Secure Information Flow and Pointer Confinement in a Java-like Language

Anindya Banerjee*
Computing and Information Sciences
Kansas State University
Manhattan KS 66506 USA
ab@cis.ksu.edu

David A. Naumann†
Computer Science
Stevens Institute of Technology
Hoboken NJ 07030 USA
naumann@cs.stevens-tech.edu

Abstract

We consider a sequential object-oriented language with pointers and mutable state, private fields and class-based visibility, dynamic binding and inheritance, recursive classes, casts and type tests, and recursive methods. Programs are annotated with security levels, constrained by security typing rules. A noninterference theorem shows how the rules ensure pointer confinement and secure information flow.

1. Introduction

There are many channels by which sensitive information can be leaked. This paper is concerned with information flows that arise in sequential object-oriented programs due to control flow, data flow, and dynamic memory allocation. Inspired by Denning's work [10, 11], Volpano, Smith and Irvine devised an elegant, syntax-directed type system for annotating program variables, commands, and procedure parameters with security levels [34, 32]. Goguen and Meseguer [13] introduced noninterference, expressed in terms of suitable simulation relations, to formalize information flow policies. Volpano and Smith proved that their type system ensures noninterference [34, 32].

Subsequently, several researchers have given similar analyses for possibilistic and probabilistic noninterference for multi-threaded programs [28, 33, 25, 19, 27]. Barthe and Serpette prove noninterference for a purely functional instance-based object calculus [4]. For sequential programs, Abadi *et al.* [1], Sabelfeld and Sands [26] and Heintze and Riecke [15] consider higher order procedures. They also make explicit the connections between the relational formulation of noninterference and other dependency analyses such as slicing and binding time analysis [1, 4], building noninterference properties into the semantics in the manner of Reynolds' relationally parametric models [24]. However, it is difficult to extend such models in a tractable way to encompass language features such as recursive types and shared mutable objects which are extensively used in languages such as Java [2].

Our contribution is to deal with dynamic memory allocation and object-oriented constructs: we prove noninterference for a sequential object-oriented language with pointers and mutable state, private fields and class-based visibility, dynamic binding and inheritance, casts and type tests, and mutually recursive classes and methods. The security type system extends that of Volpano and Smith [32] to encompass data flow via mutable object fields and control flow in dynamically dispatched method calls.

Myers [20] gave a security typing system for a fragment of Java even richer than ours, but left open the problem of justifying the rules with a noninterference result. This is hardly surprising, as the rules are quite complicated. Some of the complications are inherent in the complexity of the language; others are introduced with the aim of accomodating dynamic access control and sophisticated security policies including declassification [12, 21, 20, 35].

In the present paper, we confine attention to the problem of proving noninterference for a "realistic" sequential language (not far from JavaCard [7]), using conventional annotations without declassification or dynamic access control. Our results are given in elementary terms. We eschew the elegant structures used in [15, 1, 26], but we can give detailed proofs in the space of a few pages. This may also help in extending our results to other language features.

We use the weak form of noninterference which does not consider termination to be observable. Strong noninterference is treated by Volpano and Smith [32] (and others), but for a sequential language this requires loop guards to have low security. We omit loops but include recursion which also admits nontermination. Thus to extend their ideas to our language we would require low security guards for conditionals that involve recursive calls, a complication

*Supported by NSF grants EIA-9806835 and CCR-0296182
†Supported by NSF grant INT-9813854

we choose to avoid here. The extension of strong noninterference to recursive procedures merits study in a simple setting before it is combined with the features of an object-oriented language.

Our work grew out of a study of data abstraction for Java [3]. We found that a straightforward compositional semantics is adequate even in the presence of recursive types and dynamically bound method calls (which are typically viewed as being akin to higher order procedures). The semantics is simple enough to extend easily to additional constructs, e.g., access control is included in [3].

Due to pointer aliasing, the language is not relationally parametric *per se*. But suitable confinement of pointers suffices to yield a strong representation-independence result for user-defined abstractions [3]. The term "confinement" appears to originate in the literature on operating system security, but its use is natural in object-oriented programming where pointer confinement has been proposed for encapsulation at the level of modules, classes, or instances [16, 8, 18, 5, 31]. For information flow, we impose a confinement condition on high-security pointers.

The following section is a detailed overview of the paper. Section 3 formalizes the language and its semantics. Section 4 gives the security typing system. Section 5 deals with confinement, which is then used in Section 6 to prove the noninterference theorem. Section 7 considers related work and prospects for further advance.

2. Overview

We consider a language that is quite complicated relative to those for which noninterference results have been proved before, but we use simple security annotations that generalize those of Volpano and Smith. We annotate local variables, fields, and method parameters using types (T, κ) where T is an ordinary program type and κ is one of the two security levels `H` and `L`. Generalization to a lattice of levels would complicate notations without adding illumination. We also annotate classes. The security level of a class is the security level of `this`, i.e., the target of a method call, and it is also used for confinement as described in the sequel.

In the following example there is a single field `f` of high security (level `H`), in a class named `C` of level `L`.

```
class C L extends D {
  (bool,H) f;

  (bool,H) m ((bool,L) x) L {
     if (x) this.f := not x;
     else   this.f := x;
     return (x == this.f);  } }
```

Our typing system assigns to method `m` the type `x:(bool,L)`$\xrightarrow{\texttt{L}}$`(bool,H)`. This designates a method which takes a parameter of level `L`, has effect `L` on the heap, and returns a value of level `H`. The heap effect `L` is given in the method declaration, following the parameter list `((bool,L) x)`.

The type of the result expression `(x == this.f)` is `(bool,H)` because the type of `this.f` is `(bool,H)`. Although both `this` and `x` are `L`, field `f` is is declared `H`.

In typing judgements, Volpano and Smith use a type `H cmd` for commands that assign only to `H` variables. Our typing judgements use a command type $(\texttt{com}\ \kappa_1, \kappa_2)$ for commands that assign only to variables of level at least κ_1 and to object fields of level at least κ_2. That is, the heap effect designated by a command type is concerned with field levels. The same is true of the heap effect designated above the arrow in a method type like `x:(bool,L)`$\xrightarrow{\texttt{L}}$`(bool,H)`.

If we modify the example to declare a heap effect `H` for the method, resulting in a type `x:(bool,L)`$\xrightarrow{\texttt{H}}$`(bool,H)`, the class is still typable because the only effect is on a field of level `H`.

Consider the following variation.

```
class C H extends D {
  (bool,??) f;
  (bool,H) m ((bool,L) x) L {
     if (x) this.f := not x
     else   this.f := x;
     return (x == this.f);  } }
```

The type of `m` is again `x:(bool,L)`$\xrightarrow{\texttt{L}}$`(bool,H)` but the class is `H`. What level, marked `??`, can be used for field `f`? The conditional statement needs to be typable in context `x:(bool,L),this:(C,H)` because the label on a class designates the level of `this`. The conditional should be given type `(com ?,L)`, as the heap effect for the method is `L`. What about the effect, marked `?`, on the local environment (i.e., local variables and parameters)? Our rule for field update requires that the level of the field be at least the level of the assigned variable: as `this` is `H`, field `f` needs to be declared as `H` and then the conditional can be typed as `(com H,L)`.

Subclassing in Java is "invariant" in the sense that method signatures cannot be specialized in subclasses. Although other alternatives merit study, our typing system is the same: the declared security levels of a method cannot be changed in subclasses. However, we allow the subclass of an `L` class to be declared `H`. This has interesting consequences and it is one of the reasons why we need a semantic notion of pointer confinement, called L-confinement, as discussed later.

These examples are far from an exhaustive illustration of the interesting patterns that arise. Nor can a brief exposition

give a thorough justification for the rules. A few more examples appear in Section 4.1. Although our type system is quite general, it disallows some sensible programs such as those involving declassification. On the other hand, it admits some declarations that are sound but not very useful, such as variables of type `(C,L)` for `H`-class `C`.

In Section 3 we give the formal syntax and denotational semantics of the underlying language for which we consider security annotations. Java is sufficiently complicated that it is a challenge to formalize its syntax in a readable way. We adapt the formalization of a smaller fragment of Java, FJ [17], which mixes standard notations from type theory with Java-like syntax. We extend FJ by adding imperative features, and modify it by treating fields as private (class-visible) rather than public. The semantics is defined in terms of an ordinary typing system; it does not depend on security annotations.

In Section 4 we give the typing system for security annotations. In practice, one would want to specify security policy by labelling certain inputs and outputs, leaving the rest to automated inference, but that is beyond the scope of the paper, as is label polymorphism. For expository purposes, it is convenient to use separate subsumption and subtyping rules that express, for example, that an `L` expression can always be used in a context where an `H` expression is allowed. But for proofs it is more convenient to use syntax-directed rules that incorporate subsumption. For lack of space we give only the syntax-directed rules for our system, at the cost that the rules have rather many constraints on security levels. Readers unfamiliar with this style of specifying a flow analysis are encouraged to read Volpano and Smith's clear and succinct presentation [32].

Our main result, in Section 6, is a noninterference theorem: For a program with annotations satisfying the rules, fields, variables, and method results with label `L` do not depend on those with label `H`. As in work cited in the introduction, noninterference is formalized in terms of an equivalence relation on states. Here a state consists of an environment η and a heap h. For two states (η, h) and (η', h') to be equivalent, written $(\eta, h) \sim (\eta', h')$, means that they agree on `L`-variables and on `L`-fields of heap objects. The noninterference theorem says that if a typable program is executed in the two related states (η, h) and (η', h'), the outcomes are also related. What this means is that changes to high security inputs cannot be observed at low outputs.

The main noninterference result pertains to the methods of a complete program consisting of a collection of classes that can involve mutually recursive fields and methods. The semantics of such a program is given as a fixpoint and the main result is proved by induction. This proof depends on a main lemma saying that commands are noninterfering—*safe*, for short—under the assumption that the methods they call are safe. Safety depends on confinement.

Section 5 is concerned with confinement. In [32], noninterference is proved on the basis of two lemmas called simple security and confinement. Simple security corresponds roughly to our L-confinement and safety Lemmas 5.1 and 6.1 for expressions. These results say that an `L`-expression cannot distinguish between related states. Confinement in [32] corresponds to what we call H-confinement, which pertains to `H`-commands: such commands do not assign to `L`-variables or fields. Noninterference for conditionals depends on H-confinement: A conditional with `H`-guard can behave differently in equivalent states, so the statement parts must be restricted to be `H`-commands.

To deal with the heap, we use a property called L-confinement. The language includes subclassing and subsumption, and we allow an `L`-class to have an `H`-subclass. As a result, simple typing does not prevent certain `H` to `L` flows. Moreover, an `L`-object can be aliased by both an `L`- and an `H`-variable. Hence, we must show that a typable `L`-expression never denotes an `H`-pointer and that commands preserve the invariant that `L`-variables and `L`-fields contain no `H`-pointers.

Besides controlling direct information flows, L-confinement is also needed to treat dynamic memory allocation and the equivalence $\sim$ for heaps. An environment maps variable identifiers to values, whereas a heap maps locations to values. One need only define $\eta \sim \eta'$ for η and η' with the same domain. In a simple imperative language, an `H`-command can distinguish between $\sim$-related states but the distinction appears in terms of the states of variables (and termination). In a language with heap allocation, there is also the possibility of differing allocations. Thus it is unreasonable for $h \sim h'$ to require identical domains. But allocation must depend on the domain of the heap, to ensure freshness. If $h \sim h'$ allows different domains then an `L`-command could potentially distinguish by observing the behavior of the allocator.

In a language with pointer arithmetic, comparison using $<$ is a real problem. In Java, pointers are abstract: they can be compared only by $=$. Although two related heaps could result in different allocations, one can argue that no information is leaked to `L`-commands because they cannot make useful distinctions between pointer values. One could formalize this idea by requiring that, for the `L` part of related heaps, the domains need not be equal but rather there should be a bijection between them so that corresponding objects are related. From related states, allocation of a new `L`-object would add a pair to the bijection. Although this approach appears viable, we have avoided it for two reasons. First, the added complication would pervade all definitions and results. We prefer to follow the lead of Volpano and Smith in using simple standard semantic notions to the extent possible. Second, although it is easy to see how to maintain the bijection in a small step semantics, it is not as simple in a

compositional semantics. Some other possibilities for relating heaps are in [29, 30], though we have not pursued these ideas.

Our approach depends on the allocator satisfying a mild parametricity condition which is also needed for the abstraction theorem of [3]. The condition says that the choice of a fresh location for an object of class C depends only on currently-allocated C objects. Capability-based systems provide similar abstractions. The benefit here is that the definition of $h \sim h'$ can simply require that h and h' have the same domain for L-objects. We have not imposed a condition that the heaps are garbage-free. Garbage in the initial state has no influence on the final state. One might have thought that we would need to garbage-collect in order for the final states to be related, but this is not necessary.

3. Language: syntax and semantics

This section presents the language without security annotations; it is this language for which the semantics is defined. The language is the same as the core language in [3], from which we have also borrowed expository material. To make the paper more self-contained, we give complete typing rules and semantic definitions. For further explanation and justification of the definitions, the reader is advised to consult [3].

The grammar is based on given sets of class names (with typical element C), field names (f), method names (m), and variable/parameter names x (including *this*). Barred identifiers like $\overline{T}$ indicate finite lists, e.g., $\overline{T}\ \overline{f}$ stands for a list $\overline{f}$ of field names with corresponding types $\overline{T}$.

$$
\begin{array}{rcl}
T & ::= & \texttt{bool} \mid \texttt{unit} \mid C \\
CL & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\
M & ::= & T\ m(\overline{T}\ \overline{x})\ \{S;\ \texttt{return}\ e\} \\
S & ::= & x := e \mid x.f := e \mid x := \texttt{new}\ C() \mid e.m(\overline{e}) \mid \\
 & & \texttt{if}\ e\ S\ \texttt{else}\ S \mid \texttt{var}\ T\ x := e\ \texttt{in}\ S \mid S;\ S \\
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \\
 & & e{=}{=}e \mid (C)\ e \mid \texttt{null} \mid e\ \texttt{instanceof}\ C
\end{array}
$$

Without formalizing it precisely, we assume there is a class `Object` with no fields or methods which can be used as a superclass. Additional base types, such as integers, can be treated in the same way as `bool` and `unit`.

Table 1 gives the typing rules. A typing environment Γ is a finite function from variable names to types. A judgement of the form $\Gamma;\ C \vdash e : T$ says that e has type T in the context of a method of class C, with parameters and local variables declared by Γ. A judgement $\Gamma;\ C \vdash S : \texttt{com}$ says that S is a command in the same context. To simplify the typing rules and semantic definitions, we assume that variable and parameter names are not re-used.

A complete program is given as a *class table* CT that associates each declared class name with its declaration. The typing rules make use of auxiliary notions that are defined in terms of CT, so the typing relation $\vdash$ depends on CT but this is elided in the notation. Because typing of each class is done in the context of the full table, methods can be mutually recursive, and so can field types.

Methods and classes are considered public. The rules for field access and update enforce visibility: fields of a class are accessible only to methods of that class, as in Java's `private` visibility.

Subsumption is built in to the rules using the subtyping relation $\leq$ on T specified as follows. For base types, $\texttt{bool} \leq \texttt{bool}$ and $\texttt{unit} \leq \texttt{unit}$. For classes C and D, we have $C \leq D$ iff either $C = D$ or the class declaration for C is $\texttt{class}\ C\ \texttt{extends}\ B\ \{\ \ldots\ \}$ for some $B \leq D$.

To define some auxiliary notations, let

$$CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}_1\ \overline{f};\ \overline{M}\ \}$$

and let M be in the list $\overline{M}$ of method declarations, with $M = T\ m(\overline{T}_2\ \overline{x})\ \{S;\ \texttt{return}\ e\}$. We record the typing information by defining $\mathit{mtype}(m, C) = (\overline{x} : \overline{T}_2) \to T$. For the declared fields, we define $\mathit{dfields}\, C = \overline{T}_1\ \overline{f}$ and $\mathit{type}(\overline{f}, C) = \overline{T}_1$. To include inherited fields, we define $\mathit{fields}\, C = \mathit{dfields}\, C \cup \mathit{fields}\, D$, and assume $\overline{f}$ is disjoint from the names in $\mathit{fields}\, D$. The built-in class `Object` has no methods or fields. Note that $\mathit{mtype}(m, C)$ is defined only if m is declared or inherited in C.

A class table is well formed if each of its method declarations is well formed according to the following rule.

$$\frac{\begin{array}{l}(\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash S : \texttt{com} \\ (\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash e : U \qquad U \leq T \\ \mathit{mtype}(m, D) \text{ is undefined or equals } (\overline{x} : \overline{T}) \to T\end{array}}{C\ \texttt{extends}\ D \vdash T\ m(\overline{T}\ \overline{x})\{S;\ \texttt{return}\ e\}}$$

Turning to semantics, the state of a method in execution is comprised of a *heap* h, which is a finite partial function from locations to object states, and an *environment* η, which assigns locations and primitive values to local variables and parameters. Every environment of interest includes the distinguished variable *this* which points to the target object. A command denotes a function from initial state to either a final state or the error value $\perp$.

For locations, we assume that a countable set Loc is given, along with a distinguished entity nil not in Loc. We treat object states as mappings from field names to values. To track the object's class we assume given a function $\mathit{loctype} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{loctype}\ \ell = C$. We write $\mathit{locs}\ C$ for $\{\ell \mid \mathit{loctype}\, \ell = C\}$. The assumption about $\mathit{loctype}$ ensures an adequate supply of fresh locations, given that the domain of any heap is finite.

$$\Gamma;\ C \vdash x : \Gamma x \qquad \Gamma;\ C \vdash \mathtt{null} : B \qquad \frac{\mathit{mtype}(m, D) = (\overline{x} : \overline{T}) \to T \quad \Gamma;\ C \vdash e : D \quad \Gamma;\ C \vdash \overline{e} : \overline{U} \quad \overline{U} \le \overline{T}}{\Gamma;\ C \vdash e.m(\overline{e}) : T}$$

$$\frac{\Gamma;\ C \vdash e_1 : T \quad \Gamma;\ C \vdash e_2 : T}{\Gamma;\ C \vdash e_1 == e_2 : \mathtt{bool}} \qquad \frac{T f \in \mathit{dfields}\, C \quad \Gamma;\ C \vdash e : C}{\Gamma;\ C \vdash e.f : T}$$

$$\frac{\Gamma;\ C \vdash e : D \quad B \le D}{\Gamma;\ C \vdash (B)\, e : B} \qquad \frac{\Gamma;\ C \vdash e : D \quad B \le D}{\Gamma;\ C \vdash e\ \mathtt{instanceof}\ B : \mathtt{bool}} \qquad \Gamma;\ C \vdash \mathtt{unit} : \mathtt{unit}$$

$$\frac{x \ne \mathit{this} \quad \Gamma;\ C \vdash e : T \quad T \le \Gamma x}{\Gamma;\ C \vdash x := e : \mathtt{com}} \qquad \frac{\Gamma x = C \quad T f \in \mathit{dfields}\, C \quad \Gamma;\ C \vdash e : U \quad U \le T}{\Gamma;\ C \vdash x.f := e : \mathtt{com}} \qquad \frac{x \ne this \quad B \le \Gamma x}{\Gamma;\ C \vdash x := \mathtt{new}\ B(\,) : \mathtt{com}}$$

$$\frac{\mathit{mtype}(m, D) = (\overline{x} : \overline{T}) \to T \quad \Gamma;\ C \vdash e : D \quad \Gamma;\ C \vdash \overline{e} : \overline{U} \quad \overline{U} \le \overline{T}}{\Gamma;\ C \vdash e.m(\overline{e}) : \mathtt{com}} \qquad \frac{\Gamma;\ C \vdash e : \mathtt{bool} \quad \Gamma;\ C \vdash S_1 : \mathtt{com} \quad \Gamma;\ C \vdash S_2 : \mathtt{com}}{\Gamma;\ C \vdash \mathtt{if}\ e\ S_1\ \mathtt{else}\ S_2 : \mathtt{com}}$$

$$\frac{\Gamma;\ C \vdash S_1 : \mathtt{com} \quad \Gamma;\ C \vdash S_2 : \mathtt{com}}{\Gamma;\ C \vdash S_1;\ S_2 : \mathtt{com}} \qquad \frac{\Gamma;\ C \vdash e : U \quad (\Gamma, x : T);\ C \vdash S : \mathtt{com} \quad U \le T}{\Gamma;\ C \vdash \mathtt{var}\ T\ x := e\ \mathtt{in}\ S : \mathtt{com}}$$

Table 1. Typing rules for expressions and commands.

$$\begin{array}{lcl}
[\![\mathtt{bool}]\!] = \{\mathsf{true}, \mathsf{false}\} \qquad [\![\mathtt{unit}]\!] = \{\bullet\} \qquad [\![C]\!] = \{\mathsf{nil}\} \cup \{\ell \mid \ell \in \mathit{Loc} \wedge \mathit{loctype}\ \ell \le C\} \\
\eta \in [\![\Gamma]\!] & \Leftrightarrow & \mathit{dom}\,\eta = \mathit{dom}\,\Gamma \wedge \forall x \in \mathit{dom}\,\eta\ .\ \eta\, x \in [\![\Gamma\, x]\!] \\
s \in [\![C\ \mathit{state}]\!] & \Leftrightarrow & \mathit{dom}\, s = \mathit{fields}\, C \wedge \forall f \in \mathit{fields}\, C\ .\ s f \in [\![\mathit{type}(f, C)]\!] \\
h \in [\![\mathit{Heap}]\!] & \Leftrightarrow & \mathit{dom}\, h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \forall \ell \in \mathit{dom}\, h\ .\ h\ell \in [\![(\mathit{loctype}\ \ell)\ \mathit{state}]\!] \\
[\![C, (\overline{x} : \overline{T}) \to T]\!] & = & [\![\overline{x} : \overline{T}, \mathit{this} : C]\!] \to [\![\mathit{Heap}]\!] \to ([\![T]\!] \times [\![\mathit{Heap}]\!])_\perp \\
[\![\mathit{MEnv}]\!] & \subseteq & (C : \mathit{ClassNames}) \rightharpoonup (m : \mathit{MethodNames}) \rightharpoonup [\![C, \mathit{mtype}(m, C)]\!]
\end{array}$$

Table 2. Semantic domains.

Methods are associated with classes, in a *method environment*, rather than with instances. For this reason the semantic domains, given in Table 2, are rather simple. There are no recursive domain equations to be solved. In addition to domains like $[\![T]\!]$ and $[\![\Gamma]\!]$ that correspond directly to syntactic notations, we use the following domains: $[\![\mathit{Heap}]\!]$ is the set of heaps, $[\![C\ \mathit{state}]\!]$ is the set of states of objects of class C, $[\![\mathit{MEnv}]\!]$ is the set of method environments (we write $\rightharpoonup$ for finite partial functions), and $[\![(C, (\overline{x} : \overline{T}) \to T)]\!]$ is the set of meanings for methods of class C with result T and parameters $\overline{x} : \overline{T}$.

The sets $[\![\mathit{Heap}]\!]$, $[\![\mathtt{bool}]\!]$, $[\![C]\!]$, and $[\![C\ \mathit{state}]\!]$ are ordered by equality. We write $\to$ for continuous function space, ordered pointwise, and $X_\perp$ for domain X with added bottom element $\perp$. Each set $[\![(C, (\overline{x} : \overline{T}) \to T)]\!]$ has a least element (the constantly-$\perp$ function) and least upper bounds of ascending chains, and this suffices for the fixpoint semantics. Without giving a precise formalization, we require that $[\![\mathit{MEnv}]\!]$ contain exactly the partial functions μ such that $\mu C m$ is defined for all classes C declared in CT and methods m declared or inherited in C.

The semantics is defined for an arbitrary allocator, but the noninterference theorem depends on parametricity.

Definition 1 (Allocator, parametric)
An *allocator* is a location-valued function fresh such that $\mathit{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$, for all C, h. An allocator is *parametric* if $\mathit{dom}\, h_1 \cap \mathit{locs}\, C = \mathit{dom}\, h_2 \cap \mathit{locs}\, C$ implies $\mathit{fresh}(C, h_1) = \mathit{fresh}(C, h_2)$. □

For example, if $\mathit{Loc} = \mathbb{N}$ the function $\mathit{fresh}(C, h) =$

$min\{\ell \mid \mathit{loctype}\,\ell = C \wedge \ell \notin \mathit{dom}\,h\}$ is parametric.

It is straightforward to show that, as in Java, no program constructs create dangling pointers, but it is slightly simpler to formulate the definitions to allow dangling pointers. Like cast failures, dereferences of dangling pointers and nil are considered an error. We identify all errors, and divergence, with the improper value $\perp$.

The semantics is defined by induction on typing judgements, and for all typings for e and S we have

$$[\![\Gamma;\ C \vdash e : T]\!] \in [\![\mathit{MEnv}]\!] \to [\![\Gamma]\!] \to [\![\mathit{Heap}]\!] \to [\![T]\!]_\perp$$
$$[\![\Gamma;\ C \vdash S : \mathtt{com}]\!] \in [\![\mathit{MEnv}]\!] \to [\![\Gamma]\!] \to [\![\mathit{Heap}]\!] \to ([\![\Gamma]\!] \times [\![\mathit{Heap}]\!])_\perp$$

To streamline the treatment of $\perp$ in the semantic definitions, we use a metalanguage construct, **let** $d = E_1$ **in** E_2, with the following meaning: If the value of E_1 is $\perp$ then that is the value of the entire let expression; otherwise, its value is the value of E_2 with d bound to the value of E_1. This construct is only exploited in the semantic definitions; later, in definitions of properties, we handle $\perp$ explicitly.

Function update is written, e.g., $[\eta \mid x \mapsto d]$. In the semantics of local variables, we write $\downharpoonright$ for domain restriction: if x is in the domain of function η then $\eta \downharpoonright x$ is the function like η but without x in its domain.

Table 3 gives the semantics of expressions and Table 4 gives the semantics of commands. The definitions are straightforward renderings of the operational semantics [2]. For example, the value of $e.f$ in state (η, h) is $\perp$ if the value of e is $\perp$ or is not in $\mathit{dom}\,h$; otherwise, the value of e is some location $\ell \in \mathit{dom}\,h$, so the object state $h\ell$ is a finite map with $f \in \mathit{dom}\,(h\ell)$ and the value of $e.f$ is $h\ell f$. Field update $x.f := e$ in state (η, h) does not change the environment; the new heap $[h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$ updates h by replacing $h\ell$ with the object state $[h\ell \mid f \mapsto d]$ obtained by updating field f to have the value d of e.

For method call as an expression, $e.m(\overline{e})$, the value is $\perp$ unless the value of e is some $\ell \in \mathit{dom}\,h$. In that case, let d be the method meaning given by μ for method m at the dynamic type $(\mathit{loctype}\,\ell)$ of e. The result of the call is obtained by applying d to the initial heap h and to the environment $[\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]$ where $\overline{d}$ is the list of values of arguments $\overline{e}$. The result, if not $\perp$, is a pair (d_0, h_0); for method call as expression, the value of $e.m(\overline{e})$ is d_0. Note that h_0 is discarded; for expository simplicity we do not model side effects of expressions (see Section 7 for a discussion). For method call as command, d_0 is discarded and the new state is η, h_0, as the call has no effect on the environment η of the caller.

The semantics of a class table is the method environment, $\hat{\mu}$, given as the least upper bound of the ascending chain $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ defined as follows.

$$\mu_0\, C\, m = \lambda\eta.\ \lambda h.\ \perp$$
$$\mu_{j+1}\, C\, m = [\![M]\!]\mu_j \quad \text{if } m \text{ is declared as } M \text{ in } C$$
$$\mu_{j+1}\, C\, m = \mu_{j+1}\, B\, m \quad \text{if } m \text{ is inherited from } B \text{ in } C$$
$$[\![M]\!]\mu\eta h$$
$$= \textbf{let } (\eta_0, h_0) = [\![(\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash S : \mathtt{com}]\!]\mu\eta h \textbf{ in}$$
$$\quad \textbf{let } d = [\![(\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash e : T]\!]\mu\eta_0 h_0 \textbf{ in } (d, h_0)$$

where in class C we have $M = T\ m(\overline{T}\ \overline{x})\{S; \mathtt{return}\ e\}$.

$$[\![\Gamma;\ C \vdash x : T]\!]\mu\eta h = \eta x$$
$$[\![\Gamma;\ C \vdash \mathtt{null} : B]\!]\mu\eta h = \mathsf{nil}$$
$$[\![\Gamma;\ C \vdash \mathtt{unit} : \mathtt{unit}]\!]\mu\eta h = \bullet$$
$$[\![\Gamma;\ C \vdash e_1 == e_2 : \mathtt{bool}]\!]\mu\eta h$$
$$= \textbf{let } d_1 = [\![\Gamma;\ C \vdash e_1 : T]\!]\mu\eta h \textbf{ in}$$
$$\quad \textbf{let } d_2 = [\![\Gamma;\ C \vdash e_2 : T]\!]\mu\eta h \textbf{ in } (d_1 = d_2)$$
$$[\![\Gamma;\ C \vdash e.f : T]\!]\mu\eta h$$
$$= \textbf{let } \ell = [\![\Gamma;\ C \vdash e : C]\!]\mu\eta h \textbf{ in}$$
$$\quad \textbf{if } \ell \notin \mathit{dom}\,h \textbf{ then } \perp \textbf{ else } h\ell f$$
$$[\![\Gamma;\ C \vdash e.m(\overline{e}) : T]\!]\mu\eta h$$
$$= \textbf{let } \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h \textbf{ in}$$
$$\quad \textbf{if } \ell \notin \mathit{dom}\,h \textbf{ then } \perp \textbf{ else}$$
$$\quad \textbf{let } (\overline{x} : \overline{T}) \to T = \mathit{mtype}(m, D) \textbf{ in}$$
$$\quad \textbf{let } d = \mu(\mathit{loctype}\ \ell)m \textbf{ in}$$
$$\quad \textbf{let } \overline{d} = [\![\Gamma;\ C \vdash \overline{e} : \overline{U}]\!]\mu\eta h \textbf{ in}$$
$$\quad \textbf{let } (d_0, h_0) = d[\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]h \textbf{ in } d_0$$
$$[\![\Gamma;\ C \vdash (B)\ e : B]\!]\mu\eta h$$
$$= \textbf{let } \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h \textbf{ in}$$
$$\quad \textbf{if } \ell \in \mathit{dom}\,h \wedge \mathit{loctype}\ \ell \leq B \textbf{ then } \ell \textbf{ else } \perp$$
$$[\![\Gamma;\ C \vdash e\ \mathtt{instanceof}\ B : \mathtt{bool}]\!]\mu\eta h$$
$$= \textbf{let } \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h \textbf{ in}$$
$$\quad \ell \in \mathit{dom}\,h \wedge \mathit{loctype}\ \ell \leq B$$

Table 3. Semantics of expressions.

4. Security typing

In this section we annotate the syntax of Section 3 with security labels. Where a type T could occur, i.e., in declarations of fields, parameters, and local variables, we use pairs (T, κ) where κ is a security level. Such a pair, written τ, is called a *security type*. The security levels are L and H, ordered $L \leq H$. We write $\kappa_1 \sqcup \kappa_2$ to denote least upper

$$
\begin{array}{l}
[\![\Gamma;\ C \vdash x := e : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ d = [\![\Gamma;\ C \vdash e : T]\!]\mu\eta h\ \textbf{in}\ ([\eta \mid x \mapsto d], h) \\
[\![\Gamma;\ C \vdash x.f := e : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ \ell = \eta x\ \textbf{in if}\ \ell \notin \mathit{dom}\, h\ \textbf{then}\ \bot\ \textbf{else} \\
\quad \textbf{let}\ d = [\![\Gamma; C \vdash e : U]\!]\mu\eta h\ \textbf{in}\ (\eta, [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]) \\
[\![\Gamma;\ C \vdash x := \mathtt{new}\ B(\,) : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ \ell = \mathit{fresh}(B, h)\ \textbf{in} \\
\quad ([\eta \mid x \mapsto \ell], [h \mid \ell \mapsto [\textit{fields}\, B \mapsto \mathit{defaults}]]) \\
[\![\Gamma;\ C \vdash e.m(\overline{e}) : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h\ \textbf{in} \\
\quad \textbf{if}\ \ell \notin \mathit{dom}\, h\ \textbf{then}\ \bot\ \textbf{else} \\
\quad \textbf{let}\ (\overline{x} : \overline{T}) \to T = \textit{mtype}(m, D)\ \textbf{in} \\
\quad \textbf{let}\ d = \mu(\mathit{loctype}\ \ell)m\ \textbf{in} \\
\quad \textbf{let}\ \overline{d} = [\![\Gamma;\ C \vdash \overline{e} : \overline{U}]\!]\mu\eta h\ \textbf{in} \\
\quad \textbf{let}\ (d_0, h_0) = d[\overline{x} \mapsto \overline{d}, \textit{this} \mapsto \ell]h\ \textbf{in}\ (\eta, h_0) \\
[\![\Gamma;\ C \vdash S_1;\ S_2 : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ (\eta_0, h_0) = [\![\Gamma;\ C \vdash S_1 : \mathtt{com}]\!]\mu\eta h\ \textbf{in} \\
\quad [\![\Gamma;\ C \vdash S_2 : \mathtt{com}]\!]\mu\eta_0 h_0 \\
[\![\Gamma;\ C \vdash \mathtt{if}\ e\ S_1\ \mathtt{else}\ S_2 : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ b = [\![\Gamma;\ C \vdash e : \mathtt{bool}]\!]\mu\eta h\ \textbf{in} \\
\quad \textbf{if}\ b\ \textbf{then}\ [\![\Gamma;\ C \vdash S_1 : \mathtt{com}]\!]\mu\eta h \\
\quad \textbf{else}\ [\![\Gamma;\ C \vdash S_2 : \mathtt{com}]\!]\mu\eta h \\
[\![\Gamma;\ C \vdash \mathtt{var}\ T\ x := e\ \mathtt{in}\ S : \mathtt{com}]\!]\mu\eta h \\
=\ \ \textbf{let}\ d = [\![\Gamma;\ C \vdash e : U]\!]\mu\eta h\ \textbf{in} \\
\quad \textbf{let}\ (\eta_0, h_0) = [\![(\Gamma, x : T);\ C \vdash S]\!]\mu[\eta \mid x \mapsto d]h\ \textbf{in} \\
\quad (\eta_0 \lfloor x, h_0)
\end{array}
$$

Table 4. Semantics of commands.

bound. The grammar is revised as follows.

$$
\begin{array}{rcl}
\kappa & ::= & L \mid H \\
\tau & ::= & (T, \kappa) \\
CL & ::= & \mathtt{class}\ C\ \kappa\ \mathtt{extends}\ C\ \{\ \overline{\tau}\ \overline{f};\ \overline{M}\ \} \\
M & ::= & \tau\ m(\overline{\tau}\ \overline{x})\ \kappa\ \{S;\ \mathtt{return}\ e\} \\
S & ::= & \ldots \mid \mathtt{var}\ \tau\ x := e\ \mathtt{in}\ S \mid \ldots
\end{array}
$$

Tables 5 and 6 give typing rules for annotated programs. We write Δ for typing environments that assign security types. A judgement $\Delta;\ C \vdash S : (\mathtt{com}\ \kappa_1, \kappa_2)$ says that S assigns only to variables (locals and parameters) of level $\geq \kappa_1$ and to object fields of level $\geq \kappa_2$ (see Lemma 5.4).

In Section 2 we noted that security typing can be presented more perspicuously using separate subtyping and subsumption rules. For example, our rule for $x := e$ in Table 6 could give the command the type $(\mathtt{com}\ \kappa_1, H)$, rather than including the unconstrained κ_4 and the constraint $\kappa_3 \leq \kappa_1$, as there is no heap effect and the most precise environment information is κ_1. Even for a syntax-directed system, it may be possible to avoid subsumptions for primitive commands, given those built into rules for command constructs like sequencing. But care needs to be taken because levels in a method declaration cannot be changed in overriding declarations, and primitive commands can occur as method bodies. We err on the side of generality.

The rules use versions of the auxiliary functions *mtype* etc. that take security levels into account. Let

$$CT(C) = \mathtt{class}\ C\ \kappa_1\ \mathtt{extends}\ D\ \{\ \overline{\tau_1}\ \overline{f};\ \overline{M}\ \}$$

and let M be in the list $\overline{M}$ of method declarations, with

$$M = \tau_1\ m(\overline{\tau_2}\ \overline{x})\ \kappa_2\ \{S;\ \mathtt{return}\ e\}$$

The security version of *mtype* is defined by $\textit{smtype}(m, C) = (\overline{x} : \overline{\tau_2}) \xrightarrow{\kappa_2} \tau_1$. Corresponding to *dfields*, *fields* and *type*, we define *sdfields*, *sfields* and *stype* which differ only in that they give security types, e.g., $\textit{sdfields}\, C = \overline{\tau_1}\ \overline{f}$.

We also need a function *level* that gives the level associated with the class itself: for the declaration above, $\textit{level}\, C = \kappa_1$. Define $\textit{level}\, \mathtt{Object} = L$. For locations, define $\textit{level}\, \ell = \textit{level}(\mathit{loctype}\, \ell)$.

The rule for method declaration imposes the condition that an overriding definition cannot change the parameter or return types, nor the heap effect. The rule for class declaration restricts inheritance of methods. This is discussed in Section 4.1 below.

We use the symbol † to erase annotations: $(T, \kappa)^\dagger = T$, and this extends to erasure for typing environments, commands, and method declarations in the obvious way. For example, if M is the method declaration

$$(\mathtt{bool}, L)\ m((\mathtt{bool}, L)\ x)\ L\ \{x := x;\ \mathtt{return}\ x\}$$

then $M^\dagger$ is $\mathtt{bool}\ m(\mathtt{bool}\ x)\ \{x := x;\ \mathtt{return}\ x\}$.

For any judgement $\Delta;\ C \vdash S : (\mathtt{com}\ \kappa_1, \kappa_2)$ derivable using the rules in Tables 5 and 6, the erased judgement $\Delta^\dagger;\ C \vdash S^\dagger : \mathtt{com}$ is derivable using the rules of Table 1. Conversely, any program typable using the rules of Table 1 can be annotated everywhere by L and typed by the rules in Tables 5 and 6.

4.1. Examples

We consider a program in which the HIV status of a medical patient is to be kept confidential. A basic patient record looks as follows. We assume that there is an `L`-class `String`.

$$\Delta;\ C \vdash x : \Delta x \qquad \Delta;\ C \vdash \texttt{null} : (D, \kappa) \qquad \Delta;\ C \vdash \texttt{unit} : (\texttt{unit}, \kappa)$$

$$\frac{\Delta;\ C \vdash e_1 : (T, \kappa_1) \quad \Delta;\ C \vdash e_2 : (T, \kappa_2)}{\Delta;\ C \vdash e_1 == e_2 : (\texttt{bool}, \kappa_1 \sqcup \kappa_2)} \qquad \frac{(T, \kappa_1) f \in \mathit{sdfields}\, C \quad \Delta;\ C \vdash e : (C, \kappa_2)}{\Delta;\ C \vdash e.f : (T, \kappa_1 \sqcup \kappa_2)} \qquad \frac{\Delta;\ C \vdash e : (D, \kappa) \quad B \leq D}{\Delta;\ C \vdash (B)\ e : (B, \kappa)}$$

$$\frac{\begin{array}{l}\mathit{smtype}(m, D) = \overline{x} : (\overline{T}, \overline{\kappa}) \xrightarrow{\kappa_3} (T, \kappa_2) \\ \Delta;\ C \vdash e : (D, \kappa_4) \quad \Delta;\ C \vdash \overline{e} : (\overline{U}, \overline{\kappa_5}) \\ \overline{U} \leq \overline{T} \quad \overline{\kappa_5} \leq \overline{\kappa}\end{array}}{\Delta;\ C \vdash e.m(\overline{e}) : (T, \kappa_2 \sqcup \kappa_4)} \qquad \frac{\Delta;\ C \vdash e : (D, \kappa) \quad B \leq D}{\Delta;\ C \vdash e\ \texttt{instanceof}\ B : (\texttt{bool}, \kappa)}$$

Table 5. Security typing rules for expressions.

```
class LPatient L extends Object {
  (String,L) name;
  (String,L) getName() H {
    return this.name     }
  (unit,L) setName((String,L) n) L {
    this.name := n                  } }
```

Note that `setName` has `L`-effect, as it assigns to an `L`-field. Method `getName` can be declared as having `L` or `H` effect; we choose the latter. A confidential field is added for HIV status in the following class.

```
class XPatient L extends LPatient {
  (String,H) hiv;
  (unit,L) setHIV((String,H) s) H {
    this.hiv := s                  }
  (String,H) getHIV() H {
    return this.hiv     } }
```

Here is an application using such objects, written using slight abuses of the official syntax.

```
class Main L extends Object {
  (unit,L) main() L {
    var (String,L) Lbuf := null;
    var (String,H) Hbuf := null;
    var (LPatient,L) lp:=readFile(...);
    var (XPatient,L) xp:=new XPatient();
    Lbuf := lp.getName();
    Hbuf := lp.getName();
(1) xp.setName(Lbuf);
    Hbuf := readFromTrustedChan(...);
(2) xp.setHIV(Hbuf); } }
```

After execution of line (1), `LBuf`, `HBuf`, and `xp.name` all reference the same object. After execution of line (2), `LBuf` aliases `xp.name`, and `HBuf` aliases `xp.hiv`. All the code above is allowed by the security typing rules, assuming appropriate types for the `read` methods. The following are not allowed in `main`.

```
(3) Lbuf := Hbuf;
(4) lp.setName(xp.getHiv());
```

In both (3) and (4), there is a direct flow from `H` to `L`.

Let us consider an `H`-subclass of `XPatient`.

```
class HPatient H extends XPatient {
  (String,H) f;
  (unit,L) setf((String,H) v) H {
    this.f := v                  }
  (String,H) getf() H {
    return this.f     } }
```

By contrast with `setName`, the effect of `setf` can be declared `H` as it assigns only to an `H`-field.

There is, however, a problem with inherited methods. Suppose `LPatient` declared a method

```
  (String,L) passSelf() H {
    ...o.m(this)...       }
```

where `o.m` is a call on some object `o` of a method `m` with `L` parameter. This is typable in `LPatient`, although it would not be typable in `HPatient` where the level of `this` is `H`. For a variable `hp` of class `HPatient`, an invocation `hp.passSelf()` of the inherited method would result in a flow from `H` to `L` in the call `o.m(this)`.

We solve the problem in a simple way: The rule for classes in Table 6 requires that an `H`-subclass of an `L`-class overrides all methods of its superclass. This is unnecessarily, and unacceptably, restrictive. It requires, for example, that `getName` be overridden although it poses no risk of bad flow; and an overriding declaration does not have access to the private field `name`. The problem can be solved in a less restrictive way using the notion of "anonymous method" of Vitek and Bokowski [31]: such methods do not leak the receiver *this*. The static analyses described in [31, 14] restrict inheritance only for methods that leak *this*. We expect that their constraints can be adapted easily to our setting.

$$\frac{x \neq \mathit{this} \quad T_2 \leq T_1 \quad \kappa_2 \leq \kappa_1 \quad \kappa_3 \leq \kappa_1 \quad \Delta, x:(T_1,\kappa_1);\ C \vdash e:(T_2,\kappa_2)}{\Delta, x:(T_1,\kappa_1);\ C \vdash x := e : (\mathtt{com}\ \kappa_3, \kappa_4)}$$

$$\frac{(T,\kappa_2) f \in \mathit{sdfields}\, C \quad \Delta, x:(C,\kappa_1);\ C \vdash e:(U,\kappa_3) \quad U \leq T \quad \kappa_1 \sqcup \kappa_3 \sqcup \kappa_5 \leq \kappa_2}{\Delta, x:(C,\kappa_1);\ C \vdash x.f := e : (\mathtt{com}\ \kappa_4, \kappa_5)}$$

$$\frac{x \neq \mathit{this} \quad B \leq D \quad \mathit{level}\, B \sqcup \kappa_2 \leq \kappa_1 \quad \kappa_3 \leq \mathit{level}\, B}{\Delta, x:(D,\kappa_1);\ C \vdash x := \mathtt{new}\ B(\,) : (\mathtt{com}\ \kappa_2, \kappa_3)}$$

$$\frac{\mathit{smtype}(m,D) = \overline{x}:(\overline{T},\overline{\kappa}) \xrightarrow{\kappa_3} (T,\kappa_2) \quad \Delta;\ C \vdash e:(D,\kappa_4) \quad \Delta;\ C \vdash \overline{e}:(\overline{U},\overline{\kappa_5}) \quad \overline{U} \leq \overline{T} \quad \overline{\kappa_5} \leq \overline{\kappa} \quad \kappa_4 \sqcup \kappa_7 \leq \kappa_3}{\Delta;\ C \vdash e.m(\overline{e}) : (\mathtt{com}\ \kappa_6, \kappa_7)}$$

$$\frac{\Delta;\ C \vdash S_1 : (\mathtt{com}\ \kappa_1, \kappa_2) \quad \Delta;\ C \vdash S_2 : (\mathtt{com}\ \kappa_3, \kappa_4) \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta;\ C \vdash S_1;\ S_2 : (\mathtt{com}\ \kappa_5, \kappa_6)}$$

$$\frac{\Delta;\ C \vdash e : (\mathtt{bool}, \kappa_5) \quad \Delta;\ C \vdash S_1 : (\mathtt{com}\ \kappa_1, \kappa_3) \quad \Delta;\ C \vdash S_2 : (\mathtt{com}\ \kappa_2, \kappa_4) \quad \kappa_5 \leq \kappa_6 \sqcap \kappa_7 \quad \kappa_6 \leq \kappa_1 \sqcap \kappa_2 \quad \kappa_7 \leq \kappa_3 \sqcap \kappa_4}{\Delta;\ C \vdash \mathtt{if}\ e\ S_1\ \mathtt{else}\ S_2 : (\mathtt{com}\ \kappa_6, \kappa_7)}$$

$$\frac{\Delta;\ C \vdash e:(U,\kappa_4) \quad \Delta, x:(T,\kappa_1);\ C \vdash S : (\mathtt{com}\ \kappa_5, \kappa_6) \quad U \leq T \quad \kappa_4 \leq \kappa_1 \quad \kappa_2 \leq \kappa_5 \quad \kappa_3 \leq \kappa_6}{\Delta;\ C \vdash \mathtt{var}\ (T,\kappa_1)\ x := e\ \mathtt{in}\ S : (\mathtt{com}\ \kappa_2, \kappa_3)}$$

$$\frac{\overline{x}:(\overline{T},\overline{\kappa}), \mathit{this}:(C,\kappa_1);\ C \vdash S : (\mathtt{com}\ \kappa_2, \kappa_3) \quad \overline{x}:(\overline{T},\overline{\kappa}), \mathit{this}:(C,\kappa_1);\ C \vdash e:(T,\kappa_4) \quad \mathit{smtype}(m,D) \text{ is undefined or equals } \overline{x}:(\overline{T},\overline{\kappa}) \xrightarrow{\kappa_3} (T,\kappa_4)}{C\ \kappa_1\ \mathtt{extends}\ D \vdash (T,\kappa_4)\ m((\overline{T},\overline{\kappa})\ \overline{x})\ \kappa_3\ \{S;\ \mathtt{return}\ e\}}$$

$$\frac{\mathit{level}\, D \leq \kappa \quad C\ \kappa\ \mathtt{extends}\ D \vdash M \ \text{ for each } M \in \overline{M} \quad \text{If } \mathit{level}\, D \neq \kappa \text{ then every } m \text{ with } \mathit{smtype}(m,D) \text{ defined is overridden in } C \text{ by some } M \in \overline{M}.}{\vdash \mathtt{class}\ C\ \kappa\ \mathtt{extends}\ D\ \{\ \overline{\tau}\ \overline{f};\ \overline{M}\ \}}$$

Table 6. Security typing rules for commands, method declarations, and classes.

Our last examples involve information leaks via the conditional control flow implicit in dynamically bound method calls. Consider these three classes.

```
class YN L extends Object {
  (bool,L) val(){ return true; } } }
class Y L extends YN {
  (bool,L) val(){ return true; } }
class N L extends YN {
  (bool,L) val(){ return false; } }
```

We add the following method to `XPatient`. The typing rules force the result to have level `H`; otherwise there would be a bad data flow.

```
  (YN,H) leak() H {
    var (YN,H) o;
    if (this.hiv) o := new Y();
    else          o := new N();
    return o;     }
```

In `main`, the expression `xp.leak().val()` has level `H` due to the return type of `leak`.

Here is a similar example but using the heap and method call commands rather than expressions.

```
class YNh L extends Object {
  (bool,H) v;
  (bool,H) val(){ return v } }
  (unit,L) setv((bool,H) w) H {
    this.v:=w }
  (unit,L) set() H {this.setv(true)} }
class Yh L extends YNh {
  (unit,L) set() H {this.setv(true)} }
class Nh L extends YNh {
  (unit,L) set() H {this.setv(false)} }
```

The `leak` method is the same as before but using `YNh` and its subclasses. Consider `x` of type `(YNh,H)` in

```
  x:=xp.leak(); x.set(); ...x.val()...
```

The declared level of `x` must be `H` because it is assigned from `leak`. The method call rule then requires for `x.set` that the heap effect of `set` be `H`, which in turn forces the level of field `v` to be `H`. Indeed, if its level was `L` then the call `x.set()` would violate noninterference.

4.2. Remarks about proofs

Proofs in the sequel involve detailed analysis of the semantics and the security typing rules. For each specific case, the semantic definition may involve several values (e.g., the value of e is needed in the semantics of $x := e$), and the rule may involve several types and security labels. In writing a given proof case, we found it convenient to write down both the rule and the semantics for reference. It is impractical to include such redundancy in the paper, however. Instead, when it comes to proving something about a particular construct we make free use of identifiers in the typing rule (in Table 5 or 6), for types and labels, and identifiers in the semantic definition for semantic values (in Table 3 or 4). We explicitly introduce identifiers for types or values only when necessity or perspicuity demands it.

Note that the semantic definition may use different identifiers for types, as the semantics is based on the typing rules in Table 1 rather than the security rules in Tables 5 and 6.

We streamline the proofs by ignoring $\bot$ outcomes in many cases. Most of the results only pertain to non-$\bot$ outcomes, and the constructs are mostly strict in $\bot$. Without comment we assume various intermediate values are non-$\bot$ unless confusion could result.

5. Confinement

This section shows that typable programs maintain the invariant that L fields and variables never hold H locations. The formalization uses the indistinguishability relation $\sim$ also used in the main results of Section 6.

In formalizing the absence of L-variables that refer to H-objects, we take advantage of the fact that nil $\notin Loc$ and $\bot \notin Loc$. We use the short name "ok" for L-confinement.

Definition 2 (L-confinement (ok))

- Define $LLoc = \{\ell \in Loc \mid \mathit{level}\,\ell = L\}$.
- For heaps, define $ok\,h$ iff for all $\ell \in dom\,h$ and every $f \in \mathit{fields}(\mathit{loctype}\,\ell)$, if $\mathit{stype}(f, \mathit{loctype}\,\ell) = (T, L)$ for some T and $h\ell f \in Loc$ then $h\ell f \in LLoc$.
- For environments, define $ok\,\Delta\,\eta$ iff for every x with $\Delta\,x = (T, L)$ for some T, if $\eta\,x \in Loc$ then $\eta\,x \in LLoc$.
- For method environments, define $ok\,\mu$ iff the following holds: for every m, C, η, h, if $ok\,h$, $ok\,\Delta\,\eta$, and $\mu C m \eta h \neq \bot$ then $ok\,h_0$ and $\kappa_3 = L \wedge d \in Loc \Rightarrow d \in LLoc$,
  where $\mathit{smtype}(m, C) = \overline{x} : (\overline{T}, \overline{\kappa}) \xrightarrow{\kappa_2} (T, \kappa_3)$
  $\Delta = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{this} : (C, \mathit{level}\,C)$
  $(d, h_0) = \mu C m \eta h$

Lemma 5.1 (L-confinement of expressions)
Let $\Delta;\ C \vdash e : (T, L)$ and let $d = [\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta h$. If $ok\,\mu$, $ok\,\Delta\,\eta$, and $ok\,h$ then $d \in Loc \Rightarrow d \in LLoc$.

Proof: By induction on the derivation of $\Delta;\ C \vdash e : (T, L)$ (for brevity: "induction on e"). Recall from Section 4.2 that throughout the proofs we ignore the $\bot$ cases.

- Case of x: Then $d = [\![\Delta^\dagger;\ C \vdash x : T]\!]\mu\eta h = \eta x$. The result follows directly from assumption $ok\ \Delta\ \eta$.
- $e.f$: By typing, $\kappa_1 = L = \kappa_2$. Because $\kappa_2 = L$ we can use induction on e; this, together with the assumption that the semantics is non-$\bot$, yields that there is $\ell = [\![\Delta^\dagger;\ C \vdash e : C]\!]\mu\eta h$ and $\ell \in LLoc$ and $\ell \in dom\ h$. Now the result follows using $\kappa_1 = L$ and assumption $ok\,h$.
- $e.m(\overline{e})$: Let $\Delta_0 = [\overline{x} : (\overline{T}, \overline{\kappa}), \mathit{this} : (\mathit{loctype}\,\ell, \mathit{level}\,\ell)]$, and $\eta_0 = [\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]$. We claim that $ok\,\Delta_0\,\eta_0$; then we get the result by $ok\ \mu$ because by typing $\kappa_2 = \kappa_4 = L$. It remains to prove the claim.

  If $\overline{\kappa} = L$, then by the typing rule, $\overline{\kappa_5} = L$. So by induction on $\overline{e}$, and since the semantics is non-$\bot$, we get $\overline{d} \in Loc \Rightarrow \overline{d} \in LLoc$. Hence $ok\,(\overline{x} : (\overline{T}, \overline{\kappa}))\,[\overline{x} \mapsto \overline{d}]$. We get $ok\,(\mathit{this} : (\mathit{loctype}\,\ell, \mathit{level}\,\ell))\,[\mathit{this} \mapsto \ell]$ directly from the definitions of ok and $LLoc$, as $\mathit{level}\,\ell = L$ iff $\ell \in LLoc$.
- $(B)\ e$: By $\kappa = L$, and because we are considering the case where $[\![(B)\ e]\!]\mu\eta h \neq \bot$, we can use induction on e to obtain $\ell \in Loc \Rightarrow \ell \in LLoc$. Moreover, ℓ must be in $dom\,h$ otherwise the semantics is $\bot$. Now the result follows directly.
- e `instanceof` B, e==e', `null`, `unit`: The result returned in each case is not in Loc. This falsifies the antecedent in the lemma. □

Lemma 5.2 (L-confinement of commands)
Let $\Delta;\ C \vdash S : (\mathtt{com}\,\kappa_1, \kappa_2)$. If $ok\,\mu$, $ok\,h$, $ok\,\Delta\,\eta$, and $[\![\Delta^\dagger;\ C \vdash S^\dagger : \mathtt{com}]\!]\mu\eta h \neq \bot$ then $ok\,\Delta\,\eta_0$ and $ok\,h_0$, where $(\eta_0, h_0) = [\![\Delta^\dagger;\ C \vdash S^\dagger : \mathtt{com}]\!]\mu\eta h$.

Proof: By induction on the derivation of $\Delta;\ C \vdash S : (\mathtt{com}\ \kappa_1, \kappa_2)$, using the assumptions of the Lemma.

- $x := e$: This has no heap effect: the result heap h_0 is h and $ok\,h$ holds by assumption. We only need to show $ok\,(\Delta, x : (T_1, \kappa_1))\,[\eta \mid x \mapsto d]$, where $d = [\![\Delta^\dagger;\ C \vdash e : T_2]\!]\mu\eta h$. Accordingly, assume that $\kappa_1 = L$. Then by typing, $\kappa_2 = L$ so Lemma 5.1 for e yields $d \in Loc \Rightarrow d \in LLoc$.
- $x.f := e$: The result environment η_0 is just η, and $ok\,\Delta\,\eta$ holds by assumption. We only need to show $ok\,h_0$,

where $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$, $\ell = \eta x$ and $d = [\![\Delta^\dagger;\ C \vdash e : U]\!]\mu\eta h$. Since the semantics is non-$\bot$, we have $\ell \in dom\, h$. Now assume that the level κ_2 of field f is L. Then by the typing rule we have $\kappa_1 = \kappa_3 = L$. Now applying Lemma 5.1 on e, we obtain, $d \in Loc \Rightarrow d \in LLoc$. This proves $ok\, h_0$.

- $x := \mathtt{new}\ B(\ )$: For the environment, we must show $ok\,(\Delta, x : (D_1, \kappa_1))[\eta \mid x \mapsto \ell]$, where $\ell = \mathit{fresh}(B, h)$. Since by assumption $ok\,\Delta\,\eta$, it is enough to deal with x; that is, if $\kappa_1 = L$ we must show $\ell \in Loc \Rightarrow \ell \in LLoc$. Indeed, the allocator *fresh* yields $\ell \in Loc$. But by the typing rule, $\kappa_1 = L$ implies $\mathit{level}\, B = L$, hence $\ell \in LLoc$ as required. Finally, we get $ok\, h_0$, where $h_0 = [h \mid \ell \mapsto [\mathit{fields}\, B \mapsto \mathit{defaults}]]$, because $ok\, h$ by hypothesis and *defaults* contains no locations (the defaults are false and nil).
- $e.m(\overline{e})$: We have $\eta_0 = \eta$, and $ok\,\Delta\,\eta$ by assumption, so it suffices to show $ok\, h_0$. Let $\Delta_0 = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{this} : (\mathit{loctype}\,\ell, \mathit{level}\,\ell)$, and $\eta_0 = [\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]$. We claim $ok\,\Delta_0\,\eta_0$; then we get the result by $ok\ \mu$. The claim is proved by the same argument as for method calls in the proof of Lemma 5.1 (literally the same argument, owing to the fact that the relevant identifiers are the same in the typing rules for method call as expression and as command).
- $\mathtt{if}\ e\ S_1\ \mathtt{else}\ S_2$: Let $b = [\![\Delta^\dagger;\ C \vdash e : \mathtt{bool}]\!]\mu\eta h$. Then if $b = \mathsf{true}$, the result follows by induction on S_1 and if $b = \mathsf{false}$, the result follows by induction on S_2.
- $\mathtt{var}\ (T, \kappa_1)\ x := e\ \mathtt{in}\ S$: First, we have $ok\ (\Delta, x : (T, \kappa_1))\ [\eta \mid x \mapsto d]$ where $d = [\![\Delta^\dagger;\ C \vdash e : U]\!]\mu\eta h$. This is because if $\kappa_1 = L$, then by typing $\kappa_4 = L$, so by Lemma 5.1 for e, $d \in Loc \Rightarrow d \in LLoc$. Induction on S yields $ok\,(\Delta, x : (T, \kappa_1))\ \eta_1$ and $ok\, h_0$, where $(\eta_1, h_0) = [\![(\Gamma, x : T);\ C \vdash S]\!]\mu[\eta \mid x \mapsto d]h$. Hence $ok\,\Delta\,(\eta_1 \lfloor x)$.
- $S_1;\ S_2$: Use induction on S_1, then on S_2. □

Lemma 5.3 (L-confinement of method environments)
For each i we have $ok\,\mu_i$, and $ok\,\hat{\mu}$. □

The proof is by induction on i, using Lemmas 5.1 and 5.2, and then fixpoint induction for $\hat{\mu}$. It follows the pattern of the proof of Theorem 6.3, and is given in the full paper.

Object states are indistinguishable by L if their L-fields are equal, and environments are indistinguishable if their L-variables are equal. In the case of heaps and object states, the relevant levels are determined by the field declarations in the class table. By contrast, the levels for environments are determined by parameter and local variable declarations, hence the dependence is explicit in the notation $\sim_\Delta$. It is straightforward to show that each of these is an equivalence relation.

Definition 3 (Indistinguishable by L)

- For $s, s' \in [\![C\ state]\!]$, define $s \sim s'$ iff $\forall f \in \mathit{fields}\, C$. **let** $(T, \kappa) = \mathit{stype}(f, C)$ **in** $(\kappa = L \Rightarrow sf = s'f)$.
- For $h, h' \in [\![Heap]\!]$, define $h \sim h'$ iff $dom\ h \cap LLoc = dom\ h' \cap LLoc$ and $\forall \ell \in dom\ h \cap LLoc$. $h\ell \sim h'\ell$.
- For $\eta, \eta' \in [\![\Delta^\dagger]\!]$, define $\eta \sim_\Delta \eta'$ iff $\forall x \in dom\ \Delta$. **let** $(T, \kappa) = \Delta x$ **in** $(\kappa = L \Rightarrow \eta x = \eta' x)$. □

If a command is typable as $(\mathtt{com}\ H, \kappa)$ it does not assign to L-variables, and if it is typable as $(\mathtt{com}\ \kappa_2, H)$ it does not assign to L-fields of objects.

Definition 4 (H-confined method environment)
Method environment μ is H-confined, written $\mathit{Hconf}\ \mu$, if $\mu C m \eta h \neq \bot \Rightarrow h_0 \sim h$, where $(d, h_0) = \mu C m \eta h$, for all C, m with $\mathit{smtype}(m, C) = \overline{x} : (\overline{T}, \overline{\kappa}) \xrightarrow{H} (T, \kappa)$. □

Lemma 5.4 (H-confinement of commands)
Let $\Delta;\ C \vdash S : (\mathtt{com}\ \kappa_1, \kappa_2)$. Then for all μ, η, h such that $\mathit{Hconf}\ \mu$ and $[\![\Delta^\dagger;\ C \vdash S^\dagger : \mathtt{com}]\!]\mu\eta h \neq \bot$ we have

- if $\kappa_1 = H$ and $(\eta_0, h_0) = [\![\Delta^\dagger;\ C \vdash S^\dagger : \mathtt{com}]\!]\mu\eta h$ then $\eta \sim_\Delta \eta_0$.
- if $\kappa_2 = H$ and $(\eta_0, h_0) = [\![\Delta^\dagger;\ C \vdash S^\dagger : \mathtt{com}]\!]\mu\eta h$ then $h \sim h_0$.

Proof: By induction on the derivation of $\Delta;\ C \vdash S : (\mathtt{com}\ \kappa_1, \kappa_2)$. As usual, we follow the conventions described in Section 4.2; in particular, level identifiers in the proof are those in the relevant rules, *not* κ_1, κ_2 as used in the statement of the Lemma.

- $x := e$: This has no effect on the heap. We need to show that $\kappa_3 = H \Rightarrow \eta \sim_{(\Delta, x:(T_1, \kappa_1))} \eta_0$, where $\eta_0 = [\eta \mid x \mapsto d]$ and $d = [\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta h$. Assuming $\kappa_3 = H$, we obtain $\kappa_1 = H$ by the typing rule. Now the result follows using definition $\sim_{(\Delta, x:(T_1, H))}$.
- $x.f := e$: This has no effect on the environment, so it suffices to show that $\kappa_5 = H \Rightarrow h \sim h_0$, where $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$, $\ell = \eta x$ and $d = [\![\Delta^\dagger;\ C \vdash e : U]\!]\mu\eta h$. If $\ell \notin LLoc$ then $h \sim h_0$ because the two heaps are identical on $LLoc$. If $\ell \in LLoc$ then we must consider the updated field $h\ell f$. Assuming $\kappa_5 = H$, the typing rule forces the level κ_2 of field f to be H; nothing else is updated, so $h \sim h_0$ holds by definition of $\sim$.

- $x := \texttt{new}\ B(\,)$: First, assume $\kappa_2 = H$. Then by typing, $\kappa_1 = H$. We must show $\eta \sim_{(\Delta, x:(D,\kappa_1))} \eta_0$ where $\eta_0 = [\eta \mid x \mapsto \ell]$ and $\ell = \mathit{fresh}(B, h)$. But this follows by definition of $\sim_{(\Delta, x:(D,\kappa_1))}$ since $\kappa_1 = H$.

  Next assume $\kappa_3 = H$. Then by typing, $\mathit{level}\, B = H$. Hence $\ell \notin \mathit{LLoc}$. We must show $h \sim h_0$, where $h_0 = [h \mid \ell \mapsto [\mathit{fields}\, B \mapsto \mathit{defaults}]]$. But this follows by definition of $\sim$ because h and h_0 are identical on LLoc.

- $e.m(\overline{e})$: This has no effect on the environment. For the heap, suppose $\kappa_7 = H$. Then we must show $h \sim h_0$, where $(d_0, h_0) = \mu(\mathit{loctype}\, \ell)m[\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]h$, and $\ell = [\![\Delta;\ C \vdash e : D]\!]\mu\eta h$ and $\overline{d} = [\![\Delta;\ C \vdash \overline{e} : \overline{U}]\!]\mu\eta h$. Because $\kappa_7 = H$, we have by the typing rule, $\kappa_3 = H$. Moreover, $\mathit{smtype}(m, (\mathit{loctype}\, \ell)) = \overline{x} : (\overline{T}, \overline{\kappa}) \xrightarrow{H} (T, \kappa_2)$. Hence, by assumption $\mathit{Hconf}\ \mu$, we get $h \sim h_0$.

- $S_1;\ S_2$: First assume $\kappa_5 = H$. Then by the typing rule, $\kappa_1 = H = \kappa_3$. By induction on S_1 we get $\eta \sim_\Delta \eta_1$ where we write (η_1, h_1) for the intermediate state. Then by induction on S_2, we get $\eta_1 \sim_\Delta \eta_0$ where $(\eta_0, h_0) = [\![\Delta^\dagger;\ C \vdash S_2 : \texttt{com}]\!]\mu\eta_1 h_1$. Hence $\eta \sim_\Delta \eta_0$ by transitivity.

  Next, assume $\kappa_6 = H$. Then $\kappa_2 = H = \kappa_4$. And, $h \sim h_0$ follows by induction on S_1 and S_2 and transitivity.

- $\texttt{if}\ e\ S_1\ \texttt{else}\ S_2$: First, assume $\kappa_6 = H$. Then by the typing rule, $\kappa_1 = H$ and $\kappa_2 = H$. Let $b = [\![\Delta^\dagger;\ C \vdash e : \texttt{bool}]\!]\mu\eta h$. Then if $b = \mathsf{true}$, the result follows by induction on S_1 and if $b = \mathsf{false}$, the result follows by induction on S_2.

  Next, assume $\kappa_7 = H$. Then by the typing rule, $\kappa_3 = H$ and $\kappa_4 = H$. Again, the result follows by induction on S_1 if $b = \mathsf{true}$ and by induction on S_2 if $b = \mathsf{false}$.

- $\texttt{var}\ (T, \kappa_1)\ x := e\ \texttt{in}\ S$: First, assume $\kappa_2 = H$. Then by the typing rule, $\kappa_5 = H$. Hence by induction on S, $[\eta \mid x \mapsto d] \sim_{(\Delta, x:(T,\kappa_1))} \eta_0$ where $(\eta_0, h_0) = [\![(\Delta^\dagger, x : T);\ C \vdash S : \texttt{com}]\!]\mu[\eta \mid x \mapsto d]h$ and $d = [\![\Delta^\dagger;\ C \vdash e : U]\!]\mu\eta h$. Hence $\eta \sim_\Delta (\eta_0 \lfloor x)$.

  Next, assume $\kappa_3 = H$. Then by the typing rule, $\kappa_6 = H$. And, by induction on S we get $h \sim h_0$. □

Note that a command $\texttt{var}\ (T, L)\ x := e\ \texttt{in}\ S$ can be typed as $\Delta;\ C \vdash S : (\texttt{com}\ H, \kappa_2)$ so the Lemma applies to such commands. But the typing rule ensures that no L-variable is assigned in S. Moreover, if $\kappa_2 = H$ then no L-fields are assigned. So x is of limited use.

Lemma 5.5 (H-confinement of method environments) For each i we have $\mathit{Hconf}\ \mu_i$, and $\mathit{Hconf}\ \hat{\mu}$. □

The proof is by induction on i, using Lemma 5.4, and then fixpoint induction. It follows the pattern of the proof of Theorem 6.3, and is given in the full paper.

6. Noninterference

A method meaning is safe, i.e., noninterfering, provided that, for terminating computations, L-indistinguishable initial heaps and environments lead to L-indistinguishable results.

Definition 5 (Safe method environment) We define $\mathit{safe}\ \mu$ iff for all C, m and all h, h', η, η' the following holds: If $\mathit{ok}\, h$, $\mathit{ok}\, h'$ $\mathit{ok}\, \Delta\, \eta$, and $\mathit{ok}\, \Delta\, \eta'$ then

$$h \sim h' \wedge \eta \sim_\Delta \eta' \wedge \mu C m \eta h \neq \bot \neq \mu C m \eta' h' \Rightarrow h_0 \sim h_0' \wedge (\kappa_3 = L \Rightarrow d = d')$$

where

$$\begin{array}{lcl} \mathit{smtype}(m, C) & = & \overline{x} : (\overline{T}, \overline{\kappa}) \xrightarrow{\kappa_2} (T, \kappa_3) \\ \Delta & = & \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{this} : (C, \mathit{level}\, C) \\ (d, h_0) & = & \mu C m \eta h \\ (d', h_0') & = & \mu C m \eta' h' \end{array}$$

□

Our main result is that the method environment denoted by a secure class table is safe. The proof uses the following two results which express noninterference for the expression and command constructs, respectively.

If an expression can be typed $\Delta;\ C \vdash e : (T, L)$ then its meaning is the same in two L-indistinguishable states, provided that it diverges in neither state.

Lemma 6.1 (Safe expressions) Suppose $\Delta;\ C \vdash e : (T, L)$. Suppose that $h \sim h'$, $\eta \sim_\Delta \eta'$, $\mathit{safe}\ \mu$, $\mathit{ok}\, \mu$, $\mathit{ok}\, h$, $\mathit{ok}\, h'$, $\mathit{ok}\, \Delta\, \eta$, $\mathit{ok}\, \Delta\, \eta'$, and $[\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta h \neq \bot \neq [\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta' h'$. Then $[\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta h = [\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta' h'$.

In this proof and subsequent ones, we extend the convention described in Section 4.2. When comparing semantics for a pair of states (η, h) and (η', h'), we use corresponding primes on identifiers in the semantic definitions. For example, the semantic definition of $[\![\Delta^\dagger;\ C \vdash x := e : \texttt{com}]\!]\mu\eta h$ involves value d denoted by e in state η, h, so we write d' for the corresponding value for $[\![\Delta^\dagger;\ C \vdash x := e : \texttt{com}]\!]\mu\eta' h'$.

Proof: By induction on the derivation of $\Delta;\ C \vdash e : (T, L)$. Using the assumptions of the Lemma, we show $[\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta h = [\![\Delta^\dagger;\ C \vdash e : T]\!]\mu\eta' h'$ by cases on e.

- x: Then $[\![\Delta^\dagger;\ C \vdash x : T]\!]\mu\eta h = \eta x$ and $[\![\Delta^\dagger;\ C \vdash x : T]\!]\mu\eta' h' = \eta' x$. By assumption $\eta \sim_\Delta \eta'$ and $\Delta\, x = (T, L)$ we have $\eta x = \eta' x$.

- $e_1 == e_2$: By typing, $\kappa_1 = L = \kappa_2$. Thus we can use induction on e_1 to obtain $[\![\Delta^\dagger;\ C \vdash e_1 : T]\!]\mu\eta h = [\![\Delta^\dagger;\ C \vdash e_1 : T]\!]\mu\eta' h'$ (we are considering the case where $[\![e_1 == e_2]\!]\mu\eta h \neq \bot$ and $[\![e_1 == e_2]\!]\mu\eta' h' \neq \bot$, so by semantics the value of e_1 is also non-$\bot$). Similarly, induction on e_2 yields $[\![\Delta^\dagger;\ C \vdash e_2 : T]\!]\mu\eta h = [\![\Delta^\dagger;\ C \vdash e_2 : T]\!]\mu\eta' h'$. The result follows directly.

- $e.f$: By typing, $\kappa_1 = L = \kappa_2$. Because $\kappa_2 = L$ we can use induction on e; this yields that there is ℓ with $[\![\Delta^\dagger;\ C \vdash e : C]\!]\mu\eta h = \ell = [\![\Delta^\dagger;\ C \vdash e : C]\!]\mu\eta' h'$, as we only consider the case that both semantics are non-$\bot$. For the same reason, ℓ is in the domain of both h and h'. By $\kappa_2 = L$ and Lemma 5.1 we have $\ell \in LLoc$ so, by assumption $h \sim h'$, we get $h\ell \sim h'\ell$; this implies $h\ell f = h'\ell f$ because field f has label $\kappa_1 = L$.

- $e.m(\overline{e})$: By the security typing rule we have $e : (D, L)$, so by induction $\ell = \ell'$. Let $\Delta_0 = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{this} : (\mathit{loctype}\,\ell, \mathit{level}\,\ell)$, $\eta_0 = [\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]$, and $\eta_0' = [\overline{x} \mapsto \overline{d'}, \mathit{this} \mapsto \ell']$. We claim that $\mathit{ok}\,\Delta_0\,\eta$ and $\mathit{ok}\,\Delta_0\,\eta'$ and $\eta_0 \sim_{\Delta_0} \eta_0'$. Then we get the result $d = d'$ by *safe* μ. It remains to prove the claims. We give the argument for the case that $\overline{x}$ is a single identifier, as the generalization is obvious but awkward to put into words.

  For $\eta_0 \sim_{\underline{\Delta_0}} \eta_0'$, note that since $\ell = \ell'$ it suffices to deal with $\overline{d}, \overline{d'}$ regardless of whether $\mathit{loctype}\,\ell = L$. If $\overline{\kappa} = L$ then we need $\overline{d} = \overline{d'}$. Now $\overline{\kappa} = L$ implies $\overline{\kappa_5} = L$ by the security typing rule, and then we get $\overline{d} = \overline{d'}$ by induction on $\overline{e}$; moreover Lemma 5.1 yields $\mathit{ok}\,(\overline{x} : (\overline{T}, \overline{\kappa}))\,[\overline{x} \mapsto \overline{d}]$. Thus $\mathit{ok}\,\Delta_0\,\eta$ because $\mathit{ok}\,(\mathit{this} : (\mathit{loctype}\,\ell, \mathit{level}\,\ell))\,[\mathit{this} \mapsto \ell]$ holds for any ℓ. We have $\mathit{ok}\,\Delta_0\,\eta'$ *mutatis mutandis*.

- $(B)\ e$: By $\kappa = L$, we can use induction on e to obtain $\ell = \ell'$. Moreover, as we are considering the case where $[\![(B)\ e]\!]\mu\eta h \neq \bot \neq [\![(B)e]\!]\mu\eta' h'$, we have that ℓ is in both $\mathit{dom}\,h$ and $\mathit{dom}\,h'$. The result follows directly.

- e `instanceof` B, `null`, `unit`: All are easy. □

Lemma 6.2 (Safe commands)
Suppose $\Delta;\ C \vdash S : (\texttt{com}\ \kappa_1, \kappa_2)$. Suppose also $\mathit{ok}\,\mu$, $\mathit{ok}\,h$, $\mathit{ok}\,h'$, $\mathit{ok}\,\Delta\,\eta$, $\mathit{ok}\,\Delta\,\eta'$, *safe* μ, $\mathit{Hconf}\,\mu$, $\eta \sim_\Delta \eta'$, $h \sim h'$, and $[\![\Delta^\dagger;\ C \vdash S^\dagger : \texttt{com}]\!]\mu\eta h \neq \bot \neq [\![\Delta^\dagger;\ C \vdash S^\dagger : \texttt{com}]\!]\mu\eta' h'$. Then $\eta_0 \sim_\Delta \eta_0'$ and $h_0 \sim h_0'$, where $(\eta_0, h_0) = [\![\Delta^\dagger;\ C \vdash S^\dagger : \texttt{com}]\!]\mu\eta h$ and $(\eta_0', h_0') = [\![\Delta^\dagger;\ C \vdash S^\dagger : \texttt{com}]\!]\mu\eta' h'$.

Proof: By induction on the derivation of $\Delta;\ C \vdash S : (\texttt{com}\ \kappa_1, \kappa_2)$. Under the assumptions of the Lemma, we show $[\![\Delta^\dagger;\ C \vdash S : \texttt{com}]\!]\mu\eta h = [\![\Delta^\dagger;\ C \vdash S : \texttt{com}]\!]\mu\eta' h'$ by cases on S.

- $x := e$: This has no effect on the heap; we only need to show $[\eta \mid x \mapsto d] \sim_{(\Delta, x:(T_1,\kappa_1))} [\eta' \mid x \mapsto d']$, where $d = [\![\Delta^\dagger;\ C \vdash e : T_2]\!]\mu\eta h$ and $d' = [\![\Delta^\dagger;\ C \vdash e : T_2]\!]\mu\eta' h'$. Under the assumption $\eta \sim_{(\Delta, x:(T_1,\kappa_1))} \eta'$, it remains to show that $\kappa_1 = L$ implies $d = d'$. If $\kappa_1 = L$ then $\kappa_2 = L$, by typing, and then Lemma 6.1 yields $d = d'$. (Use of Lemma 6.1 depends on the assumptions $\mathit{ok}\,h$ etc.)

- $x.f := e$: This has no effect on the environment, so we only need to show that the result heaps are related. Let $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$ and let $h_0' = [h' \mid \ell' \mapsto [h'\ell' \mid f \mapsto d']]$. We must show $h_0 \sim h_0'$. If the level κ_2 of f is H, then $h_0 \sim h$ (and $h_0' \sim h'$) because h_0 and h are identical except for the H field f. So the result follows by transitivity of $\sim$. For the other case, $\kappa_2 = L$, we have $\kappa_1 = \kappa_3 = L$ by the typing rule. Since $\eta \sim_{(\Delta, x:(C_1,\kappa_1))} \eta'$, we obtain $\ell = \eta x = \eta' x = \ell'$. So it remains to show that $h_0\ell f \sim h_0'\ell f$, i.e., $d = d'$. And this holds by Lemma 6.1 for e, using that $\kappa_3 = L$.

- $x := \texttt{new}\ B(\ )$: For the environment, we must show $[\eta \mid x \mapsto \ell] \sim_{\Delta, x:(D_1,\kappa_1)} [\eta' \mid x \mapsto \ell']$. By assumption $\eta \sim_{\Delta, x:(D_1,\kappa_1)} \eta'$ it is enough to deal with x; that is, if $\kappa_1 = L$ we need $\ell = \ell'$. By the typing rule, $\kappa_1 = L$ implies $\mathit{level}\,B = L$. Thus, by $h \sim h'$, we have $\mathit{dom}\,h \cap \mathit{locs}\,B = \mathit{dom}\,h' \cap \mathit{locs}\,B$; then $\ell = \ell'$ by parametricity of the allocator (Definition 1).

  Finally, we get $h_0 \sim h_0'$ as follows. If either ℓ or ℓ' is in $LLoc$ then $\mathit{level}\,B = L$ so by parametricity of the allocator we get $\ell = \ell'$, satisfying the domain condition for $h_0 \sim h_0'$. For the range, i.e., $h\,\ell \sim h'\,\ell$, the result holds because the new object states are identical.

- $e.m(\overline{e})$: By semantics, the command has no effect on the environment, so it suffices to show $h_0 \sim h_0'$. (This is fortunate, because the statement of the Lemma uses identifiers η_0, η_0' that are used differently in the semantics.)

  We show $h_0 \sim h_0'$ by cases on κ_4. If $\kappa_4 = H$ then it is possible that $\ell \neq \ell'$ and thus the two calls can have different behavior. But by the typing constraint $\kappa_4 \leq \kappa_3$ we have $\kappa_3 = H$ and thus $\mathit{Hconf}\,\mu$ yields $h_0 \sim h \sim h' \sim h_0'$. It remains to consider the case $\kappa_4 = L$. In this case, we have $\ell = \ell'$ by Lemma 6.1. Now let $\Delta_0 = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{this} : (\mathit{loctype}\,\ell, \mathit{level}\,\ell)$, $\eta_0 = [\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]$, and $\eta_0' = [\overline{x} \mapsto \overline{d'}, \mathit{this} \mapsto \ell']$. We claim that $\mathit{ok}\,\Delta_0\,\eta$ and $\mathit{ok}\,\Delta_0\,\eta'$ and $\eta_0 \sim_{\Delta_0} \eta_0'$. Then we get the result $h_0 = h_0'$ by *safe* μ. Owing to our choice of identifiers in the typing rules, the proof of the claim is just the same as in the case for method call as expression (see the proof of Lemma 6.1).

- $S_1;\ S_2$: Use induction and L-confinement (Lemma 5.2) for S_1; then induction on S_2.

- `if` e S_1 `else` S_2: We proceed by cases on level κ_5 of the guard e. Suppose $\kappa_5 = L$. Then by Lemma 6.1 for e, $b = b'$. If $b = \mathsf{true}$, the result follows by induction on S_1 and if $b = \mathsf{false}$, the result follows by induction on S_2. Consider the other case, $\kappa_5 = H$. By typing, $\kappa_6 = H = \kappa_7$ and $\kappa_1 = \kappa_2 = \kappa_3 = \kappa_4 = H$. Let $(\eta_0, h_0) = [\![\Delta^\dagger;\ C \vdash \texttt{if}\ e\ S_1\ \texttt{else}\ S_2 : \texttt{com}]\!]\mu\eta h$ and $(\eta_0', h_0') = [\![\Delta^\dagger;\ C \vdash \texttt{if}\ e\ S_1\ \texttt{else}\ S_2 : \texttt{com}]\!]\mu\eta' h'$.

By H-confinement Lemma 5.4 we have $\eta \sim_\Delta \eta_0$, $\eta' \sim_\Delta \eta'_0$, $h \sim h_0$, and $h' \sim h'_0$. Using assumptions $\eta \sim_\Delta \eta'$ and $h \sim h'$ we get $\eta_0 \sim_\Delta \eta'_0$ and $h_0 \sim h'_0$ by transitivity.

- `var` (T, κ_1) $x := e$ `in` S: First, we have $[\eta \mid x \mapsto d] \sim_{(\Delta, x:(T,\kappa_1))} [\eta' \mid x \mapsto d']$ because if $\kappa_1 = L$ then by typing $\kappa_4 = L$, so, by Lemma 6.1, $d = d'$. Second, we have $ok\,(\Delta, x : (T, \kappa_1))\,[\eta \mid x \mapsto d]$ by Lemma 5.1. So we can use induction on S to get $\eta_0 \sim_{(\Delta, x:(T,\kappa_1))} \eta'_0$ and $h_0 \sim h'_0$, and hence $(\eta_0 \lfloor x) \sim_\Delta (\eta'_0 \lfloor x)$.

□

Theorem 6.3 (Noninterfering programs) The meaning $\hat{\mu}$ of a well-formed class table is safe: $safe\ \hat{\mu}$.

Proof: Because $safe\ \hat{\mu}$ is defined as a fixpoint, we first show that $safe\ \mu_i$ for all i, by induction on i. Then the result follows by fixpoint induction.

We have $safe\ \mu_0$ because $\mu_0 Cm$ is constantly $\bot$.

Suppose $safe\ \mu_i$, to show $safe\ \mu_{i+1}$. By definition, we must show $safe\ \mu_{i+1} Cm$ for each C, m. There are two cases, depending on whether m is declared or inherited.

Suppose m has declaration $M = \tau_1\ \kappa_1\ m(\overline{\tau}\ \overline{x})\ \kappa_2\ \{S;\ \texttt{return}\ e\}$ in C and let $\Delta = \overline{x} : \overline{\tau}, \mathit{this} : (C, \mathit{level}\, C)$. By Lemmas 5.3 and 5.5 we have $ok\,\mu_i$ and $Hconf\,\mu_i$. Suppose $ok\,\Delta\,\eta$ and $ok\,h$, and let $(\eta_0, h_0) = [\![(\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash S : \texttt{com}]\!]\mu_i \eta h$ (if the outcome is $\bot$ there is nothing more to prove). By Lemma 5.2, L-confinement of commands, we have $ok\,\Delta\,\eta_0$, $ok\,\Delta\,\eta'_0$, $ok\,h$, and $ok\,h'$. By Lemma 6.2, safety for commands, we have $h_0 \sim h'_0$ and $\eta_0 \sim_\Delta \eta'_0$. It remains to show that if the result level κ_4 for m is L we have $d = d'$. But by security typing for method declarations, if $\kappa_4 = L$ then the return expression e is typed L. Using $ok\,\mu_i$, $\eta_0 \sim_\Delta \eta'_0$, etc., Lemma 6.1, safety for expressions, yields $d = d'$. This concludes the proof of $safe\ \mu_{i+1} Cm$.

Suppose m is inherited in C from superclass D. Let $\Delta_C = \overline{x} : \overline{\tau}, \mathit{this} : (C, \mathit{level}\, C)$ and $\Delta_D = \overline{x} : \overline{\tau}, \mathit{this} : (D, \mathit{level}\, D)$. We claim that, for any η, η', $ok\,\Delta_C\,\eta \Rightarrow ok\,\Delta_D\,\eta$ and $\eta \sim_{\Delta_C} \eta' \Rightarrow \eta \sim_{\Delta_D} \eta'$. Then $safe\ \mu_{i+1} Cm$ follows from the claim and $safe\ \mu_{i+1} Dm$ which was already proved. (Strictly speaking we are using secondary induction on inheritance chains.)

For the claim, we only need to consider *this*, as otherwise Δ_C and Δ_D are the same. For *this*, $ok\,\Delta_D\,\eta$ requires $\mathit{level}\, D = L \Rightarrow \eta\, this \in LLoc$. From $C \leq D$ we get $\mathit{level}\, D \leq \mathit{level}\, C$ by the typing rule for classes. Moreover, since m is inherited from C the rule requires $\mathit{level}\, D = \mathit{level}\, C$ so we are done. □

7. Discussion

Beyond the progress reported here, much remains to be done. Non-interference is an attractive property because it can be easily formalized and can provide a precise description of end-to-end security in a system. By itself, however, noninterference as an information flow policy can be rather restrictive. As has been shown by several researchers, a controlled amount of declassification or downgrading of sensitive information is needed in realistic systems for them to be useful. As a future extension of this work, we expect to formalize noninterference in the presence of declassification following the work of Zdancewic and Myers [35].

One direction of work that we have already pursued is adding Java's access control mechanism to the core language [3]. We plan to add information flow annotations to this language. Then, allowing declassification may lead to leakage of information, but the access control mechanism can possibly be used to obtain a noninterference result.

Java has quite a few features beyond the language treated here. To extend our language to the remaining features of JavaCard [7], the semantics can be extended using standard techniques. To treat expressions with side effects, both the environment and the heap would be threaded through the semantics of expressions. We have avoided this in the current paper because it is unilluminating. Exceptional control flow would add further semantic complications of a similar kind. The other missing features have to do with scope and visibility: protected fields, private and protected classes, interfaces, and packages. These features can be treated in the typing rules, similarly to our treatment of private fields, and the semantic consequences could perhaps be exploited to reduce the need for security annotations.

Features of Java beyond those of JavaCard pose a bigger challenge: threads, class loading [9], reflection, and serialization. Specifying noninterference for such constructs would probably go hand-in-hand with specification of pointer confinement and data abstraction properties.

As a step towards more general pointer confinement and abstraction, we are already studying polymorphic classes as in GJ [6]. Label polymorphism is also desirable [21, 20], e.g., for library classes. Label polymorphism might lessen the practical need for `H`-subclasses of `L`-classes, which in turn would allow simplification of the security typing rules.

An important implementation issue is which security annotations can be left implicit, to be inferred by a type reconstruction algorithm. We have not addressed type reconstruction in the current work, but expect that techniques from Pottier *et al.* can be adapted [22, 23].

Acknowledgement: To Geoffrey Smith for discussions that helped clarify noninterference and to the anonymous referees for their comments.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 147–160, 1999.

[2] K. Arnold and J. Gosling. *The Java Programming Language, second edition*. Addison-Wesley, 1998.

[3] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177, 2002.

[4] G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldorp and T. Sato, editors, *Proceedings of FLOPS'99*, volume 1722 of *LNCS*, pages 53–67. Springer-Verlag, 1999.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP 2001 - Object-Oriented Programming*, pages 2–27, 2001.

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Oct. 1998.

[7] Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, 2000.

[8] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, pages 53–76, 2001.

[9] R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[10] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[11] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[12] E.Ferrari, P.Samarati, E.Bertino, and S.Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 130–140, 1997.

[13] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[14] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 241–253, 2001.

[15] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–377, 1998.

[16] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 271–285, 1991.

[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–459, May 2001.

[18] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Technical Report 160, COMPAQ Systems Research Center, Nov. 2000. To appear in TOPLAS.

[19] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 126–142, 2001.

[20] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[21] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings, IEEE Symposium on Security and Privacy*, pages 186–197, 1998.

[22] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the fifth ACM International Conference on Functional Programming*, pages 46–57, 2000.

[23] F. Pottier and V. Simonet. Information flow inference for ML. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[24] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.

[25] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 200–215, 2000.

[26] A. Sabelfeld and D. Sands. A Per model of secure information flow in sequential programs. *Higher-order and Symbolic Computation*, 14(1):59–91, 2001.

[27] G. Smith. A new type system for secure information flow. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 115–125, 2001.

[28] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 355–364, 1998.

[29] I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, 1994. http://www.dcs.ed.ac.uk/home/stark/publications/thesis.html.

[30] E. Sumii and B. Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop*, pages 256–269, 2001.

[31] J. Vitek and B. Bokowski. Confined types in java. *Software Practice and Experience*, 31(6):507–532, 2001.

[32] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, number 1214 in LNCS, pages 607–621. Springer-Verlag, 1997.

[33] D. Volpano and G. Smith. Confinement properties for multi-threaded programs. *Electronic Notes in Theoretical Computer Science*, 20, 1999.

[34] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[35] S. Zdancewic and A. Myers. Robust declassification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.