the latent *security effect* of the method. The "guts" of security effects are represented as *row types* $\mathtt{V}$, discussed in more detail below. Object types are denoted $[\mathtt{T}\,\mathtt{C}]$, where $\mathtt{C}$ is the class name of the object and $\mathtt{T}$ is either a type variable or a sequence of method type bindings for the object $(\mathtt{m}_1 : \mathtt{T}_1)..(\mathtt{m}_n : \mathtt{T}_n)$. To accommodate stacks, we introduce *stack types* $\mathtt{S}$, which are just sequences of types.

The type language specified in the grammar is more liberal than what is actually allowed in type judgements. We endow types with *kinding rules* defined in Fig. 6. Kinding rules are necessary for the correct definition of row types, to impose the syntactic constraint that row type variables can mention only a specific subset of fields

$$\frac{\mathtt{X} \in \mathcal{V}_k}{\mathtt{X} : k} \qquad \mathtt{c} : \mathit{Pres} \qquad \mathtt{c}* : \mathit{Row}_R$$

$$\frac{\mathtt{V}_0 : \mathit{Pres} \qquad \mathtt{V}_1 : \mathit{Row}_{R \cup \{\mathtt{r}\}} \qquad \mathtt{r} \notin R}{(\mathtt{r} : \mathtt{V}_0; \mathtt{V}_1) : \mathit{Row}_R} \qquad \frac{\mathtt{V} : \mathit{Row}_\varnothing}{\{\mathtt{V}\} : \mathit{Eff}}$$

$$\frac{\begin{array}{c} \mathit{mtype}_{CT}(\mathtt{m}, \mathtt{C}) = \mathtt{D}_0..\mathtt{D}_n \rightarrow \mathtt{D} \\ [\mathtt{T}_0\, \mathtt{D}_0] : \mathit{Type} \;\cdots\; [\mathtt{T}_n\, \mathtt{D}_n] : \mathit{Type} \qquad [\mathtt{T}\, \mathtt{D}] : \mathit{Type} \qquad \mathtt{R} : \mathit{Eff} \end{array}}{[\mathtt{T}_0\, \mathtt{D}_0]..[\mathtt{T}_n\, \mathtt{D}_n] \xrightarrow{\mathtt{R}} [\mathtt{T}\, \mathtt{D}] : \mathit{Meth}_{\mathtt{m},\mathtt{C}}}$$

$$\frac{\mathit{meths}_{CT}(\mathtt{C}) = \mathtt{m}_1..\mathtt{m}_n \qquad \mathtt{T}_1 : \mathit{Meth}_{\mathtt{m}_1,\mathtt{C}} \;\cdots\; \mathtt{T}_n : \mathit{Meth}_{\mathtt{m}_n,\mathtt{C}}}{(\mathtt{m}_1 : \mathtt{T}_1)..(\mathtt{m}_n : \mathtt{T}_n) : \mathit{Body}_\mathtt{C}}$$

$$\frac{\mathtt{T} : \mathit{Body}_\mathtt{C}}{[\mathtt{T}\, \mathtt{C}] : \mathit{Type}} \qquad \frac{\mathtt{T} : \mathit{Type} \qquad \mathtt{S} : \mathit{Stack}}{(\mathtt{T} :: \mathtt{S}) : \mathit{Stack}} \qquad \mathtt{nil} : \mathit{Stack}$$

Figure 6. Type Kinding Rules

[20], wherever they occur in a type. Context free grammars are not sufficient to express this constraint. *Kinds* k are defined as follows, and we require that type terms be well-kinded:

$$k ::= \mathit{Type} \mid \mathit{Pres} \mid \mathit{Row}_R \mid \mathit{Eff} \mid \mathit{Meth}_{\mathtt{m},\mathtt{C}} \mid \mathit{Body}_\mathtt{C} \mid \mathit{Stack}$$

Here, the kind Row_R is the kind of rows that mention every field *except* those in R. Note that each kind k has a distinct associated set of type variables $\mathcal{V}_k$– except for $\mathit{Meth}_{\mathtt{m},\mathtt{C}}$, since we will not need function type variables in our theory.

Abusing notation, variables of kind *Stack* are denoted $\mathtt{s}$, of kinds Row_R and *Pres* are denoted $\mathtt{v}$, and of kind *Type* and $\mathit{Body}_\mathtt{C}$ are denoted $\mathtt{t}$ or $\mathtt{u}$. We also let $\mathtt{U}$ range over types of kind *Type*. An important feature of the type language is that in any object type $[\mathtt{T}\,\mathtt{C}]$, if $\mathtt{T}$ is a vector of type bindings of kind, the kinding rules for $\mathit{Body}_\mathtt{C}$ restrict the set bound names to be exactly the method names of $\mathtt{C}$, providing an "inlined" width constraint on the form of the type.

Restricted Row Types for Effects Security effects $\mathtt{R}$ represent the security needs of instructions and methods– i.e. the privileges required to execute and invoke them. We use a restricted form of row type to denote security effects. This allows us to leverage techniques developed in previous related work on types for Java access control [20]. As we will see, subtyping is invariant on security effects, i.e. subtyping on rows is just row equality. This allows the implementation to use well-developed algorithms for row type equality constraint solutions [19], with high efficiency since our row field presence types are just nullary type constructors, not arbitrary types as for row types in general.

Specifically, security effects are constructed from *presence constructors* $\mathtt{Pre}$ and $\mathtt{Abs}$, denoting the presence or absence of particular resources in security effects. The row type $\mathtt{Pre}*$ represents a constant row specifying every element as being present, and $\mathtt{Abs}*$ specifies every element as being absent. The row type $r : \mathtt{V}_1; \mathtt{V}_2$ says something particular about the presence or absence of $\mathtt{r}$, along with the information in $\mathtt{V}_2$; note that $\mathtt{V}_1$ and $\mathtt{V}_2$ can both be variables, in which case $\mathtt{r}$ and all remaining elements can be present *or* absent. This sort of polymorphism provides flexibility in our theory, for example if some code block requires a privilege $\mathtt{r}$ to be used, but not $\mathtt{r}'$, then its security effect may be of the form $\{\mathtt{r} : \mathtt{Pre}; \mathtt{r}' : \mathtt{v}; \mathtt{V}\}$, where $\mathtt{r}'$ can be constrained to be $\mathtt{Pre}$ or $\mathtt{Abs}$ at a later stage if conditions require. When typing method bodies, the effect of the body is constrained to only allow presence of privileges allowed to the method owner, so for example if $\pi = \{r\}$ owns a method $\mathtt{m}$, then the security effect of the body of $\mathtt{m}$ will be of the form $\{\mathtt{r} : \mathtt{V}; \mathtt{Abs}*\}$.

Interpretation of Constraints Subtyping constraints C are conjunctions of *atomic* constraints of the form $\mathtt{T} <: \mathtt{T}$, with $\mathbf{true}$ being the trivial constraint. We also let D and E range over constraints. We require that in any atomic constraint $\mathtt{T} <: \mathtt{U}$ that $\mathtt{T}$ and $\mathtt{U}$ be of the same kind $k \neq \mathit{Body}_\mathtt{C}$ for any $\mathtt{C}$, disallowing constraints on naked class type bodies.

Constraints are interpreted in a regular tree model, the full details of which we omit here for brevity. It is essentially a combination of the models in [22] and [20] with new features to accommodate stack types. *Interpretations* μ map types of kind k to elements of the semantic structure for each kind, denoted $[\![k]\!]$, which are sets of regular trees in the usual sense. A primitive subtyping relation $\preccurlyeq$ is defined as a partial order over these structures. The relation can be understood as the logical equivalent of consistent closure as defined in Sect. 3 (see Lemma 4.3), so details of subtyping can be gleaned from that definition. The major points are that the interpretation obtains width and depth subtyping for class types while incorporating standard nominal JVM subtyping, and specifies subtyping on row types as row type equality. The interpretation of subtyping constraints is formalized on the basis of the model as follows.

DEFINITION 3.1 (Interpretation of Constraints). *We axiomitize the relation $\mu \vdash_{CT} C$, pronounced μ CT* satisfies *or CT* solves *C, as follows:*

$$\mu \vdash_{CT} \mathbf{true} \qquad \frac{\mu(\mathtt{S}) \preccurlyeq \mu(\mathtt{T}) \text{ given } CT}{\mu \vdash_{CT} \mathtt{S} <: \mathtt{T}}$$

$$\frac{\mu \vdash_{CT} C \qquad \mu \vdash_{CT} D}{\mu \vdash_{CT} C \wedge D}$$

The relation $C \Vdash_{CT} D$, pronounced C CT entails *D, holds iff $\mu \vdash_{CT} C$ implies $\mu \vdash_{CT} D$ for all interpretations μ. Constraints C and D are CT* equivalent, *written $C =_{CT} D$, iff $C \Vdash_{CT} D$ and $D \Vdash_{CT} C$.*

For example, given a class table CT containing classes $\mathtt{A}$ and $\mathtt{B}$ as defined previously and letting $C = [\mathtt{t}_0\, \mathtt{A}] <: [\mathtt{t}_1\, \mathtt{B}] \wedge [\mathtt{t}_1\, \mathtt{B}] <: [\mathtt{t}_2\, \mathtt{B}]$, then C has a solution, and the relation $C \Vdash_{CT} [\mathtt{t}_0\, \mathtt{A}] <: [\mathtt{t}_2\, \mathtt{B}]$ holds. A constraint of the form $[\mathtt{T}_0\, \mathtt{B}] <: [\mathtt{T}_1\, \mathtt{A}]$ has no solution due to the declared nominal relation $\mathtt{A} <: \mathtt{B}$.

3.2 Type Judgements and Validity

Types are logically assigned to JVM$_{\text{sec}}$ programs via type judgement derivations. Since we use a constraint representation of types, it is also necessary to restrict typing judgements to mention only solvable constraints. Subsequently, when considering type inference, we will show that solvability is decidable.

Execution in JVM$_{\text{sec}}$ is based on evaluation of configurations, which comprise a number of elements. Since type safety relates typing and execution, typing must apply to these elements. Thus, different type judgement forms need to be defined. The fundamental judgement is typing of instructions $\Gamma, \mathcal{L} \vdash C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{T}$, where Γ is a class and object reference type environment, $\mathcal{L}$ is a mapping from labels to types, $\mathtt{S}$ is the type of the argument stack for $\mathtt{B}$, the security effect $\mathtt{R}$ is a static approximation of which privileges $\mathtt{B}$ needs to execute, and $\mathtt{T}$ is the type returned by $\mathtt{B}$. We require that $\mathtt{T}$ be of kind *Type*.

To accommodate polymorphism, we introduce constrained type schemes of the form $\forall \mathtt{X}_1..\mathtt{X}_n[C].\mathtt{T}$, ranged over by σ. Letting $\mathrm{fv}(\mathtt{T})$ denote the free variables in $\mathtt{T}$, if $\mathtt{X}_1..\mathtt{X}_n \cap \mathrm{fv}(\mathtt{T}) = \varnothing$ we abbreviate $\forall \mathtt{X}_1..\mathtt{X}_n[\mathbf{true}].\mathtt{T}$ as $\mathtt{T}$. *Substitutions* mapping type variables to types are denoted $[\mathtt{T}_1/\mathtt{X}_n..\mathtt{T}_n/\mathtt{X}_n]$, and are extended to types in

T-BLOCKS
$$\frac{\mathcal{L}(\ell_1) = \mathtt{S}_1, \mathtt{R}_1, \mathtt{T}_1 \cdots \mathcal{L}(\ell_n) = \mathtt{S}_n, \mathtt{R}_n, \mathtt{T}_n \qquad \Gamma, \mathcal{L} \vdash C, \mathtt{S}_1, \mathtt{R}_1 \rhd \mathtt{B}_1 : \mathtt{T}_1 \cdots \Gamma, \mathcal{L} \vdash C, \mathtt{S}_n, \mathtt{R}_n \rhd \mathtt{B}_n : \mathtt{T}_n}{\Gamma, \mathcal{L}, C \vdash \{\ell_1 : \mathtt{B}_1; \ldots; \ell_n : \mathtt{B}_n\}}$$

T-METH
$$\frac{\begin{array}{c} \mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = M^{\{\mathtt{r}_1, \ldots, \mathtt{r}_n\}} \qquad \Gamma, \mathcal{L}, C \vdash M \qquad \mathcal{L}(\mathit{entry}) = \mathtt{S}_1, \mathtt{R}_1, \mathtt{U}_1 \qquad \Gamma(\mathtt{C}).\mathtt{m} = \mathtt{T}_1..\mathtt{T}_j \xrightarrow{\mathtt{R}_0} \mathtt{U}_0 \\ C \Vdash_{CT} \mathtt{U}_1 <: \mathtt{U}_0 \wedge \mathtt{T}_1 :: \cdots :: \mathtt{T}_j :: \Gamma(\mathtt{C}) :: \mathtt{S}_0 <: \mathtt{S}_1 \qquad C \Vdash_{CT} \mathtt{R}_1 <: \{\mathtt{r}_1 : \mathtt{V}_1; \ldots; \mathtt{r}_n : \mathtt{V}_n; \mathtt{Abs}*\} \wedge \{\mathtt{r}_1 : \mathtt{V}_1; \ldots; \mathtt{r}_n : \mathtt{V}_n; \mathtt{V}\} <: \mathtt{R}_0 \end{array}}{\Gamma, C \vdash \mathtt{m}, \mathtt{C}}$$

T-CLASS
$$\frac{\Gamma; \mathtt{C} : [\mathtt{T}\,\mathtt{C}], C \vdash \mathtt{m}, \mathtt{C} \text{ for all } \mathtt{m} \in \mathit{meths}_{CT}(\mathtt{C}) \qquad C \Vdash_{CT} [\mathtt{T}\,\mathtt{C}] <: [\mathtt{U}\,\mathtt{C}] \qquad \mathtt{X}_1..\mathtt{X}_n \cap \mathrm{fv}(\Gamma) = \varnothing}{\Gamma \vdash \mathtt{C} : \forall \mathtt{X}_1..\mathtt{X}_n[C].[\mathtt{U}\,\mathtt{C}]}$$

Figure 8. JVM$_{\text{sec}}$ Typing Rules for Blocks, Methods, and Classes

T-OBJ
$$\frac{\Gamma(\mathtt{C}) = \forall \bar{\mathtt{X}}[D].\mathtt{T} \qquad C \Vdash_{CT} D[\bar{\mathtt{T}}/\bar{\mathtt{X}}]}{\Gamma, C \vdash \mathtt{new}\ \mathtt{C} : \mathtt{T}[\bar{\mathtt{T}}/\bar{\mathtt{X}}]}$$

T-LOC
$$\Gamma, C \vdash h : \Gamma(h)$$

T-STACK-EMPTY
$$\Gamma, C \vdash \mathbf{nil} : \mathtt{nil}$$

T-STACK-CONS
$$\frac{\Gamma, C \vdash \mathtt{v} : \mathtt{T} \qquad \Gamma, C \vdash \varsigma : \mathtt{S}}{\Gamma, C \vdash (\mathtt{v} :: \varsigma) : (\mathtt{T} :: \mathtt{S})}$$

T-HEAP
$$\frac{\Gamma, C \vdash H(h) : \Gamma(h) \text{ for all } h \in \mathit{dom}(H)}{\Gamma, C \vdash H}$$

T-DUMP-EMPTY
$$\Gamma \vdash C, \mathtt{R} \rhd \mathbf{nil}[\mathtt{T}] : \mathtt{T}$$

T-DUMP-CONS
$$\frac{\begin{array}{c} \pi = \{\mathtt{r}_1, \ldots, \mathtt{r}_n\} \qquad \pi \cap R = \{\mathtt{r}_1, \ldots, \mathtt{r}_j\} \qquad \mathtt{R} = \{\mathtt{r}_1 : \mathtt{Pre}; \ldots; \mathtt{r}_j : \mathtt{Pre}; \mathtt{r}_{j+1} : \mathtt{V}_{j+1}; \ldots; \mathtt{r}_n : \mathtt{V}_n; \mathtt{Abs}*\} \\ \Gamma, \mathcal{L}, C \vdash M \qquad \Gamma, C \vdash \varsigma : \mathtt{S} \qquad \Gamma, \mathcal{L} \vdash C, \mathtt{U}_0 :: \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{U}_1 \qquad \Gamma \vdash C, \mathtt{R} \rhd D[\mathtt{U}_1] : \mathtt{T} \end{array}}{\Gamma \vdash C, \{\mathtt{r}_1 : \mathtt{V}_1; \ldots; \mathtt{r}_n : \mathtt{V}_n; \mathtt{Abs}*\} \rhd ((R, \varsigma, M^{\pi}\{\mathtt{B}\}) :: D)[\mathtt{U}_0] : \mathtt{T}}$$

T-CONFIG
$$\frac{\Gamma, C \vdash \mathtt{H} \qquad \Gamma, C \vdash \mathtt{v} : \mathtt{U} \qquad \Gamma \vdash C, \mathtt{R} \rhd ((R, \varsigma, M^{\pi}\{\mathtt{B}\}) :: D)[\mathtt{U}] : \mathtt{T}}{\Gamma, C, \mathtt{R} \vdash \langle R, \mathtt{v} :: \varsigma, M^{\pi}\{\mathtt{B}\}, D, H\rangle : \mathtt{T}}$$

Figure 9. JVM$_{\text{sec}}$ Runtime Entity Typing Rules

the usual manner. We require that substitutions be idempotent. Environments are specified as follows. Environments contain typing assumptions for classes, and also for run-time heap locations:

$$\Gamma ::= \varnothing \mid \Gamma; (\mathtt{C} : \sigma) \mid \Gamma; (h : \mathtt{T}) \qquad \textit{type environments}$$

Lookup in an environment, denoted $\Gamma(C)$ or $\Gamma(h)$, is defined as usual, returning the rightmost binding for the given parameter in Γ and undefined if the parameter has no binding.

Typing rules for selected instructions are defined in Fig. 7 (we omit a full listing for brevity). We use the notation $[\mathtt{T}\,\mathtt{C}].\mathtt{m}$ to abbreviate $\mathtt{T}'$ such that $(\mathtt{m} : \mathtt{T}') \in \mathtt{T}$, implying that $\mathtt{T}$ must be a sequence of method type bindings of kind $\mathit{Body}_{\mathtt{C}}$ for the notation to be defined. Note that this restricts the form of $\mathtt{T}_0$ in the T-INVOKE rule. A weakening rule (T-WEAKEN) integrates subtyping with type judgements, which is contravariant in the type of the stack. Recalling contravariance of subtyping for function domain types, this is necessary since the type of the stack represents the types of instruction arguments. Typing rules for blocks, methods, and classes are given in Fig. 8. Label typings $\mathcal{L}$ are local to method bodies, since method labels are. The stack type for method typing reflects that the method expects `self` at the bottom of its argument stack. Now, we can define well-formedness of type environments with respect to a given class table:

DEFINITION 3.2 (Realizability of Environments). *We say that Γ is CT realizable iff every class type binding in Γ is of the form $\mathtt{C} : \forall \mathtt{X}_1..\mathtt{X}_n[D].[\mathtt{T}\,\mathtt{C}]$ and D is CT solvable.*

Validity of typing is then predicated on both derivability and realizability of type judgements, as follows.

DEFINITION 3.3 (Type Judgement Validity). *A type judgement is CT* valid *iff it is derivable given CT, and for any Γ and C occurring in the judgement, Γ is CT realizable and C is CT solvable.*

Type Safety Now we are ready to consider type safety. The result is stated to anticipate the implementation of link time type inference defined in the next section. In the JVM, when configurations evaluate with linked and loaded class tables, it is only required that linked classes are verified on the basis of type declarations in the loaded class table. So, we formalize the idea of sound typing of a class table (the linked class table), on the basis of type declarations in a superset of that class table (the loaded class table).

DEFINITION 3.4 (Class Table Typing). *The environment Γ is CT'* sound *for a class table CT iff $CT \subseteq CT'$ and for all $\mathtt{C} \in \mathit{dom}(CT)$ the judgement $\Gamma \vdash \mathtt{C} : \Gamma(C)$ is CT' valid.*

Intuitively, our formal results guarantee safe configuration reduction by well-typedness of the class table linkages allowing that reduction, rather than whole-program typing. Most of the proofs are omitted here for brevity.

In order to formulate type safety, we extend typing judgements to run-time entities in Fig. 9. Notable among these rules is the judgement form for dumps. Observe that a dump is a low-level evaluation context, with the top of the stack being the "hole" in the

T-Return
$C, \mathtt{T} :: \mathtt{S}, \mathtt{R} \rhd \mathtt{return} : \mathtt{T}$

T-Goto
$$\frac{\mathcal{L}(\ell) = \mathtt{S}, \mathtt{R}, \mathtt{T}}{C, \mathtt{S}, \mathtt{R} \rhd \mathtt{goto}(\ell) : \mathtt{T}}$$

T-Enable
$$\frac{C, \mathtt{S}, \{\mathtt{r} : \mathtt{Pre}; \mathtt{V}_1\} \rhd \mathtt{B} : \mathtt{T}}{C, \mathtt{S}, \{\mathtt{r} : \mathtt{V}_0; \mathtt{V}_1\} \rhd \mathtt{enable}(\mathtt{r}) \cdot \mathtt{B} : \mathtt{T}}$$

T-Check
$$\frac{C, \mathtt{S}, \{\mathtt{r} : \mathtt{Pre}; \mathtt{V}_0\} \rhd \mathtt{B} : \mathtt{T}}{C, \mathtt{S}, \{\mathtt{r} : \mathtt{Pre}; \mathtt{V}_0\} \rhd \mathtt{check}(\mathtt{r}) \cdot \mathtt{B} : \mathtt{T}}$$

T-Invoke
$$\frac{C, \mathtt{U} :: \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{T} \qquad C \Vdash_{CT} \mathtt{T}_0.\mathtt{m} <: \mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{R}} \mathtt{U}}{C, \mathtt{T}_1 :: \cdots :: \mathtt{T}_n :: [\mathtt{T}_0\ \mathtt{C}] :: \mathtt{S}, \mathtt{R} \rhd \mathtt{invoke}(\mathtt{C}, \mathtt{m}) \cdot \mathtt{B} : \mathtt{T}}$$

T-New
$$\frac{\Gamma(\mathtt{C}) = \forall \mathtt{X}_1..\mathtt{X}_n[D].\mathtt{T}_0 \qquad C \Vdash_{CT} D[\mathtt{T}_1/\mathtt{X}_n..\mathtt{T}_n/\mathtt{X}_n] \qquad C, \mathtt{T}_0[\mathtt{T}_1/\mathtt{X}_n..\mathtt{T}_n/\mathtt{X}_n] :: \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{T}}{C, \mathtt{S}, \mathtt{R} \rhd \mathtt{new}\ \mathtt{C} \cdot \mathtt{B} : \mathtt{T}}$$

T-Weaken
$$\frac{C, \mathtt{S}_0, \mathtt{R}_0 \rhd \mathtt{B} : \mathtt{T} \qquad C \Vdash_{CT} \mathtt{S} <: \mathtt{S}_0 \wedge \mathtt{R}_0 <: \mathtt{R}}{C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{T}}$$

Figure 7. JVM$_{\text{sec}}$ Instruction Typing

context. Hence, the judgement form $C, \mathtt{R} \rhd D[\mathtt{T}_1] : \mathtt{T}_2$ assigns the type $\mathtt{T}_2$ as the return type of the dump, assuming that a value of type $\mathtt{T}_1$ is pushed on the stack. This technique allows us to demonstrate subject reduction as the foundation of type safety, as opposed to previous techniques that require a big-step operational view [14]. Subject reduction is stated as follows. By viewing CT as the linked class table and CT' as the loaded class table, it becomes clear how this statement formally anticipates link time typing.

Lemma 3.1 (Subject Reduction). *Given an environment Γ that is CT' sound for CT, and $\mathcal{K}_1 \rightarrow_{CT} \mathcal{K}_2$, and suppose $\Gamma, C, \mathtt{R} \vdash \mathcal{K}_1 : \mathtt{T}$ is CT' valid. Then there exists a CT' valid judgement $\Gamma', \mathcal{L}, C, \mathtt{R} \vdash \mathcal{K}_2 : \mathtt{T}'$, where $C \Vdash_{CT'} \mathtt{T}' <: \mathtt{T}$ and Γ' is an extension of Γ to new heap locations in $\mathcal{K}_2$.*

While our subject reduction result guarantees standard type safety properties, for example well-formedness of method invocation, we focus here on how the result guarantees that stack inspection checks will succeed. First, we observe that the top level security effect $\mathtt{R}$ in a sound typing of a configuration where the next instruction is a check for a resource $\mathtt{r}$ will definitely reflect whether $\mathtt{r}$ has been activated.

Lemma 3.2 (Type Form of Checks). *Given a CT valid judgement as follows:*

$$\Gamma, C, \mathtt{R} \vdash \langle R, \mathtt{v} :: \varsigma, M^{\pi}\{\mathtt{check}(\mathtt{r}) \cdot \mathtt{B}\}, D, H\rangle : \mathtt{T}$$

Then $C \Vdash_{CT} \{\mathtt{r} : \mathtt{Pre}; \mathtt{V}\} <: \mathtt{R}$ for some $\mathtt{V}$ iff $\mathtt{r} \notin R$.

On the basis of these results, we can demonstrate that sound typing of an initial configuration ensures that it will not encounter a failing check during execution.

Theorem 3.1 (Type Enforcement of Checks). *If Γ is CT' sound for CT and $\mathcal{K}_\iota \rightarrow^{\star}_{CT} \langle R, \varsigma, M^{\pi}\{\mathtt{check}(\mathtt{r}) \cdot \mathtt{B}\}, D, H\rangle$, then $\mathtt{r} \in R$.*

Proof. By Definition 2.2, class Main is owned by $\varnothing$. By virtue of this, soundness of Γ, and rules T-Config and T-DumpCons, it follows that a judgement of the form $\Gamma, C, \{\mathtt{Abs}*\} \vdash \mathcal{K}_\iota : \mathtt{T}$ is CT' valid. Thus by Lemma 3.1 and induction on the length of the computation there exists a CT' valid judgement $\Gamma', C, \{\mathtt{Abs}*\} \vdash \langle R, \varsigma, M^{\pi}\{\mathtt{check}(\mathtt{r}) \cdot \mathtt{B}\}, D, H\rangle : \mathtt{T}'$. The result then follows by Lemma 3.2. □

3.3 Examples and Discussion

The formalism just presented allows us to statically assign typings in the manner discussed in Sect. 1.1. In particular, recalling class Foo as defined in Sect. 2.2 and letting T be the type:

$$(\mathtt{Unit}, [\mathtt{run} : \mathtt{Unit} \xrightarrow{\{\mathtt{v}\}} \mathtt{Unit\ Runnable}]) \xrightarrow{\{\mathtt{v}\}} \mathtt{Unit}$$

we observe that the following typing is valid:

$$\mathtt{Foo} : \forall \mathtt{v}[\mathbf{true}].[\mathtt{m} : \mathtt{T\ Foo}]$$

Note that the typing is presented in unconstrained form for clarity. The polymorphic effect v is quantified, so that different instances of the class can be independently constrained. For example, recall also the classes Bar and Baz introduced in Sect. 2.2, that can be statically assigned the following types in our system:

$$\mathtt{Bar} \ : \ \forall \mathtt{v}_3[\mathbf{true}].[\mathtt{run} : \mathtt{Unit} \xrightarrow{\{\mathtt{r}_1 : \mathtt{Pre}; \mathtt{v}_3\}} \mathtt{Unit\ Bar}]$$
$$\mathtt{Baz} \ : \ \forall \mathtt{v}_4[\mathbf{true}].[\mathtt{run} : \mathtt{Unit} \xrightarrow{\{\mathtt{v}_4\}} \mathtt{Unit\ Baz}]$$

Now, imagine that some program contains the following code snippets:

$$\mathtt{new\ Foo} \cdot \mathtt{new\ Bar} \cdot \mathtt{new\ Object} \cdot \mathtt{invoke}(\mathtt{Foo}, \mathtt{m}) \qquad (1)$$
$$\mathtt{new\ Foo} \cdot \mathtt{new\ Baz} \cdot \mathtt{new\ Object} \cdot \mathtt{invoke}(\mathtt{Foo}, \mathtt{m}) \qquad (2)$$

Types of the different Foo instances created by the first instruction in each snippet can instantiate v differently, say with $\mathtt{v}_{(1)}$ and $\mathtt{v}_{(2)}$ respectively. The constraints imposed by typing of this code will then entail the following relations, assuming that the type instances of Bar and Baz are instantiated trivially:

$$\mathtt{r}_1 : \mathtt{Pre}; \mathtt{v}_3 <: \mathtt{v}_{(1)} \qquad \mathtt{v}_4 <: \mathtt{v}_{(2)}$$

However, it is important to observe that a relation between $\mathtt{v}_{(1)}$ and $\mathtt{v}_{(2)}$ is *not* imposed in general, so the types assigned to snippets (1) and (2) accurately reflect their independent security requirements, unlike the join of all effects approach [14], which will in fact reject snippet (2) outside of a context where $\mathtt{r}_1$ is activated, violating the principle of least privilege.

How Static Type Safety Anticipates Link Time Safety Now, let CT be a class table containing the definitions of Foo, Bar, and Baz as in Sect. 2.2, let CT' be a class table containing only the definition of Foo, and suppose Γ' contains only the above binding for Foo and the binding $\mathtt{Bar} : \forall \mathtt{t}[\mathbf{true}].[\mathtt{t\ Bar}]$. Although this latter binding is abstract and not a sound typing for Bar, the environment Γ' is CT sound for CT' by definition. Observe that a CT derivable typing of the code snippet (1) given Γ' can assign a purely abstract typing for the effect of this snippet, since the type of Bar is abstract. Clearly this does not reflect the run-time security requirements of the invocation of Bar's version of run during execution of the snippet given CT. However, given CT', this code snippet can only execute up to the invocation of Foo's method m, since Bar is not in CT' hence $mbody_{CT'}(\mathtt{run}, \mathtt{Bar})$ is undefined. Furthermore, the typing obtained given Γ' does ensure safety up to the point of that instruction. Therefore, since Γ' is CT sound for CT', this ensures safety for program execution given CT'. While progress in general cannot be guaranteed due to remaining class linkage issues, Theorem 3.1 does ensure progress of stack inspections. What remains is to devise a general way to *reify* class typings at link time without requiring re-analysis of linked code, that will in

$$\frac{\mathit{mtype}_{CT}(\mathtt{m},\mathtt{C}) = \mathtt{D}_1..\mathtt{D}_n \rightarrow \mathtt{D}}{\mathit{texpand}_{CT}(\mathtt{m},\mathtt{C}) = [\mathtt{t}_1\,\mathtt{D}_1]..[\mathtt{t}_n\,\mathtt{D}_n] \xrightarrow{\{\mathtt{v}\}} [\mathtt{t}\,\mathtt{D}]}$$

$$\frac{\mathit{meths}_{CT}(\mathtt{C}) = \mathtt{m}_1..\mathtt{m}_n}{\begin{array}{c}\mathit{texpand}_{CT}(\mathtt{C}) = \\ [(\mathtt{m}_1 : \mathit{texpand}_{CT}(\mathtt{m}_1,\mathtt{C}))..(\mathtt{m}_n : \mathit{texpand}_{CT}(\mathtt{m}_n,\mathtt{C}))\ \ \mathtt{C}]\end{array}}$$

Figure 10. *texpand* Type Construction

this example concretize the purely abstract binding for $\mathtt{A}$ in Γ' with the sound binding for $\mathtt{A}$ in Γ at link time, as well as types of code derived under Γ' that depend on this binding. We consider these issues in the next section.

4. Type System Implementation

We now consider the implementation of link time type analysis based on our type theory. We will show that an algorithm exists for inferring valid type judgements, based on constraint *closure*. In addition to these standard type inference features, we also specify an implementation of link time typing in the presence of dynamic loading and linking. Essentially, we will extend the machine model to include an environment representing the types of linked classes. When a class is loaded, an abstract type for it is added to the environment. When a class is dynamically linked, the class is verified by typing and the environment is reified with the new class typing, or rejected as ill-typed. Our approach does not require any sort of recomputation of previously inferred types– in fact, the link time analysis can be viewed as the same as static analysis, but performed incrementally over the course of computation. Correctness of the algorithms with respect to the logical specification in the previous section obtains a link time type safety result for the implementation.

4.1 Link Time Typing: Intuitions

When class types are inferred, the type of a class $\mathtt{C}$ is of the form $\forall\mathtt{X}_1..\mathtt{X}_n[C].[\mathtt{t}\ \mathtt{C}]$, where the constraint C forms a lower bound on the abstract type $\mathtt{t}$. This constraint is the main artifact of type inference. However, since type inference is a verification technique, at load time this constraint is not available. But dynamic loading and linking in Java allows linked code to reference unverified classes; hence, when typing code that references an unlinked class $\mathtt{C}$, some type must be assumed for $\mathtt{C}$. We call this the *load version* typing of $\mathtt{C}$.

A load version class typing is a maximally general class type of the form $\forall\mathtt{t}[\mathbf{true}].[\mathtt{t}\ \mathtt{C}]$. At link time the appropriate constraint on $\mathtt{t}$ will be generated by inference. The process of *reification* fills in the constraint. Both the type binding for $\mathtt{C}$, and every instruction $\mathtt{new\ C}$ in linked code that will have been typed by an instance of $\mathtt{C}$'s load version, needs to be reified. Each of these reified instances must be uniquely instantiated to enjoy the full benefit of polymorphism in our system. It is also critical to distinguish load version typings of objects from the typing of method parameters, since the latter are directly lower bounded at method usage points, not by class typings.

Thus, for each class name $\mathtt{C}$ in a given class table, we distinguish a countable subset of variables in $\mathcal{V}_{\mathit{Body}_\mathtt{C}}$, denoted $\mathtt{t}^\mathtt{C}$. These *marked* variables differentiate load version class typings from types of method parameters. They are not interpreted any differently from normal type variables by any processes other than type instantiation and reification. The normal machinery of polymorphism allows to differentiate distinct type instances, by requiring that whenever types are instantiated by *renaming* quantified variables, marked

I-Return
$$\mathbf{true}, \mathtt{t} :: \mathtt{s}, \mathtt{v} \rhd \mathtt{return} : \mathtt{t}$$

I-Goto
$$\frac{\mathcal{L}(\ell) = \mathtt{s},\mathtt{v},\mathtt{t}}{\mathbf{true}, \mathtt{s}, \mathtt{v} \rhd \mathtt{goto}(\ell) : \mathtt{t}}$$

I-Enable
$$\frac{C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}}{C \wedge \mathtt{R} <: \{\mathtt{r} : \mathtt{Pre}; \mathtt{v}_1\}, \mathtt{S}, \{\mathtt{r} : \mathtt{v}_0; \mathtt{v}_1\} \rhd \mathtt{enable}(\mathtt{r}) \cdot \mathtt{B} : \mathtt{t}}$$

I-Check
$$\frac{C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}}{C \wedge \mathtt{R} <: \{\mathtt{r} : \mathtt{Pre}; \mathtt{v}\}, \mathtt{S}, \mathtt{R} \rhd \mathtt{check}(\mathtt{r}) \cdot \mathtt{B} : \mathtt{t}}$$

I-Invoke
$$\frac{\begin{array}{c} C_0, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t} \\ \mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{v}} \mathtt{U} = \mathit{texpand}_{CT}(\mathtt{m},\mathtt{C}) \qquad [\mathtt{T}_0\ \mathtt{C}] = \mathit{texpand}_{CT}(\mathtt{C}) \\ C =_{set} C_0 \wedge \mathtt{T}_0.\mathtt{m} <: \mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{v}} \mathtt{U} \wedge \mathtt{U} :: \mathtt{s} <: \mathtt{S} \wedge \mathtt{R} <: \mathtt{v} \end{array}}{C, \mathtt{T}_1 :: \cdots :: \mathtt{T}_n :: [\mathtt{T}_0\ \mathtt{C}] :: \mathtt{s}, \mathtt{v} \rhd \mathtt{invoke}(\mathtt{C},\mathtt{m}) \cdot \mathtt{B} : \mathtt{t}}$$

I-New
$$\frac{\begin{array}{c} C_0, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t} \\ \Gamma(\mathtt{C}) = \forall\mathtt{X}_1..\mathtt{X}_n[D].\mathtt{T}_0 \qquad C =_{set} C_0 \wedge \rho(D) \wedge \rho(\mathtt{T}_0) :: \mathtt{s} <: \mathtt{S} \end{array}}{C, \mathtt{s}, \mathtt{R} \rhd \mathtt{new\ C} \cdot \mathtt{B} : \mathtt{t}}$$

Figure 11. JVM$_{\text{sec}}$ Instruction Type Inference

variables are substituted for marked variables, so markings promulgate appropriately.

Definition 4.1. *A* renaming *is a total substitution mapping type variables to type variables. For each $\mathcal{V}_{\text{Body}_\mathtt{C}}$ we distinguish a strict, countable subset of* marked *variables ranged over by $\mathtt{t}^\mathtt{C}$. We require that for all ρ and $\mathtt{t}_0^\mathtt{C}$ there exists $\mathtt{t}_1^\mathtt{C}$ such that $\rho(\mathtt{t}_0^\mathtt{C}) = \mathtt{t}_1^\mathtt{C}$. The* load version *of a class $\mathtt{C}$, denoted $\mathit{loadvers}(\mathtt{C})$, is defined as follows:*

$$\mathit{loadvers}(\mathtt{C}) = \forall\mathtt{t}^\mathtt{C}[\mathbf{true}].[\mathtt{t}^\mathtt{C}\ \mathtt{C}]$$

Hereafter we assume that the type implementation uses marked variables only in load version typings and their renamings.

4.2 Type Inference

Derivable typing judgements are reconstructed by type inference. The fundamental judgement in type inference applies to instructions, is written $\Gamma, \mathcal{L} \vdash_W C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}$, and is entirely analogous to typing judgements for instructions. The W in the inference relation symbol is intended to evoke the original polymorphic type inference algorithm. When Γ and $\mathcal{L}$ are clear from context, these judgements may be abbreviated as $C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}$. In inference, the label environment $\mathcal{L}$ maps distinct labels in its domain to distinct type variable triples $\mathtt{s}, \mathtt{v}, \mathtt{t}$, a form we call *canonical*.

Selected instruction type inference rules are given in Fig. 11 (we omit a full listing for brevity). These rules and later definitions use the following notation for brevity, which allows constraints to be viewed as sets of *atomic constraints*:

Definition 4.2. *Let $\hat{C}$ range over* atomic *constraints, i.e. constraints of the form $\mathbf{true}$ or $\mathtt{T} <: \mathtt{U}$, and for all $C = \hat{C}_1 \wedge \cdots \wedge \hat{C}_n$, let $\mathit{set}(C) = \{C_1, \ldots, C_n\}$. Then define:*

$$C =_{set} D \iff \mathit{set}(C) = \mathit{set}(D)$$

$$D \subseteq C \iff \mathit{set}(D) \subseteq \mathit{set}(C) \qquad \hat{C} \in C \iff \hat{C} \in \mathit{set}(C)$$

$$\forall\mathtt{X}_1..\mathtt{X}_n[C].\mathtt{T} =_{set} \forall\mathtt{X}_1..\mathtt{X}_n[D].\mathtt{T} \iff C =_{set} D$$

Clearly $C =_{set} D$ implies $C =_{CT}$ for all CT.

I-Block

$$\frac{\mathcal{L}(\ell) = \mathtt{s}, \mathtt{v}, \mathtt{u} \qquad \Gamma, \mathcal{L} \vdash_W C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}}{\Gamma, \mathcal{L}, C \wedge \mathtt{s} <: \mathtt{S} \wedge \mathtt{R} <: \mathtt{v} \wedge \mathtt{t} <: \mathtt{u} \vdash_W \ell, \mathtt{B}}$$

I-Blocks

$$\frac{\Gamma, \mathcal{L}, C_1 \vdash_W \ell_1, \mathtt{B}_1 \cdots \Gamma, \mathcal{L}, C_n \vdash_W \ell_n, \mathtt{B}_n}{\Gamma, \mathcal{L}, C_1 \wedge \cdots \wedge C_n \vdash_W \{\ell_1 : \mathtt{B}_1; \ldots ; \ell_n : \mathtt{B}_n\}}$$

I-Meth

$$\frac{\begin{array}{c} \mathit{mbody}_{CT}(\mathtt{m}, \mathtt{C}) = M^{\{\mathtt{r}_1, \ldots, \mathtt{r}_n\}} \qquad \Gamma(\mathtt{C}).\mathtt{m} = \mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{v}_0} \mathtt{U} \\ \mathcal{L}(\mathit{entry}) = \mathtt{s}_1, \mathtt{v}_1, \mathtt{t}_1 \qquad \Gamma, \mathcal{L}, C_0 \vdash_W M \\ C_1 =_{set} \mathtt{t}_1 <: \mathtt{U} \wedge \mathtt{T}_1 :: \cdots :: \mathtt{T}_j :: \Gamma(\mathtt{C}) :: \mathtt{s}_0 <: \mathtt{s}_1 \\ C_2 =_{set} \mathtt{v}_1 <: \{\mathtt{r}_1 : \mathtt{w}_1; \ldots ; \mathtt{r}_n : \mathtt{w}_n; \mathtt{Abs}*\} \\ C_3 =_{set} \{\mathtt{r}_1 : \mathtt{w}_1; \ldots ; \mathtt{r}_n : \mathtt{w}_n; \mathtt{w}\} <: \mathtt{v}_0 \end{array}}{\Gamma, C_0 \wedge C_1 \wedge C_2 \wedge C_3 \vdash_W \mathtt{m}, \mathtt{C}}$$

I-Class

$$\frac{\begin{array}{c} \mathit{meths}_{CT}(\mathtt{C}) = \mathtt{m}_1..\mathtt{m}_n \qquad \mathtt{T} = \mathit{texpand}_{CT}(\mathtt{C}) \\ \mathtt{X}_1..\mathtt{X}_n = \mathrm{fv}(C) \qquad \Gamma; \mathtt{C} : \mathtt{T}, C_i \vdash_W \mathtt{m}_i, \mathtt{C} \text{ for all } \mathtt{m}_i \in \mathtt{m}_1..\mathtt{m}_n \\ C =_{set} C_1 \wedge \cdots \wedge C_n \wedge \mathtt{T} <: [\mathtt{t}\,\mathtt{C}] \end{array}}{\Gamma \vdash_W \mathtt{C} : \forall \mathtt{X}_1..\mathtt{X}_n[C].[\mathtt{t}\,\mathtt{C}]}$$

Figure 12. JVM$_{\text{sec}}$ Type Inference for Blocks, Methods, and Classes

The inference rules are nondeterministic in the choice of type variables. We call *canonical* those derivations that choose globally fresh variables whenever possible, both explicitly, and via renamings and *texpand* construction as defined in Fig. 10. Hereafter we assume that all derivations are canonical. A formal definition of global freshness is easily obtained by adapting techniques such as in [22], but we omit this definition and associatated machinery for simplicity. The *texpand* construction returns an object type of the form $[\mathtt{T}\,\mathtt{C}]$, where $\mathtt{T}$ includes fields for all the methods in $\mathtt{C}$ with fresh abstract security effects, providing a "skeleton" on which to hang inferred type constraints describing the class. As in most constraint systems, inference proceeds by adding constraints appropriate to expressions in a syntax-directed manner, from the leaves towards the root of expressions. We observe that inference is sound, in the following sense. The result follows by straightforward induction on derivations.

Lemma 4.1. *Given* Γ *and* $\mathcal{L}$*. Then if* $\Gamma, \mathcal{L} \vdash_W C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}$ *is derivable given* CT*, so is* $\Gamma, \mathcal{L} \vdash C, \mathtt{S}, \mathtt{R} \rhd \mathtt{B} : \mathtt{t}$.

Type inference is extended to blocks, methods, and classes in Fig. 12. The rules are unremarkable except for I-Meth, which ties together inferred type constraints and type assumptions in the class and label type environments. By straightforward inversion of these rules and induction on derivations, we are able to obtain the following result, which is in a convenient form for establishing link time type safety in a later section.

Theorem 4.1 (Soundness of Class Type Inference). *Given environment* Γ *such that* $\mathit{fv}(\Gamma) = \varnothing$*. Then if* $\Gamma \vdash_W \mathtt{C} : \forall \mathtt{X}_1..\mathtt{X}_n[C].\mathtt{t}$ *is derivable given* CT*, so is* $\Gamma \vdash \mathtt{C} : \forall \mathtt{X}_1..\mathtt{X}_n[C].\mathtt{t}$.

4.3 Closure and Consistency

To automatically check satisfiability of constraints, the type implementation comprises a constraint closure algorithm and consistency check. We say that a constraint C is CT *consistent* iff $\vdash C : \mathit{ok}$ is derivable given CT, as axiomatized in Fig. 14. These rules assume given a class table CT. We say that an environment Γ is CT consistent iff for all $\mathtt{C} \in \mathit{dom}(\Gamma)$ with $\Gamma(\mathtt{C}) = \forall \mathtt{X}_1..\mathtt{X}_n[C].\mathtt{T}$, it is the case that C is CT consistent.

C-Fn

$$(\mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{R}} \mathtt{T} <: \mathtt{U}_1..\mathtt{U}_n \xrightarrow{\mathtt{R}'} \mathtt{U})$$
$$\rightsquigarrow_{close}$$
$$(\mathtt{U}_1 <: \mathtt{T}_1 \wedge \cdots \wedge \mathtt{U}_n <: \mathtt{T}_n \wedge \mathtt{T} <: \mathtt{U} \wedge \mathtt{R} <: \mathtt{R}')$$

C-Trans

$$(\mathtt{T}_0 <: \mathtt{T}_1 \wedge \mathtt{T}_1 <: \mathtt{T}_2) \rightsquigarrow_{close} \mathtt{T}_0 <: \mathtt{T}_2$$

C-Stack

$$\mathtt{T}_0 :: \mathtt{S}_0 <: \mathtt{T}_1 :: \mathtt{S}_1 \rightsquigarrow_{close} \mathtt{T}_0 <: \mathtt{T}_1 \wedge \mathtt{S}_0 <: \mathtt{S}_1$$

C-Obj

$$[(\mathtt{m}_1 : \mathtt{T}_1)..(\mathtt{m}_n : \mathtt{T}_n)\,\mathtt{C}] <: [(\mathtt{m}_1 : \mathtt{U}_1)..(\mathtt{m}_n : \mathtt{U}_n)..(\mathtt{m}_m : \mathtt{U}_m)\,\mathtt{D}]$$
$$\rightsquigarrow_{close}$$
$$\mathtt{T}_1 <: \mathtt{U}_1 \wedge \cdots \wedge \mathtt{T}_n <: \mathtt{U}_n$$

C-Context

$$\frac{C' \subseteq C \qquad C' \rightsquigarrow_{close} D \qquad D \not\subseteq C}{C \rightarrow_{close} C \wedge D}$$

Figure 13. Constraint Closure (Selected Rules)

$$\vdash \mathbf{true} : \mathit{ok} \qquad \vdash \mathtt{T} <: \mathtt{T} : \mathit{ok} \qquad \frac{\mathtt{X}, \mathtt{T} : k}{\vdash \mathtt{X} <: \mathtt{T} : \mathit{ok}}$$

$$\frac{\mathtt{X}, \mathtt{T} : k}{\vdash \mathtt{T} <: \mathtt{X} : \mathit{ok}} \qquad \vdash \mathtt{T}_0 :: \mathtt{S}_0 <: \mathtt{T}_1 :: \mathtt{S}_1 : \mathit{ok}$$

$$\frac{\vdash C : \mathit{ok} \qquad \vdash D : \mathit{ok}}{\vdash C \wedge D : \mathit{ok}} \qquad \frac{CT \vdash \mathtt{C} <: \mathtt{D}}{\vdash [\mathtt{T}\,\mathtt{C}] <: [\mathtt{S}\,\mathtt{D}] : \mathit{ok}}$$

$$\vdash (\mathtt{T}_1..\mathtt{T}_n \xrightarrow{\mathtt{R}} \mathtt{T} <: \mathtt{U}_1..\mathtt{U}_n \xrightarrow{\mathtt{R}'} \mathtt{U}) : \mathit{ok}$$

Figure 14. Constraint Consistency (Selected Rules)

Well-developed and efficient techniques for solving row type equality constraints have been presented by previous authors [19], so we won't dwell on the issue here. Given any constraint C, a subset of the constraints generated by closure will be row type equality constraints, and for the purposes of this presentation we imagine that they are filtered out and dealt with by existing techniques [19]. Constraint closure for the remaining JVM$_{\text{sec}}$ type forms is defined via the rewrite rules given in Fig. 13 and Definition 4.3.

Definition 4.3 (Closure). *The rewrite rules* $\rightsquigarrow_{close}$ *and* $\rightarrow_{close}$ *are defined in Fig. 13.* C *is* closed *iff there does not exist* D *such that* $C \rightarrow_{close} D$*. The relation* $\rightarrow^{\star}_{close}$ *is the reflexive, transitive closure of* $\rightarrow_{close}$*. We define* $\mathit{close}(C)$ *as a closed constraint such that* $C \rightarrow^{\star}_{close} \mathit{close}(C)$*, and say that* C *is* closed *iff* $C =_{set} \mathit{close}(C)$*. Define* $\mathit{close}(\forall \mathtt{X}_1..\mathtt{X}_n[C].\mathtt{T}) = \forall \mathtt{X}_1..\mathtt{X}_n[\mathit{close}(C)].\mathtt{T}$*, and* $\mathit{close}(\Gamma) = \Gamma'$ *iff* $\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma')$ *and* $\Gamma'(\mathtt{C}) = \mathit{close}(\Gamma(\mathtt{C}))$ *for all* $\mathtt{C} \in \mathit{dom}(\Gamma)$.

We observe that closure doesn't change the logical meaning of a constraint C. Intuitively, it simply makes explicit all the constraints implicit in C for consistency analysis.

Lemma 4.2. $\mathit{close}(C) =_{CT} C$ *for all* C *and* CT.

The following result establishes that consistent closure is equivalent to solvability of constraints. It is obtained by composition of relevant results for row types, and for a similar Java source language

$$\frac{\rho(\mathtt{t}) = \mathtt{t}^{\mathtt{C}}}{\mathit{reify}(\mathtt{t}^{\mathtt{C}}, C, \mathtt{t}) = \rho(C)}$$

$$\frac{\begin{array}{c}\mathtt{t}_1^{\mathtt{D}}..\mathtt{t}_k^{\mathtt{D}} = \mathit{labvars}_{\mathtt{D}}(\mathtt{X}_1..\mathtt{X}_n)\\ \forall i \in 1..k \;.\; \mathit{reify}(\mathtt{t}_i^{\mathtt{D}}, D, \mathtt{u}) = D_i \\ E =_{set} C \wedge D_1 \wedge \cdots \wedge D_k\end{array}}{\mathit{reify}(\forall \mathtt{X}_1..\mathtt{X}_n[C].[\mathtt{t}\,\mathtt{C}], \forall \mathtt{Y}_1..\mathtt{Y}_m[D].[\mathtt{u}\,\mathtt{D}]) = \forall \mathrm{fv}(\mathtt{E},\mathtt{t})[E].[\mathtt{t}\,\mathtt{C}]}$$

Figure 15. Constraint Reification

R-EVAL

$$\frac{\mathcal{K} \rightarrow_{CT_{lk}} \mathcal{K}'}{(CT_{ld}, CT_{lk}), \mathcal{K}, \Gamma \rightarrow (CT_{ld}, CT_{lk}), \mathcal{K}', \Gamma}$$

R-LOAD

$$\begin{array}{c}(CT_{ld}, CT_{lk}), \mathcal{K}, \Gamma \\ \rightarrow \\ (CT_{ld}[\mathtt{C} : \mathtt{L}], CT_{lk}), \mathcal{K}, (\Gamma; \mathtt{C} : \mathit{loadvers}(\mathtt{C}))\end{array}$$

R-LINK

$$\frac{\begin{array}{c}\mathtt{C} \notin \mathit{dom}(CT_{lk}) \qquad CT_{ld}(\mathtt{C}) = \mathtt{L} \qquad \Gamma \vdash_W \mathtt{C} : \sigma \text{ given } CT_{ld} \\ \Gamma' = \mathit{close}(\mathit{reify}(\Gamma, \sigma)) \qquad \Gamma' \text{ is } CT_{ld} \text{ consistent}\end{array}}{(CT_{ld}, CT_{lk}), \mathcal{K}, \Gamma \rightarrow (CT_{ld}, CT_{lk}[\mathtt{C} : \mathtt{L}]), \mathcal{K}, \Gamma'}$$

Figure 16. Semantics of Type Safe Dynamic Linking

type constraint system [22] (modulo security effects), extended in a straightforward manner to accommodate stack types.

LEMMA 4.3. *C is CT satisfiable iff $\mathit{close}(C)$ is CT consistent for all C and CT.*

4.4 Reification and Type Safe Dynamic Linking

Now we are ready to put the pieces together for type safe dynamic linking. It remains to define reification, to define the operational semantics of dynamic linking to incorporate our bytecode verification technique, and to frame the metatheory correctly to obtain a type safety result in the presence of dynamic linking.

We begin by specifying a type inference relation for loaded and linked class table pairs (CT_{ld}, CT_{lk}) that is the implementation analogue of Definition 3.4 as observed in the lemma following the definition. In this definition and afterwards, we assume that in any pair (CT_{ld}, CT_{lk}) that $CT_{lk} \subseteq CT_{ld}$.

DEFINITION 4.4. *The relation $\Gamma \vdash_W (CT_{ld}, CT_{lk})$ holds iff Γ is closed and CT_{ld} consistent, and for all $\mathtt{C} \in \mathit{dom}(CT_{lk})$ there exists a CT_{ld} derivable judgement $\Gamma \vdash_W \mathtt{C} : \sigma$ such that $\Gamma(\mathtt{C}) =_{set} \mathit{close}(\sigma)$, and for all $\mathtt{C} \in \mathit{dom}(CT_{ld}) - \mathit{dom}(CT_{lk})$ it is the case that $\Gamma(CT_{ld}) = \mathit{loadvers}(\mathtt{C})$.*

LEMMA 4.4. *If $\Gamma \vdash_W (CT_{ld}, CT_{lk})$ then Γ is CT_{ld} valid for CT_{lk}.*

The type inference relation just defined applies to a given pair of loaded and linked class tables, but in the presence of dynamic linking and loading these tables will evolve as computation proceeds. Since type inference accompanies linking, and old types may need to be reified and new types may depend on existing types, a type environment must also be maintained during computation. Hence, states in our full machine model are of the form $(CT_{ld}, CT_{lk}), \mathcal{K}, \Gamma$, where Γ maintains load version typings of loaded classes and fully verified typings of linked classes. The goal of our type implementation is to maintain the relation that Γ is CT_{ld} sound for CT_{lk} throughout computation. This is accomplished through composition of type inference, reification, closure, and consistency checks, which are all performed at link time.

The full operational semantics with type safe dynamic linking is defined in Fig. 16. Evaluation based on the linked class table is allowed via the R-EVAL rule. The R-LOAD rule specifies dynamic loading, which includes an extension of the environment Γ with a load version typing of the loaded class. Type verification is performed at link time, as specified in the R-LINK rule: after inferring the type of the newly linked class $\mathtt{C}$, the load version typings of $\mathtt{C}$ in existing typings in Γ are reified, the reified environment is closed, and its consistency checked. Reification has been defined at an intuitive level above. We now define it formally:

DEFINITION 4.5. *Constraint* reification *for type schemes is defined in Fig. 15, where $\mathit{labvars}_{\mathtt{C}}(D, \mathtt{T}) = \{\mathtt{t}^{\mathtt{C}} \mid \mathtt{t}^{\mathtt{C}} \in \mathit{fv}(D, \mathtt{T})\}$. Furthermore, we define $\mathit{reify}(\Gamma, \sigma) = \Gamma'$ iff $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma)$ and for all $\mathtt{C} \in \mathit{dom}(\Gamma)$ it is the case that $\mathit{reify}(\Gamma(\mathtt{C}), \sigma) = \Gamma'(\mathtt{C})$.*

In the metatheory, we need to establish soundness of reification. That is, we need to show that reified types are inferrable. The lemma follows by inversion of the type inference rules, and induction on instruction type derivations:

LEMMA 4.5. *Given $\Gamma(\mathtt{C}) = \mathit{loadvers}(\mathtt{C})$ and $\Gamma \vdash_W \mathtt{C} : \sigma$ derivable given CT. Then if $\Gamma \vdash_W \mathtt{D} : \sigma'$ is derivable given CT, so is $\mathit{reify}(\Gamma, \sigma) \vdash_W \mathtt{D} : \mathit{reify}(\sigma', \sigma)$.*

We also need to show that "staged" closure in the presence of dynamic linking and reification is the same as whole program closure.

LEMMA 4.6. *If $\Gamma \vdash_W \mathtt{C} : \sigma$ is CT derivable then so is the judgement $\mathit{close}(\Gamma) \vdash_W \mathtt{C} : \sigma'$ with $\mathit{close}(\sigma) =_{set} \mathit{close}(\sigma')$.*

Now, we can show a crucial technical result, which establishes that type verification in the R-LINK rule preserves soundness of class table typings.

LEMMA 4.7. *Given:*

$$\Gamma(\mathtt{C}) = \mathit{loadvers}(\mathtt{C}) \qquad\qquad \Gamma \vdash_W (CT_{ld}, CT_{lk})$$

$$\Gamma \vdash_W \mathtt{C} : \sigma \textit{ given } CT_{ld} \qquad\qquad \Gamma' = \mathit{close}(\mathit{reify}(\Gamma, \sigma))$$

$$\Gamma' \textit{ is } CT_{ld} \textit{ consistent}$$

Then $\Gamma' \vdash_W (CT_{ld}, CT_{lk}[\mathtt{C} : \mathtt{L}])$.

Proof (Sketch). By assumption Γ' is closed and consistent, and by the definition of reify it is easy to see that for all $\mathtt{D} \in \mathit{dom}(CT_{ld}) - \mathit{dom}(CT_{lk}[\mathtt{C} : \mathtt{L}])$ it is the case that $\mathit{reify}(\mathit{loadvers}(\mathtt{D}), \sigma) = \mathit{loadvers}(\mathtt{D})$, since $\mathit{loadvers}(\mathtt{D})$ will not contain any variable marked by $\mathtt{C}$. Assuming $\mathtt{D} \in \mathit{dom}(CT_{lk}[\mathtt{C} : \mathtt{L}])$, it remains to be shown that there exists a CT_{ld} derivable judgement $\Gamma' \vdash_W \mathtt{D} : \sigma'$ such that $\Gamma'(\mathtt{D})n =_{set} \mathit{close}(\sigma')$, which follows by assumption, Definition 4.4, Lemma 4.5, and Lemma 4.6. □

On the basis of these results and link time type safety (Theorem 3.1), we are now able to demonstrate that our link time bytecode verification technique enforces access control in JVM$_{\text{sec}}$, writing $\rightarrow^{\star}$ to denote the reflexive, transitive closure of $\rightarrow$.

THEOREM 4.2 (Link Time Enforcement of Checks). *If the relation $\Gamma \vdash_W (CT_{ld}, CT_{lk})$ holds and:*

$$\begin{array}{c}(CT_{ld}, CT_{lk}), \mathcal{K}_{\iota}, \Gamma \rightarrow^{\star} \\ (CT'_{ld}, CT'_{lk}), \langle R, \varsigma, M^{\pi}\{\mathtt{check(r)} \cdot \mathtt{B}\}, D, H\rangle, \Gamma'\end{array}$$

then $\mathtt{r} \in R$.

Proof. By induction on the length of the computation and case analysis on reduction steps, the crucial case being R-LINK where

Lemma 4.7 applies, we have that $\Gamma' \vdash_W (CT'_{ld}, CT'_{lk})$. But it is easy to show that $\mathcal{K}_\iota \rightarrow^{\star}_{CT'_{lk}} \langle R, \varsigma, M^\pi\{\mathtt{check(r)} \cdot \mathtt{B}\}, D, H\rangle$, so the result follows by Lemma 4.4 and Theorem 3.1. □

4.5 Examples and Discussion

We now return to the running examples last visited in Sect. 3.3, to illustrate the main ideas behind reification. For clarity, we present typings in term form not actually generated by type inference, that generates typings in constraint form. Assume that classes Bar and Baz are in the linked class table CT_{lk}, and have been assigned typings as in Sect. 3.3. Assume also that class Foo is in the loaded class table CT_{ld}, but has not yet been linked, so that it has been assigned a load version typing $\forall \mathtt{t}^{\mathtt{Foo}}[\mathbf{true}].[\mathtt{t}^{\mathtt{Foo}}\,\mathtt{Foo}]$. Recalling the code snippets (1) and (2) in Sect. 3.3, observe that each snippet will generate a distinct Foo instance. Type instantiation due to the I-New inference rule will assign distinct types $[\mathtt{t}_1^{\mathtt{Foo}}\,\mathtt{Foo}]$ and $[\mathtt{t}_2^{\mathtt{Foo}}\,\mathtt{Foo}]$ to these distinct instances. Additionally, the I-Invk rule will impose these types as lower bounds of the types of the objects whose m methods are invoked on Bar and Baz objects in snippets (1) and (2) respectively. When Foo is eventually linked, reification, and closure will fill in these type instances with lower bounds defined by class type inference on Foo and instantiation, obtaining constraints of the following form, where T is as defined in Sect. 3.3; variables $\mathtt{v}_{(1)}$ and $\mathtt{v}_{(2)}$ are chosen for the purposes of the example, though any fresh variables would be canonically correct:

$$[(\mathtt{m} : \mathtt{T}[\mathtt{v}_{(1)}/\mathtt{v}])\ \mathtt{Foo}] <: [\mathtt{t}_1^{\mathtt{Foo}}\,\mathtt{Foo}]$$

$$[(\mathtt{m} : \mathtt{T}[\mathtt{v}_{(2)}/\mathtt{v}])\ \mathtt{Foo}] <: [\mathtt{t}_2^{\mathtt{Foo}}\,\mathtt{Foo}]$$

Now, because $[\mathtt{t}_1^{\mathtt{Foo}}\,\mathtt{Foo}]$ and $[\mathtt{t}_2^{\mathtt{Foo}}\,\mathtt{Foo}]$ form lower bounds of the invoked object, therefore types $\mathtt{T}[\mathtt{v}_{(1)}/\mathtt{v}]$ and $\mathtt{T}[\mathtt{v}_{(1)}/\mathtt{v}]$ form lower bounds on the m invocation types by transitivity, and the types of Bar and Baz lower bound the argument types of these by contravariance of function domain subtyping, thus closure will yield the following constraints, assuming the types of Bar and Baz in Sect. 3.3 are inferred and instantiated trivially:

$$\mathtt{r}_1 : \mathrm{Pre}; \mathtt{v}_3 <: \mathtt{v}_{(1)} \qquad\qquad \mathtt{v}_4 <: \mathtt{v}_{(2)}$$

In short, reification yields the same accurate typings obtainable statically as in Sect. 3.3, via the mechanism of polymorphism.

5. Conclusion: Related and Future Work

Related Work The work most related to ours is Higuchi et al.'s type system for JVM access control [13, 14], that we have discussed throughout the paper. Their work underlies and inspires ours. The type system we present here is also based on a previous one that uses polymorphism to achieve flexible typings in a Featherweight Java (FJ) model [21, 22] called FJ$_{\mathrm{sec}}$. However FJ$_{\mathrm{sec}}$ is at the sourcecode level, not the bytecode level, its execution model does not incorporate dynamic loading and linking, and although FJ$_{\mathrm{sec}}$ does employ an access control mechanism, it is more general, more computationally expensive to verify, and not an existing component of the JVM.

Our use of polymorphic row types for type enforcement of stack inspection is also inspired by previous type systems for stack inspection, in a much simpler high level language model based on the λ-calculus [20]. A variety of bytecode-level type theories have been developed for enforcing type safety in the JVM [18, 12, 23], that have focused on a rich set of type safety issues other than access control.

The formalism we've developed for dynamic loading and linking in the JVM is based on previous formalisms [8, 5], especially work by Jensen et al. [15], though a number of previous authors have studied the problem of formalizing the JVM to prove type safety properties such as safe initialization of objects [7], and class loader safety [17]. However, these works have not considered type verification of stack inspection. Previous type systems for more flexible dynamic linking in the JVM are related in their technical approach, such as the use of polymorphism for expressive typing of linked code [3, 2]. Other related foundational work in type theory for languages with dynamic linking are in the same general vein as ours [11], especially previous approaches to so-called incremental typing for web applications that exploit dynamic linking [10].

Future Work A number of technical issues for continued study exist. In particular, the language model studied in this paper lacks many features of JVM bytecode. Object downcasting can be easily addressed by adapting the "soft subtyping" relation developed in previous work [22], but side effecting features such as state, exception handling, and threading remain a significant issue. In this richer setting, a soft typing approach probably needs to be adapted, especially if parameterized privileges are considered as in [14].

Our type theory as it currently exists could also be refined, for example while we have proven type inference sound for the logical type specification, we have not demonstrated completeness. Efficiency of the analysis could be improved, by modifying aspects of the algorithm we have kept simple for purposes of this presentation. For example, in the I-Class type inference rules, when a class is typed every one of its methods is, including those that are inherited from superclasses. This is to ensure sound typing of method overrides and methods employing self-reference, but if an inherited method is detected as not being self-referential its typing can be inherited as well. In the same vein, we have maintained a closure for whole environments, but for a sound (though less eager) analysis only a closure of class Main needs to be maintained.

At a higher level, empirical testing is an appropriate next step to ensure that our proposed extension has tractable behavior in practice. Furthermore, a comprehensive survey of common design patterns should be made, to ensure that our system is sufficiently flexible to accommodate them.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[2] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Even more principal typings for Java-like languages. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*, 2004.

[3] A. Buckley and S. Drossopoulou. Flexible dynamic linking. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*, 2004.

[4] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM Press, 1991.

[5] Sophia Drossopoulou. An abstract model of Java dynamic linking and loading. In *Workshop on Types in Compilation (TIC)*, Lecture Notes in Computer Science, pages 53–84, London, UK, 2001. Springer-Verlag.

[6] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1995.

[7] Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[8] Allen Goldberg. A specification of Java loading and bytecode verification. In *CCS '98: Proceedings of the 5th ACM conference*

on Computer and communications security, pages 49–58, New York, NY, USA, 1998. ACM Press.

[9] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, December 1997.

[10] Paul Graunke, Robert Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In *European Symposium on Programming (ESOP)*, 2003.

[11] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In *Workshop on Types in Compilation (TIC)*, 2001.

[12] Tomoyuki Higuchi and Atsushi Ohori. Java bytecode as a typed term calculus. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 201–211, New York, NY, USA, 2002. ACM Press.

[13] Tomoyuki Higuchi and Atsushi Ohori. A static type system for JVM access control. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 227–237. ACM Press, 2003.

[14] Tomoyuki Higuchi and Atsushi Ohori. A static type system for JVM access control. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

[15] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalization. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, Washington, DC, USA, 1998. IEEE Computer Society.

[16] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[17] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 36–44, 1998.

[18] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–78, New York, NY, USA, 1999. ACM Press.

[19] François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

[20] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, May 2005.

[21] Christian Skalka. Trace effects and object orientation. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 139–150, New York, NY, USA, 2005. ACM Press.

[22] Christian Skalka. Types and trace effects for object orientation. *Journal of Higher Order and Symbolic Computation*, 2006. Accepted for publication pending revision.

[23] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 149–160, New York, NY, USA, 1998. ACM Press.

[24] D. S. Wallach and E. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998.