Representation Independence, Confinement and Access Control [extended abstract and appendices]

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

Denotational semantics is given for a Java-like language with pointers, subclassing and dynamic dispatch, class oriented visibility control, recursive types and methods, and privilege-based access control. Representation independence (relational parametricity) is proved, using a semantic notion of confinement similar to ones for which static disciplines have been recently proposed.

1. INTRODUCTION

For scalable systems, scalable system-building tools, and scalable development methods, abstraction is essential. Abstraction means that for reasoning about an individual component it is sufficient to consider other components in terms of their behavioral interface rather than the internals that implement the interface. Abstraction is needed for the automated reasoning embodied in static analysis tools, e.g., bytecode verifiers, and it is needed for formal and informal reasoning about functional correctness during development and validation. Modular reasoning has always been a central issue in software engineering and in static analysis, but with the ascendancy of mobile code it has become absolutely essential.

Abstraction is only sound to the extent that implementation internals are encapsulated. For example, the privilege-based access control system of Java [11] is intended to ensure certain security properties, but its proper functioning depends on language-based encapsulation, e.g., type safety. Visibility controls, such as private fields and opaque types, ensure certain information hiding properties. The classical way to give a precise analysis of such properties is in terms of representation independence [33, 24]: client programs of a component are insensitive to differences in data representation, provided the representation is private and the behavior of the component through its visible interface is the same.

The main proof technique for representation independence is so fundamental that it has appeared in many places, with a variety of names, e.g., simulation, logical relations, abstraction mappings, relational parametricity (e.g., [31, 35, 20, 8]). Our main result, an abstraction theorem, says that simulation can be used for a rich fragment of Java. The result is given using a denotational semantics in the manner of Scott-Strachey [39]. We apply the result to one of the Meyer-Sieber equivalences [21] that was a longstanding challenge for the semantics of Algol-like languages [30]. We discuss relational proofs of the equivalence of "security passing style" [42] with the lazy "stack inspection" implementation of Java's privilege-based access control mechanism [11], and then extend our language to include access control. We give an abstraction theorem for this extended language.

The need for confinement. Language-based encapsulation often runs afoul of aliasing. For variables and parameters, aliasing can be prevented through syntactic restrictions that are tolerable in practice. Aliasing via pointers is an unavoidable problem in object oriented programming where shared mutable objects are pervasive. Here is the problem: Suppose class A has a private field f that points to a mutable object intended to be an encapsulated part of the representation of A. If a client program can also reference the object, the client can behave in representation-dependent ways. Moreover, if it updates the object's state, an invariant of A may be violated.

In simply typed languages, typed pointers help somewhat: pointer variables x, y are not aliased if they have different types. This help is undercut by subclass polymorphism: in Java, a variable x of type `Object` can alias y of any type.

The need for confinement is ubiquitous and pressing in practice [17] (for the academic reader, evidence can be found in the number of recent OOPSLA and ECOOP papers on the topic). To achieve encapsulation for heap-allocated objects, there have been many proposals for confinement. For example, fields can be designated as unshared or read-only. Quite a few confinement disciplines have been proposed (e.g., see [40, 7, 27, 4] and citations therein). Most have significant shortcomings; they are too restrictive for practical use, or not efficiently checkable. Few have been formally justified. Existing justifications use disparate techniques such that it

*Supported by NSF grants EIA-9806835 and CCR-0093080

†Supported by NSF grant INT-9813854

POPL '02 Jan. 16–18, 2002 Portland, OR USA

is quite hard to assess and compare confinement disciplines [7, 25, 18]. One of our contributions is to show how standard semantic techniques can be used for such assessments.

The challenges. A widely held view seems to be that classical techniques based on denotational semantics and logical relations are inadequate in the face of the complex language features of interest. The combination of local state with higher order procedures makes it difficult to prove representation independence even for Algol, where procedures can be passed as arguments but not assigned to state variables [29]; objects exhibit similar features. If procedures can be stored in state variables or in heap objects, the resulting domain equations become quite challenging to solve. For applications in security and automated static checking, it is important to devise robust, comprehensible models that support not only the idealized languages of research studies but also the full languages used in practice.

Difficulties with denotational semantics led to considerable advances using small-step operational semantics [12]. One of the most relevant works is that of Grossman *et al.* [13] where representation independence is approached using the notion of *principal* from the literature on security. To prove that clients are independent from the representation of an abstraction provided by a host program, a wrapper construct is used to tag code fragments with their owner principal (e.g., client or host), and to provide an opaque type for the client's view of the abstraction. This is a promising approach, although their results seem rather limited: it is shown that if no host code is involved, the client cannot distinguish between different values of the abstract type. This sort of result is used in analysis of information flow, and can be proved using logical relations [1]: a client is shown to preserve the complete (everywhere-true) relation on confidential data. But logical relations are more powerful: the client can actually use host code, and its behavior can depend on values of the abstract type. A strong representation independence result says that, although the behavior can depend on the abstract values, it cannot depend on their internal representations [22, 23, 24]. Among the many uses of strong representation independence are program transformations and justification of logics for reasoning about data abstraction and modification of encapsulated state.

Although Grossman *et al.* offer their work as a simpler alternative to domain theoretic semantics, the technical treatment is somewhat intricate by the time the language is extended to include references, recursive and polymorphic types. Except for parametric polymorphism, we treat all these features, as well as subclassing and privilege-based access control.

Results. Using an idealized Java-like language, our abstraction theorem says that if two implementations are given for a class A, and methods of these implementations preserve a given relation on A-objects together with their confined representation objects, then there are induced relations for all environments and heaps, and those relations are preserved by all client programs. An abstraction theorem can be used to prove equivalence or refinement of class implementations, general program equivalences, and correctness of static analyses. e.g., secure information flow. The proof technique has three steps. (1) Give a relation describing the way in which two class declarations implement "the same abstraction", and show that it is preserved by all methods of the class. (2) Show that the induced (family of) relations are preserved by all client programs (the *abstraction theorem*). (3) Show that, for programs in which the abstraction is encapsulated (e.g., in private fields or local variables), the relation is the identity (the *identity extension lemma*).[1]

As we have formulated the abstraction theorem, it can be applied directly to prove command and class equivalences, and we give some examples. For applications in static analysis, the problem is usually to show that a syntax directed system of types and effects approximates some property like secure information flow. Such results are also proved using relations, and it was our own work on security analysis [3] that gave us the courage to tackle parametricity for Java. We have not attempted to formulate an abstraction theorem general enough to apply directly in such analyses; they use analysis-specific typing systems rather than the language's own types and syntax. But the essence of our result is that the language is relationally parametric, given suitable confinement conditions. It is clear, for example, that our static analysis for Java access control will extend to the language treated in the sequel.

Why the results are achievable. Our core language is quite rich. We include recursive classes and methods, and inheritance and casts as in Featherweight Java (FJ) [15] but with private visibility for fields. We also include mutable state, pointers, and type tests (`instanceof`).

Our work grew out of a study of Java's stack-based access control mechanism. We used an idealized language similar to related work in the area [38, 42] and were surprised to find that a straightforward denotational semantics could be used to prove strong results using simulation. But the mechanism itself just protects calls to certain methods. To prove, for some program using the mechanism, that some intrinsic security property holds, one needs to show that information does not leak. So we wanted to prove a general parametricity result that could be applied to information flow, and we wanted to include references.

The problem is that, to get an adequate induction hypothesis, parametricity needs to be imposed on the latent effects of procedure abstractions.[2] These conditions are most easily expressed in terms of a denotational model, but if procedures can be stored in the heap on which they act, difficult domain equations must be solved and the resulting complexity then pervades the theory. Domain equations must also be solved to interpret most languages with recursive types. Although Java syntax seems less elegant than, say, lambda calculus, it has several features that ease the difficulties. Owing to name-based type equivalence and subtyping, and the binding of methods to objects via their class, we can use simple domains. The absence of pointer arithmetic means we need only a mild assumption about the memory allocator. The fact that type names are semantically relevant lets us use them to formulate in semantic terms a confinement condition similar to those in the literature [40, 25, 7] (but to keep things simple, we do not distinguish read-only access). Our results are proved on the basis of this semantic condition. A modular static analysis, which does not require code anno-

[1] In the case of refinement, identity is replaced by inequality [28]. In this paper we do not emphasize refinement.

[2] As a property to be proved or, ideally, as an intrinsic feature of the semantic model [34, 29].

tations, can be derived from the semantic definition [2].

Overview of the paper. Section 2 introduces the language with an example showing the necessity of confinement. Then typing and semantic definitions are given; we defer the access control facility to Section 6. Section 3 formulates the basic situation to which the abstraction theorem applies, including confinement, and defines the induced relations. Section 4 gives the theorem and 5 applies it a Meyer-Sieber equivalence. Section 6 adds access control to the core language and extends the abstraction theorem accordingly. The results are discussed in Section 7.

2. CORE LANGUAGE

The concrete syntax is based on that of Java, with some restrictions for ease of formalization; for example, "return" statements appear only at the end of a method, and effects like object construction (`new`) occur in commands rather than expressions. There are a couple of minor deviations from Java, e.g., the keyword `var` marks local variable declarations. We retain the format `T x` for declaration of a variable `x` of type `T`, while writing `x:T` in typing rules.

2.1 Examples

A program consists of a collection of class declarations like the following one.

```
class Boolean extends Object        {
  bool f;
  unit set(bool x){ this.f := x; return unit }
  bool get(){ skip; return this.f } }
```

Instances of class `Boolean` have a private field `f` with the primitive type `bool`. There is no constructor; fields of new objects are given their Java defaults (null, false). Fields are considered to be private to their class and methods public: fields are only visible to methods declared in this class, but methods are visible to all classes. Fields are accessed in expressions of the form `this.f`, using "`this`" to refer to the current object; a bare identifier `x` is either a parameter or a local variable. Every method has a return type; `unit` corresponds to Java's "void" and is used for methods like `set` that are called only for their effect on object state. In subsequent examples we omit returns for the `unit` value, and sometimes omit "`this`". Object types are implicitly pointers, so assignments create aliases. The built-in equality test `==` compares references.

A convenient simplification is to preclude side effects in expressions. We make no such restriction on the syntax, but the semantics discards effects of expressions, so our results are only interesting for programs that do not exploit expression effects. For example, the `get` method above is useful in expressions, but not `set`.

Here is a class that uses a `Boolean` for its private state.

```
class A0 extends Object         {
  Boolean g;
  unit init(){ this.g:= new Boolean(); g.set(true)}
  unit setg(bool x){ g.set(x) }
  bool getg(){ return g.get() } }
```

An alternative implementation of `A0` uses an isomorphic representation to achieve the same behavior.

```
class A0 extends Object                {
  Boolean g';
  unit init(){
    this.g':= new Boolean(); g'.set(false) }
  unit setg(bool x){ g'.set(not(x)) }
  bool getg(){ return not(g'.get()) } }
```

Let us consider, informally, how representation independence is formulated and used for the example. First, we give a relation between states of objects `o` and `o'` for the two implementations.[3] We say `o` and `o'` are related just if either `o.g` = `null` = `o'.g'` or `o.g` ≠ `null` ≠ `o'.g'` and `o.g.f` = ¬(`o'.g'.f`). If `o,o'` are newly constructed, `o.g` = `null` = `o'.g'` holds. Invocations of `setg` and `getg` clearly maintain the relation.

According to the abstraction theorem, the relation is maintained by all client programs. And this implies that a client using `o` in a local variable or field cannot be distinguished from one using `o'`. However, such a conclusion is unwarranted unless `o,o'` are suitably confined. For example, suppose we add to `A0` a method

```
Object bad(){ return g }
```

(For the second, primed version of `A0`, let it return `g'`.) A client class `C` can exploit method `bad` using a `(Boolean)` cast:

```
var A0 z := null in
  z := new A0();
  var Boolean w := (Boolean) z.bad() in
    if (w.get()) then skip else diverge;
```

Although the field `g` of `A0` is not visible to methods in class `C`, method `bad` leaks a reference to the representation object. The command diverges if the primed implementation of `A0` is chosen, but does not diverge with the first implementation.

2.2 Syntax

To formalize the language, we adapt some notations from FJ [15]. To avoid burdening reader with straightforward technicalities we deliberately confuse surface syntax with abstract syntax, and we do not distinguish between classes and class types. We also confuse syntactic categories with names of their typical elements. 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 $\overline{T}$. The bar has no semantic import; $\overline{T}$ has nothing to do with T.

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*).

$$
\begin{array}{rcl}
T & ::= & \texttt{bool} \mid \texttt{unit} \mid C \\
CL & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\
M & ::= & T\ m(\overline{T}\ \overline{x})\{S;\ \texttt{return}\ e\} \\
S & ::= & x := e \mid x.f := e \mid x := \texttt{new}\ C() \mid x.m(\overline{e}) \mid \\
 & & \texttt{if}\ e\ S_1\ \texttt{else}\ S_2 \mid \texttt{var}\ T\ x := e\ \texttt{in}\ S \mid S;\ S \\
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \\
 & & e{=}{=}e \mid (C)\ e \mid \texttt{null} \mid e\ \texttt{instanceof}\ C
\end{array}
$$

[3] In practice, one would show that this relation is established by the constructors. Our omission of constructors means that examples may need initialization methods and extra fields to track whether initialization has been done. In the case at hand, nullity of `g` suffices.

Table 1: Typing rules for expressions and commands

$$\Gamma;\ C \vdash x : \Gamma x \qquad \Gamma;\ C \vdash \texttt{null} : B$$

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

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

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

$$\frac{x \neq \mathit{this} \quad \Gamma;\ C \vdash e : T \quad T \le \Gamma x}{\Gamma;\ C \vdash x := e : \texttt{com}} \qquad \frac{\Gamma x = C \quad T f \in \mathit{dfields}\, C \quad \Gamma;\ C \vdash e : U \quad U \le T}{\Gamma;\ C \vdash x.f := e : \texttt{com}}$$

$$\frac{x \neq \mathit{this} \quad B \le \Gamma x}{\Gamma;\ C \vdash x := \texttt{new}\ B(\,) : \texttt{com}} \qquad \frac{\mathit{mtype}(m, \Gamma x) = (\overline{x} : \overline{T}) \to T \quad \Gamma;\ C \vdash \overline{e} : \overline{U} \quad \overline{U} \le \overline{T}}{\Gamma;\ C \vdash x.m(\overline{e}) : \texttt{com}}$$

$$\frac{\Gamma;\ C \vdash e : \texttt{bool} \quad \Gamma;\ C \vdash S_1 : \texttt{com} \quad \Gamma;\ C \vdash S_2 : \texttt{com}}{\Gamma;\ C \vdash \texttt{if}\ e\ S_1\ \texttt{else}\ S_2 : \texttt{com}} \qquad \frac{\Gamma;\ C \vdash S_1 : \texttt{com} \quad \Gamma;\ C \vdash S_2 : \texttt{com}}{\Gamma;\ C \vdash S_1;\ S_2 : \texttt{com}}$$

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

Well formed class declarations are specified by rules below and in Table 1. 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. 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 several auxiliary notions that are defined in terms of CT; dependence on CT 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. The rules for field access and update enforce privacy. Slight desugaring is needed to express the examples, e.g., `g.set(x)` is short for `var Boolean y := this.g in y.set(x)`.

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

To define some auxiliary notation, let M be in $\overline{M}$, with

$$\begin{array}{lcl} CT(C) & = & \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\ M & = & T\ m(\overline{T}\ \overline{x})\{S;\ \texttt{return}\ e\} \end{array}$$

Then we define $\mathit{mtype}(m, C) = (\overline{x} : \overline{T}) \to T$. For the declared fields, we define $\mathit{dfields}\, C = \overline{T}\ \overline{f}$ and $\mathit{type}(\overline{f}, C) = \overline{T}$. 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 undeclared class `Object` has no methods or fields. Note that $\mathit{mtype}(m, C)$ is undefined if m is not declared or inherited in C. Here is the typing rule for method declarations:

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

A class declaration is well formed if all of its methods are:

$$\frac{C\ \texttt{extends}\ D \vdash M \ \text{ for each } M \in \overline{M}}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \}}$$

2.3 Semantics

The state of a method in execution is comprised of a *heap* h, which maps locations to object states, and an *environment* η, which assigns locations and primitive values to local variables and parameters. Every environment of interest includes *this* which points to the target object.

We assume that a countable set *Loc* is given, along with a distinguished entity nil not in *Loc*. A heap h is a finite partial function from *Loc* to object states. To streamline notation, we treat object states as mappings from field names to values. We also need to track the object's class, which is not mutable. As a harmless coding trick, we assume given a function $\mathit{loctype} : \mathit{Loc} \to \mathit{ClassNames}$ such that for every class C there are an infinite number of locations ℓ with $\mathit{loctype}\ \ell = C$. We write $\mathit{locs}\ C$ for $\{\ell \mid \mathit{loctype}\ \ell = C\}$ and $\mathit{locs}(C{\downarrow})$ for $\{\ell \mid \mathit{loctype}\ \ell \le C\}$.

One of the shortcomings of classical Scott-Strachey semantics is its sensitivity to choice of locations. If two encapsulated data representations take different amounts of memory, it is quite possible that a real memory allocator will return different addresses even in states where the two representations are related. One way to deal with this issue is to consider all possible allocators, and then quotient the resulting model. We choose a simpler solution; we just assume that a parametric allocator is given.

Definition 1. An *allocator* is a 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* provided that for all classes C and heaps h_1, h_2

$$\begin{array}{l}\mathit{dom}\, h_1 \cap \mathit{locs}\, C = \mathit{dom}\, h_2 \cap \mathit{locs}\, C \\ \Rightarrow \mathit{fresh}(C, h_1) = \mathit{fresh}(C, h_2)\end{array}$$

For example, if $\mathit{Loc} = \mathbb{N}$ the function $\mathit{fresh}(C, h) = \mathit{min}\{\ell \mid \mathit{loctype}\ \ell = C \wedge \ell \notin \mathit{dom}\, h\}$ is a parametric allocator. Capability-based systems provide locations as abstract values in order to achieve parametricity for secure information flow.

Table 2 gives the semantic domains. There are several auxiliary domains, such as *Heap* and *C state*, elements of which are not directly denotable. We describe these semantic categories θ in a grammar, for use later in the definition of the induced relations. We write $\rightharpoonup$ for finite partial functions. It is easy to show if $T \le U$ then $[\![T]\!] \subseteq [\![U]\!]$.

Table 3 gives the semantics of expressions, dependent on an arbitrary method environment μ in $[\![\mathit{MEnv}]\!]$. Table 4 gives the semantics of commands.

It is straightforward to show that, as in Java, no program constructs create dangling pointers. One might expect us

Table 2: Semantic domains.

$$\theta \ ::= \ T \mid \Gamma \mid C \ state \mid Heap \mid (C, (\overline{x} : \overline{T}) \to T) \mid MEnv \mid \theta_\perp \mid \theta \times \theta$$

$[\![\mathtt{bool}]\!] = \{\mathsf{true}, \mathsf{false}\} \qquad [\![\mathtt{unit}]\!] = \{\bullet\} \qquad [\![C]\!] = \{\mathsf{nil}\} \cup \{\ell \mid \ell \in Loc \wedge loctype\ \ell \leq C\}$

$\eta \in [\![\Gamma]\!] \Leftrightarrow dom\,\eta = dom\,\Gamma \wedge \forall x \in dom\,\eta\ .\ \eta\,x \in [\![\Gamma\,x]\!]$

$s \in [\![C\ state]\!] \Leftrightarrow dom\,s = fields\,C \wedge \forall f \in fields\,C\ .\ sf \in [\![type(f, C)]\!]$

$h \in [\![Heap]\!] \Leftrightarrow dom\,h \subseteq Loc \wedge \forall \ell \in dom\,h\ .\ h\ell \in [\![(loctype\ \ell)\ state]\!]$

$[\![C, (\overline{x} : \overline{T}) \to T]\!] \ = \ [\![\overline{x} : \overline{T}, this : C]\!] \to [\![Heap]\!] \to ([\![T]\!] \times [\![Heap]\!])_\perp$

$[\![MEnv]\!] \ = \ (C : ClassNames) \rightharpoonup (m : MethodNames) \rightharpoonup [\![C, mtype(m, C)]\!]$

$[\![\theta_1 \times \theta_2]\!] = [\![\theta_1]\!] \times [\![\theta_2]\!] \qquad [\![\theta_\perp]\!] = [\![\theta]\!]_\perp$ (we assume $\perp$ is not in $[\![\theta]\!]$)

Meanings of expressions: $[\![\Gamma;\ C \vdash e : T]\!] \in [\![MEnv]\!] \to [\![\Gamma]\!] \to [\![Heap]\!] \to [\![T]\!]_\perp$

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

to confine attention to heaps that are *closed*, in the sense that every location in the range is in the domain. But this would add unenlightening complications to some definitions. So the semantics is defined even for heaps with dangling pointers. Like cast failures, dereferences of dangling pointers and nil are considered an error. Rather than modelling exceptions, we identify all errors, and divergence, with the improper value $\perp$.

The semantic definitions 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. Function update is written, e.g., $[\eta \mid x \mapsto d]$.

The semantic definitions are given with respect to an arbitrary method environment, but we are really interested in the method environment given as a fixpoint, properly interpreting mutually recursive methods. Toward this end, it is convenient to define the semantics of method declarations. If, in class C, we have $M = T\ m(\overline{T}\ \overline{x})\{S;\ \mathtt{return}\ e\}$, then for any μ, define $[\![M]\!]\mu \in [\![(C, (\overline{x} : \overline{T} \to T))]\!]$ by

$$\begin{array}{l}[\![M]\!]\mu\eta h \\ = \ \ \textbf{let}\ (\eta_0, h_0) = [\![(\overline{x} : \overline{T}, this : C);\ C \vdash S : \mathtt{com}]\!]\mu\eta h\ \textbf{in} \\ \quad\ \ \textbf{let}\ d = [\![(\overline{x} : \overline{T}, this : C);\ C \vdash e : T]\!]\mu\eta_0 h_0\ \textbf{in}\ (d, h_0)\end{array}$$

We only take fixpoints for method environments; these are ordered pointwise, in terms of method meanings. The latter are function spaces $[\![C, (\overline{x} : \overline{T}) \to T]\!]$ into lifted sets. Thus, to get pointed complete partial orders for method environments, it suffices to use equality as the order on $[\![Heap]\!]$, $[\![\mathtt{bool}]\!]$, $[\![C]\!]$, and $[\![C\ state]\!]$. We define an ascending chain $\mu \in \mathbb{N} \to [\![MEnv]\!]$ as follows.

$$\begin{array}{l}\mu_0\,C\,m = \lambda\eta.\ \lambda h.\ \perp \\ \mu_{j+1}\,C\,m = [\![M]\!]\mu_j \quad \text{if } m \text{ declared as } M \text{ in } C \\ \mu_{j+1}\,C\,m = \mu_{j+1}\,B\,m \quad \text{if } m \text{ inherited from } B \text{ in } C\end{array}$$

The semantics of the class table is the least upper bound, $\hat{\mu}$, of this chain.

3. SIMULATION AND CONFINEMENT

The aim of this section is to formulate the (indexed family of) relations defined inductively, based on a given relation connecting two representations for an instance of a class A. After introducing convenient notation for heaps, from BI pointer logic [16], we formalize the basic situation in which a relation is given. Then we formalize confinement and the induced relation.

We say heaps h_1 and h_2 are *disjoint* if $dom\,h_1 \cap dom\,h_2 = \varnothing$. Let $h_1 * h_2$ be the union of h_1 and h_2 if they are disjoint, and undefined otherwise. To say that no objects in h_1 contain references to objects in h_2, we define[4]

$$h_1 \not\rightsquigarrow h_2 \ \Leftrightarrow\ \forall \ell \in dom\,h_1\ .\ rng(h_1\,\ell) \cap dom\,h_2 = \varnothing$$

It is crucial for our formulation of confinement that we work with partitions $h_1 * h_2$ where h_1 and h_2 have dangling pointers (as in the pioneering work of Reynolds [36]).

In our formalization, representation objects for A all have a common supertype *Rep*. We shall partition the heap as $h = hA * hRep * hOut$ where hA contains just the object of class A being abstracted, $hRep$ contains zero or more objects of class *Rep* that comprise the representation of A, and $hOut$ contains all other objects.

Beware that we use multi-letter identifiers like hA that beg to be (wrongly) parsed as an application $h\,A$ of h to A.

Our use of class names depends on the assumption that $Rep \not\leq A$ and $Rep \not\geq A$, and the same for Rep'. It also precludes use of, say, a library class both in representions and in outside objects. But these convenient coding tricks are easily circumvented (e.g., as in [25]).

Definition 2. A *basic simulation* is given by:

- well formed class tables CT, CT' that are identical save for their values on A, where

  $$\begin{array}{l}CT(A) = \mathtt{class}\ A\ \mathtt{extends}\ B\ \{\ \overline{T}\,\overline{g};\ \overline{M}\ \} \\ CT'(A) = \mathtt{class}\ A\ \mathtt{extends}\ B\ \{\ \overline{T}'\,\overline{g}';\ \overline{M}'\ \}\end{array}$$

  We write $\vdash, \vdash'$ for the typing relations determined by CT, CT' respectively, and $[\![-]\!], [\![-]\!]'$ for the semantics.

- two distinguished classes Rep, Rep'

[4] One might think it better to define the positive notion that all pointers from h_1 are into h_2, but this is less convenient because we allow dangling pointers.

Table 3: Semantics of expressions.

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

- relation R such that $R \subseteq [\![\mathit{Heap}]\!] \times [\![\mathit{Heap}]\!]'$
- for any h, h', if $R\ h\ h'$ then there are partitions $h = hA * hRep$ and $h' = hA' * hRep'$ and location ℓ with $\mathit{loctype}\ \ell \leq A$, such that
  - $\mathit{dom}\, hA = \{\ell\} = \mathit{dom}\, hA'$,
  - $\mathit{dom}(hRep) \subseteq \mathit{locs}(Rep\downarrow)$ (and for $hRep', Rep'$),
  - $\mathcal{R}\ (\mathit{type}(f, \mathit{loctype}\,\ell))\ (h\ell f)\ (h'\ell f)$ for all fields $f \in \mathit{fields}(\mathit{loctype}\,\ell)$ with $f \notin \overline{g}$ (see below).
- default states are related: for any $\ell \in \mathit{locs}(A\downarrow)$, R relates $[\ell \mapsto [\mathit{fields}(\mathit{loctype}\,\ell) \mapsto \mathit{defaults}]]$ to $[\ell \mapsto [\mathit{fields}'(\mathit{loctype}\,\ell) \mapsto \mathit{defaults}']]$

Relation R is used to connect the private fields of a pair of objects for the two implementations of A, along with the representation objects referenced by those fields. The condition involving $\mathcal{R}$ is a healthiness assumption imposed on inherited fields and subclass fields. It is formulated using the induced relations $\mathcal{R}$ of Definition 5. This forward reference introduces no circularity in our definitions. In fact the condition simply requires $h\ell f = h'\ell f$, because $\mathcal{R}\,T$ is the identity for every data type T, but we feel that explicit use of $\mathcal{R}$ is conceptually more clear.

The induced relations $\mathcal{R}$ handle objects other than *Rep* objects. We say C is a *non-rep* class iff $C \not\leq \mathit{Rep}$ and $C \not\leq \mathit{Rep}'$. We extend this to categories θ so that C *state* and $(C, (\overline{x} : \overline{T}) \to T)$ are non-rep iff C is.

Examples. For the example of `A0` in Section 2.1, we take $\mathit{Rep}, \mathit{Rep}'$ to be `Boolean,Boolean'` where `Boolean'` is a fresh copy of `Boolean`. We define R to relate h, h' just if [5] $h = [\ell_1 \mapsto [g \mapsto \mathsf{nil}]]$ and $h' = [\ell_1 \mapsto [g' \mapsto \mathsf{nil}]]$, or

$$
\begin{array}{l}
h = [\ell_1 \mapsto [g \mapsto \ell_2], \ell_2 \mapsto [f \mapsto d]] \\
h' = [\ell_1 \mapsto [g' \mapsto \ell_3], \ell_3 \mapsto [f \mapsto \neg d]]
\end{array}
$$

for some boolean d and locations ℓ_1, ℓ_2, ℓ_3.

As another example, consider a class $A2$ that is to contain a non-negative integer. In one version, this is stored as a primitive `int` in a field named f. In the other version, field f' points to a singly linked, nil-terminated list of objects of class Node. We take Rep arbitrary and $\mathit{Rep}' = \mathit{Node}$, and R relates h, h' just if h is a singleton heap $[\ell \mapsto [f \mapsto j]]$ for some $j \geq 0$, and $h' = [\ell \mapsto [f' \mapsto \ell_1]] * h'_1$ where h'_1 contains a singly linked list of length j starting with some ℓ_1.

Confinement. Confinement is imposed on the heap, which can contain multiple instances of the related class A, each with zero or more associated representation objects. We require that in every state the heap can be partitioned as shown in Figure 1, with references restricted as in the picture. In practice, it is important to distinguish read-only references, e.g., if the *Rep* objects are used as nodes in a data structure representing a container or set, these nodes

[5] If `A0` has subclasses, the relation on their fields is determined by the healthiness assumption for basic simulations.

Table 4: Semantics of commands

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

should be able to point to the "outside" objects in *hOut*, but only for reading [25]. But this would complicate our presentation; our aim is to show a set-up in which one can formalize confinement disciplines, compare them, and prove that syntactic restrictions do ensure the desired semantic confinement.

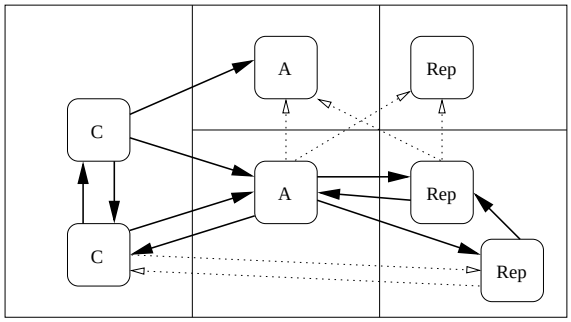

Figure 1: Confinement example. Dotted references disallowed.

Definition 3. An *admissible partition* of heap h is a set of pairwise disjoint heaps $hOut, hA_1, hRep_1, \ldots, hA_k, hRep_k$ with $h = hOut * hA_1 * hRep_1 * \ldots * hA_k * hRep_k$ and for all i

- $dom\, hA_i \subseteq locs(A\downarrow)$ and $size(dom\, hA_i) = 1$
- $dom\, hRep_i \subseteq locs(Rep\downarrow)$
- $dom\, hOut \cap (locs(A\downarrow) \cup locs(Rep\downarrow)) = \varnothing$

Note that a heap may have several admissible partitions, e.g., if there are inaccessible *Rep* objects. The reader can check that the definitions and results do not depend on choice of partition in these cases.

The condition $hOut \not\rightsquigarrow hRep_i$ is clearly needed to preclude representation-dependent behavior as in the example in Section 2.1 using the *bad* method. We also need to impose the condition $hRep_i \not\rightsquigarrow hOut$ to prevent *Rep* objects from depending on outside objects that could be changed by clients; such changes need not preserve the basic simulation relation. Similar conditions are ensured by module-based notions of confinement in the literature (e.g., [40]). But our notion of simulation is instance-based, so we also require that A objects are isolated from each other and each other's representations. This is similar to the ownership model [7, 25] and notions of unique references such as [4].

Definition 4. We say heap h is *confined*, and write *conf* h, iff h has an admissible partition such that for all i, j

- $hOut \not\rightsquigarrow hRep_i$
- $hRep_i \not\rightsquigarrow hOut$
- $i \neq j \;\Rightarrow\; hA_i * hRep_i \not\rightsquigarrow hA_j * hRep_j$

Confinement of an environment η depends on the heap and on the class in which the environment is used. For non-rep C different from A, define *conf* $C\ \eta\ h$ iff $rng\ \eta \cap locs(Rep\downarrow) = \varnothing$. For A, define *conf* $A\ \eta\ h$ iff there is a partition with $rng\ \eta \cap (locs(Rep\downarrow) \cup locs(A\downarrow)) \subseteq dom(hRep_j * hA_j)$ for some j with $dom(hA_j) = \{\eta\, this\}$.

Method environment μ is confined, written *conf* μ, iff for all non-rep C, and all m, h, η, if *conf* h and *conf* $C\ \eta\ h$ then

$$\mu C m \eta h \neq \bot \Rightarrow$$
$$\textbf{let } (d, h_0) = \mu C m \eta h \textbf{ in } conf\ h_0 \wedge d \notin locs(Rep\downarrow)$$

For commands, we say, for non-rep C, that $\Gamma;\ C \vdash S : \texttt{com}$ is confined iff for any confined h, μ and η confined in C, h:

$$[\![\Gamma;\ C \vdash S : \texttt{com}]\!]\mu\eta h \neq \bot \Rightarrow$$
$$\textbf{let } (\eta_0, h_0) = [\![\Gamma;\ C \vdash S : \texttt{com}]\!]\mu\eta h \textbf{ in}$$
$$conf\ h_0 \wedge conf\ C\ \eta_0\ h_0$$

For expressions, for non-rep $C \neq A$ we say $\Gamma;\ C \vdash e : T$ is confined iff for any confined h and μ, and η confined in C, h:

$$[\![\Gamma;\ C \vdash e : T]\!]\mu\eta h \notin locs(Rep\downarrow)$$

For A, we say $\Gamma;\ A \vdash e : T$ is confined iff for any confined h and μ, and η confined in A, h:

$$[\![\Gamma;\ A \vdash e : T]\!]\mu\eta h \neq \bot \Rightarrow$$
$$\textbf{let } d = [\![\Gamma;\ A \vdash e : T]\!]\mu\eta h \textbf{ in}$$
$$d \in (locs(A\downarrow) \cup locs(Rep\downarrow)) \;\Rightarrow\; d \in dom(hA_j * hRep_j)$$

for some j with $dom(hA_j) = \{\eta\, this\}$.

Finally, CT is confined iff (i) for every non-rep $C \neq A$ and every M in $CT(C)$ of form $T\ m(\overline{T}\ \overline{x})\{S; \texttt{return}\ e\}$, it is the case that S, e, and all constituents thereof are confined;
(ii) for every $T\ m(\overline{T}\ \overline{x})\{S; \texttt{return}\ e\}$ declared in A, both S and e are confined; and
(iii) for every C, m with $mtype(m, C) = (\overline{x} : \overline{T}) \to T$ we have (a) $C \leq A \Rightarrow \overline{T} \not\gtrsim Rep \wedge \overline{T} \not\gtrsim A \wedge T \not\gtrsim Rep$
(b) $C \not\leq A \wedge C \not\leq Rep \Rightarrow \overline{T} \not\gtrsim Rep$

The relation $\not\gtrsim$ is defined by $T_1 \not\gtrsim T_2 \Leftrightarrow T_1 \not\leq T_2 \wedge T_1 \not\geq T_2$. Note that $T_1 \not\gtrsim T_2 \Rightarrow [\![T_1]\!] \cap [\![T_2]\!] = \varnothing$. We write $\overline{T} \not\gtrsim T_1$ to express that $T \not\gtrsim T_1$ for all T in $\overline{T}$.

Confinement for CT involves syntactic conditions which ensure confinement of method calls, e.g., a client object cannot pass to one $A\downarrow$-object a reference to another $A\downarrow$-object. This is difficult to express in terms of the semantics. At first glance the syntactic conditions may appear rather strong, but we have not found non-trivial examples that violate the conditions without also violating heap confinement. Elsewhere [2] we give mild syntactic conditions that suffice for confinement of a class table, e.g., methods of non-rep class $C \not\leq A$ cannot construct new *Rep* objects. In practice, *Rep* would be local to a module not containing C.

As an example, let A be class `A0` in Section 2.1, and let *Rep* be `Boolean`. Consider an initial heap with no objects of these types; it is confined. If `x` is a variable of type `A0`, the command `x := new A0()` in a method of some client class $C \not\leq A$ is confined, as the new object has `x.g` initially null. A subsequent call `x.init()` is also confined, as the newly constructed `Boolean` object is not leaked. All of the methods in both versions of `A0` are confined. However, if the `bad` method is added, the example command using it is not confined, due to the expression `z.bad()`. And the `bad` method violates (iii)(a) in the definition of confined class table.

In an earlier version of this paper, we erroneously used the confinement condition for A-environments for subclasses of A, misled by considerations about inheritance. The following result is the key to reasoning about confinement of calls to inherited methods.

LEMMA 3.1. *For nonrep C, B, if $C \leq B$ and conf $C\ \eta\ h$ then conf $B\ \eta\ h$, for all environments η for parameters of methods of B.*

For the main Lemma 4.2, we need constituents of non-A methods to be confined, hence the separate cases (i) and (ii)

in the definition of confined class table. For the following result we only need that method bodies are confined.

LEMMA 3.2. *Suppose CT is confined. Then the environment $\hat{\mu}$ obtained as a fixpoint is confined, as is each μ_i in the approximation chain.*

Induced relation. Finally, we can define the induced relation $\mathcal{R}$. For non-rep categories θ we have $\mathcal{R}\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$.

Definition 5. For heaps h, h', we define $\mathcal{R}$ *Heap* h h' iff *conf* h and *conf* h' and there exist admissible partitions of h, h', of the same size, such that for the given R

$$R\ (hA_i * hRep_i)\ (hA'_i * hRep'_i)$$

for all i, and $dom(hOut) = dom(hOut')$, and
$\forall \ell \in dom(hOut)\,.\,\mathcal{R}\ ((loctype\ \ell)\ state)\ (h\ell)\ (h'\ell)$.
For other categories, we define $\mathcal{R}\,\theta$ as follows.

$$\begin{array}{lll}
\mathcal{R}\ \texttt{bool}\ d\ d' & \Leftrightarrow & d = d' \\
\mathcal{R}\ \texttt{unit}\ d\ d' & \Leftrightarrow & d = d' \\
\mathcal{R}\ C\ \ell\ \ell' & \Leftrightarrow & \ell = \ell' \\
\mathcal{R}\ \Gamma\ \eta\ \eta' & \Leftrightarrow & \forall x \in dom\,\Gamma\,.\,\mathcal{R}\ (\Gamma x)\,(\eta x)\,(\eta' x) \\
\mathcal{R}\ (C\ state)\ s\ s' & \Leftrightarrow & \\
\quad \forall f \in fields\,C\,.\,\mathcal{R}\ (type(f,C))\ (s\ f)\ (s'\ f), \quad \text{for } C \not\leq A & & \\
\mathcal{R}\ (C, (\overline{x} : \overline{T}) \to T)\ d\ d' & \Leftrightarrow & \forall \overline{d}, \overline{d'}, h, h', \ell\,. \\
\quad (\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'} \wedge \mathcal{R}\ Heap\ h\ h' \wedge conf\ C\ \eta\ h \wedge conf\ C\ \eta'\ h') & & \\
\quad \Rightarrow \mathcal{R}\ (T \times Heap)_\perp\ (d\eta h)\ (d'\eta' h') & & \\
\quad \text{where } \eta = [\overline{x} \mapsto \overline{d}, this \mapsto \ell],\ \eta' = [\overline{x} \mapsto \overline{d'}, this \mapsto \ell] & & \\
\mathcal{R}\ MEnv\ \mu\ \mu' & \Leftrightarrow & \text{for all non-rep } C \\
\quad \forall m\,.\,\mathcal{R}\ (C, mtype(m,C))\ (\mu\,C\,m)\ (\mu'\,C\,m) & & \\
\mathcal{R}\ (\theta_\perp)\ \alpha\ \alpha' & \Leftrightarrow & \\
\quad (\alpha = \perp = \alpha') \vee (\alpha \neq \perp \neq \alpha' \wedge \mathcal{R}\ \theta\ \alpha\ \alpha') & &
\end{array}$$

Note that we do not need to define $\mathcal{R}\ (C\ state)\ s\ s'$ for $C \leq A$; the relation on states is only used in defining the relation on heaps, and there only for non-rep $C \not\leq A$.

FACT 3.3. *For all h, h' and $\ell \notin (locs(Rep\downarrow) \cup locs(Rep'\downarrow))$, if $\mathcal{R}$ Heap h h' then $\ell \in dom\,h \Leftrightarrow \ell \in dom\,h'$.*

FACT 3.4. *For any data type T, $\mathcal{R}\ T$ is the identity relation. As a corollary, if $\overline{U} \leq \overline{T}$ and $\mathcal{R}\ \overline{U}\ \overline{d}\ \overline{d'}$ then $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$.*

4. THE ABSTRACTION THEOREM

THEOREM 4.1. *Suppose we are given a basic simulation, with CT, CT' confined. Suppose further that for every confined μ, μ', and for every method declaration M in $CT(A)$ (respectively M' in $CT'(A)$), we have*

$$(*)\quad \mathcal{R}\ MEnv\ \mu\ \mu' \Rightarrow \mathcal{R}\ (A, mtype(m,A))\ ([\![M]\!]\mu)\ ([\![M']\!]'\mu')$$

Then $\mathcal{R}\ (C, mtype(m,C))\ (\hat{\mu}\,C\,m)\ (\hat{\mu}'\,C\,m)$ for every m and non-rep C, that is, $\mathcal{R}\ MEnv\ \hat{\mu}\ \hat{\mu}'$, where $\hat{\mu}, \hat{\mu}'$ are the fixpoints for CT, CT'.

PROOF. First, we show by induction that for every i we have $\mathcal{R}\ MEnv\ \mu_i\ \mu'_i$. For the base case, for every C, m we have $\mathcal{R}\ (C, mtype(m,C))\ (\mu_0\,C\,m)\ (\mu'_0\,C\,m)$ because $\perp$ relates to $\perp$. For the induction step, suppose $\mathcal{R}\ MEnv\ \mu_i\ \mu'_i$. We consider cases on whether $C = A$. For $C \neq A$, we apply Lemma 4.2, below, to μ_i, μ'_i. For $C = A$ and m declared in A, we have $\mathcal{R}\ (A, mtype(m,A))\ (\mu_{i+1}\,A\,m)\ (\mu'_{i+1}\,A\,m)$ by hypothesis $(*)$ and definition of μ_{i+1}. For m inherited in A from $B \geq A$, we use Lemma 3.1 and the definition of $\mathcal{R}$ to show that $\mathcal{R}\ (A, mtype(m,A))\ (\mu_{i+1}\,A\,m)\ (\mu'_{i+1}\,A\,m)$ follows from $\mathcal{R}\ (B, mtype(m,A))\ (\mu_{i+1}\,B\,m)\ (\mu'_{i+1}\,B\,m)$.

Now, $\hat{\mu}, \hat{\mu}'$ are the least upper bounds of the chains we just showed are related. It remains to show that

$$\forall i\ .\ \mathcal{R}\ (C, mtype(m,C))\ (\mu_i\,C\,m)\ (\mu'_i\,C\,m)$$

implies $\mathcal{R}\ (C, mtype(m,C))\ (\sqcup_i \mu_i\,C\,m)\ (\sqcup_i \mu_i{}'\,C\,m)$. This is routine. □

To serve as an induction hypothesis, the statement of the lemma is a little complicated. In essence, it says that each constituent expression e and command S of each method outside class A preserves the relation.

LEMMA 4.2. *Suppose a basic simulation is given, such that CT and CT' are confined. Then, for all non-rep classes $C \neq A$, all constituent expressions $\Gamma;\ C \vdash e : T$ and commands $\Gamma;\ C \vdash S$: `com` in methods declared in C, and all confined μ, μ', the following holds. For all confined heaps h (resp. h') and environments η confined for C, h (resp. η' for C, h'), if $\mathcal{R}$ MEnv μ μ', $\mathcal{R}$ Heap h h', and $\mathcal{R}\ \Gamma\ \eta\ \eta'$ then*

$$\mathcal{R}\ (T_\perp)\ ([\![\Gamma;\ C \vdash e : T]\!]\mu\eta h)\ ([\![\Gamma;\ C \vdash' e : T]\!]'\mu'\eta' h')$$
$$\mathcal{R}\ (\Gamma \times Heap)_\perp\ ([\![\Gamma;\ C \vdash S]\!]\mu\eta h)\ ([\![\Gamma;\ C \vdash' S]\!]'\mu'\eta' h')$$

PROOF. For lack of space, we consider only the case of method call as a command, using identifiers as in the semantic definition in Table 4 (and their primed counterparts). By $\mathcal{R}\ \Gamma\ \eta\ \eta'$, we have $\mathcal{R}\ (\Gamma\ x)\ \ell\ \ell'$, hence $\ell = \ell'$ by Fact 3.4. By *conf* $C\ \eta\ h$ and $C \neq A$ we have $\ell \notin locs(Rep\downarrow)$ and $\ell \notin locs(Rep'\downarrow)$. Hence, by Fact 3.3, if $\ell \notin dom\ h$ then $\ell \notin dom\ h'$, in which case both semantics are $\perp$ and $\mathcal{R}\ (\Gamma \times Heap)_\perp\ \perp\ \perp$. It remains to consider the case $\ell \in dom\ h \cap dom\ h'$.

By induction on $\overline{e}$, either $[\![\Gamma;\ C \vdash \overline{e} : \overline{U}]\!]\mu\eta h = \perp$ and $[\![\Gamma;\ C \vdash \overline{e} : \overline{U}]\!]'\mu'\eta' h' = \perp$, and thus we get the result because $\mathcal{R}\ (\Gamma \times Heap)_\perp\ \perp\ \perp$, or neither is $\perp$. In the latter case, let $\eta_0 = [\overline{x} \mapsto \overline{d}, this \mapsto \ell]$ (resp. η'_0) where $\overline{d}, \overline{d'}$ are as in the semantic definition. Since $\mathcal{R}$ *MEnv* μ μ', we have $\mathcal{R}\ ((loctype\ \ell), (\overline{x} : \overline{T}) \to T)\ (\mu(loctype\ \ell)m)\ (\mu'(loctype\ \ell)m)$ by definition $\mathcal{R}(MEnv)$. Since $\mathcal{R}$ *Heap* h h' and assuming we can show *conf* $(loctype\ \ell)\ \eta_0\ h$ (resp. *conf* $(loctype\ \ell)\ \eta'_0\ h'$) and $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$, we have
$\mathcal{R}\ (T \times Heap)_\perp\ (\mu(loctype\ \ell)\eta_0 h)\ (\mu(loctype\ \ell)\eta'_0 h')$, by definition $\mathcal{R}((loctype\ \ell), (\overline{x} : \overline{T}) \to T)$ and $loctype\ \ell \not\leq Rep$. Now, either $\mu(loctype\ \ell)\eta_0 h = \perp = \mu(loctype\ \ell)\eta'_0 h'$, in which case we have related results, or $\mathcal{R}$ *Heap* $h_0\ h'_0$, whence $\mathcal{R}\ (\Gamma \times Heap)_\perp\ (\eta, h_0)\ (\eta', h'_0)$, where h_0, h'_0 are the heaps returned by $\mu(loctype\ \ell)\eta_0 h, \mu(loctype\ \ell)\eta'_0 h'$.

It remains to show $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$ and *conf* $(loctype\ \ell)\ \eta_0\ h$ (resp. *conf* $(loctype\ \ell)\ \eta'_0\ h'$), for which we go by cases on $loctype\ \ell$. Recall that $\ell \notin locs(Rep\downarrow)$, so $(loctype\ \ell) \not\leq Rep$.

(i) $loctype\ \ell \neq A$, i.e., either $loctype\ \ell < A$ or $loctype\ \ell \not\leq A$. In either case, since CT is confined, by condition (iii)(a)

or (iii)(b) in the definition of confined CT, we know $\overline{T} \not\leqslant Rep$. By induction on $\overline{e}$, we get $\mathcal{R}\ \overline{U}\ \overline{d}\ \overline{d'}$ (as we are considering the non-$\bot$ case). Thus, by $\overline{U} \leq \overline{T}$ and the corollary to Fact 3.4, $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$. Because $\overline{T} \not\leqslant Rep$, we have, $\overline{d} \notin locs(Rep\downarrow)$.

Now $conf\ (loctype\ \ell)\ \eta_0\ h \Leftrightarrow rng\ \eta_0 \cap locs(Rep\downarrow) = \varnothing$. And $rng\ \eta_0 \cap locs(Rep\downarrow) = \{\overline{d}, \ell\} \cap locs(Rep\downarrow) = \varnothing$, since $\overline{d} \notin locs(Rep\downarrow)$ and $\ell \notin locs(Rep\downarrow)$.

(ii) $loctype\ \ell = A$: Hence $\ell \in locs(A\downarrow)$. Since CT is confined, by condition (iii)(a) in the definition of confined CT, we know $\overline{T} \not\leqslant A$ and $\overline{T} \not\leqslant Rep$. As in case (i), induction on $\overline{e}$ yields $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$. Because $\overline{T} \not\leqslant A$ and $\overline{T} \not\leqslant Rep$, we have, $\overline{d} \notin locs(A\downarrow)$ and $\overline{d} \notin locs(Rep\downarrow)$.

Now $conf\ (loctype\ \ell)\ \eta_0\ h \Leftrightarrow rng\ \eta_0 \cap (locs(Rep\downarrow) \cup locs(A\downarrow)) \subseteq dom(hA_j * hRep_j)$ for some j where $dom(hA_j) = \{\eta_0\ this\} = \{\ell\}$. And $rng\ \eta_0 \cap (locs(Rep\downarrow) \cup locs(A\downarrow)) = \{\overline{d}, \ell\} \cap (locs(Rep\downarrow) \cup locs(A\downarrow)) = \{\ell\} \subseteq dom(hA_j * hRep_j)$, for some j and $dom(hA_j) = \{\eta_0\ this\} = \{\ell\}$. □

Identity extension. The abstraction theorem is usually used in conjunction with an identity extension lemma. A typical formulation says that $\mathcal{R}\,T$ is the identity on any type T for which it is the identity on all base types b that occur in T. The reason is that no value of type b can occur in a value of type T if b does not occur in T. This fails with extensible records and structural subtyping, and with procedures that may have global variables [28]. It can be made to work using name-based (declaration) subclassing [6], but it turns out that for our purposes it is enough to deal with the heap.

In our language, $\mathcal{R}\ T$ is the identity for every data type T (Fact 3.4), but that is only because the interesting data is in the heap —which is not typed at all.[6] In general, $[\![A\ state]\!] \neq [\![A\ state]\!]'$ and $\mathcal{R}(A\ state)$ is not the identity. Related heaps can contain $A\downarrow$ objects with different states that may point to completely different $Rep\downarrow$ objects. But consider executing a method on an object o from whose fields no A objects are reachable, i.e., A objects are not part of the representation of o. The resulting heap may contain A objects that were assigned to local variables, but if the method is confined then those objects are unreachable in the final state. Enter garbage collection. For a set or list $\overline{d}$ of values, define the heap $collect(\overline{d}, h)$ to be the restriction of h to cells reachable from elements of $\overline{d}$. Say h is *A-free* just if $dom\,h \cap locs(A\downarrow) = \varnothing$.

LEMMA 4.3. *Suppose* $\mathcal{R}\ (\Gamma \times Heap)\ (\eta, h)\ (\eta', h')$. *If both* $collect((rng\ \eta), h)$ *and* $collect((rng\ \eta'), h')$ *are A-free then* $collect((rng\ \eta), h) = collect((rng\ \eta'), h')$.

PROOF. Related heaps are confined, by definition $\mathcal{R}\ Heap$. In confined heaps, $Rep\downarrow$ objects are only reachable from $A\downarrow$ objects. Now the argument is a straightforward induction using the definition of $\mathcal{R}$. □

An A-free heap in $[\![Heap]\!]$ is also an element of $[\![Heap]\!]'$, and $[\![\Gamma]\!] = [\![\Gamma]\!]'$ for any Γ. For any $\mathcal{R}$, a confined A-free h has no $Rep\downarrow$-objects, and $\mathcal{R}\ Heap\ h\ h$.

To compare commands interpreted with respect to CT and CT', we say commands $\Gamma;\ C \vdash S$: `com` and $\Gamma;\ C \vdash' S'$: `com` are *equivalent* iff the following holds, where d is the semantics of S in CT and d' the semantics of S' in CT': For every confined A-free heap h and every $\eta \in [\![\Gamma]\!]$ with $conf\ C\ \eta\ h$, either $d\,\eta\,h = \bot = d'\,\eta\,h$ or neither is $\bot$ and moreover $\eta_0 = \eta'_0$ and $collect((rng\ \eta_0), h_0) = collect((rng\ \eta_0), h'_0)$, where $(\eta_0, h_0) = d\,\eta\,h$ and $(\eta'_0, h'_0) = d'\,\eta\,h$.

Example. Consider a client class `C` with a method containing the following command, typed in some `A0`-free environment Γ with Γ `b` = `bool`.

```
var A0 z := null in
  z := new A0(); z.setg(true); b := z.getg()
```

Let d (resp. d') be the meaning of this command, using the first version (resp. primed version) of `A0` in Section 2.1. Let R be the relation described in Section 2.1 and formalized following Definition 2. It is straightforward to show that the induced relation holds between methods of the two versions of `A0`. To show that d is equivalent to d', consider any confined, `A0`-free heap h and environment η confined in C, h. We have $\mathcal{R}\ Heap\ h\ h$ and $\mathcal{R}\ \Gamma\ \eta\ \eta$, and by the abstraction theorem d and d' are related, so we get related outcomes $d\,\eta\,h$ and $d'\,\eta\,h$. If the outcome is not $\bot$ then these related environments are equal (they do not contain any `A0` locations) and the related heaps, once collected, are equal by the identity extension Lemma 4.3.

Section 2.1 gives an example command that exploits the representation object leaked by the `bad` method. For that command the abstraction theorem does not apply because that method is not confined. The two interpretations of the command are not equivalent; it acts as skip using the first version of `A0` and diverges using the primed version.

5. APPLICATION

The following example is based on one of the Meyer-Sieber equivalences [21], which we first recall in Algol syntax.

The following two commands are equivalent, because they both diverge, for any `P` that takes a command as argument.

```
var x := 0; P(x:=x+2); if x is even then diverge
```

```
var x := 0; P(x:=x+2); diverge
```

Informally, the argument is that in the first example `x` is invariably even. This is because `P` is declared somewhere not in the scope of `x` so the variable can only be affected by (possibly repeated) executions of the command `x:=x+2` and this maintains the invariant.

Now we consider an adaptation, due to Peter O'Hearn, of the example to Java. The Java version uses a private field in place of local variable `x`. Instead of passing the command `x:=x+2` as argument, the object passes a reference to itself; this gives access to a public method `inc`. Using the implementation of `A1` below, which corresponds to the first Algol example, the command

```
(*)  var C y := new C() in
     var A1 x := new A1() in x.callP(y)
```

[6] Nor would we want to impose a typing system on the heap, as it would likely preclude unbounded data structures [13].

diverges because after calling `y.P`, method `callP` diverges.

```
class A1 extends Object    {
  int g; /* initially 0 */
  unit callP(C y){
    y.P(this);
    if (isEven(g)) then diverge else skip }
  unit inc(){ g := g + 2 } }
```

Here is an implementation for which $(*)$ corresponds to the second Algol example.

```
class A1 extends Object      {
  int g'; /* initially 0 */
  unit callP(C y){ y.P(this); diverge }
  unit inc(){ g' := g' + 2 } }
```

Both interpretations of $(*)$ are equivalent, provided that any overriding declarations for `inc` preserve its invariant (behavioral subclassing [19, 10]), and similarly for `callP`.

PROPOSITION 5.1. *The command $(*)$ has the same meaning with either of the implementations for `A1`, provided that the corresponding class tables are confined, `callP` is not overridden, and any declaration overriding `inc` maintains evenness.*

To prove equivalence we choose Rep, Rep' to be arbitrary classes (unusable by clients but not needed by `A1`), as they are not relevant here. Let the basic simulation R relate h to h' just if they have the same singleton domain ℓ with *loctype* $\ell \leq$ `A1` and $h\,\ell\,g = h'\,\ell\,g' = 2 \times m$ for some $m \geq 0$.

The proposition can be proved without using the identity extension Lemma 4.3, but that is only because the example uses divergence to make it obvious that the two commands are equivalent. Identity extension is needed for other variations. For example, we can make $(*)$ behave as skip by replacing the first implementation of `callP` with

```
y.P(this); if(isEven(g))then skip else diverge
```

and the second implementation with `y.P(this); skip`. Because `x` is local in $(*)$, there are no reachable `A1`-objects in the final states, so the outcomes from the two versions of $(*)$ are equal.

The preceding examples do not involve representation objects or confinement. What they confirm is that in our language local variables and private fields provide first-class stateful procedures in a form that does not have the problematic interactions found in Algol. The same is true of some other conventional languages (e.g., C has no nested procedure declarations, and Oberon does not allow local procedures to be passed as arguments), making it possible to prove the Meyer-Sieber equivalences in simple models [28] for those languages.

It is straightforward to extend the examples to include representation objects. For example, the integer field `g` can be replaced by an `Integer` object similar to the `Boolean` object used in the example of Section 2.1. Then confinement is needed just as in that example.

6. PRIVILEGE-BASED ACCESS CONTROL

In the Java access control mechanism [11], each class has an associated principal, the signer of the class (e.g., the user, or an authenticated remote site from which the code was obtained). An access control matrix associates with each principal n a set $\mathcal{A}(n)$ of privileges, or resources, for which it is authorized. For example, a user program might have the privilege p to change their password but not the privilege w for directly writing the password file. The stack frame for a method activation is marked with the signer of the class file declaring the method, and also contains a set of enabled privileges. (Enabled privileges need not be authorized for the signer.) Let us write $\langle n, P\rangle$ for such frame, where $n \in \mathsf{Principals}$ and $P \in \mathcal{P}(\mathsf{Privs})$. Privileges must be explicitly enabled by an operation called `doPrivileged`, which adds the privilege to the current frame. Before executing an operation, like hardware `write`, that is to be protected, a check (`checkPermission`) is performed. This "stack inspection" decides $\mathsf{chk}(p, S)$, which is defined as follows, where we write $\langle n, P\rangle :: S$ for $\langle n, P\rangle$ on top of stack S. For the empty stack, $\mathsf{chk}(p, nil) \Leftrightarrow \mathsf{false}$. For nonempty stacks, $\mathsf{chk}(p, (\langle n, P\rangle :: S)) \Leftrightarrow p \in \mathcal{A}(n) \wedge (p \in P \vee \mathsf{chk}(p, S))$.

Here are two example classes in our syntax.

```
class Sys signer sys extends Object          {
  unit writepass(String x){
    check w; write(x,"passfile") }
  unit passwd(String x){
    check p; dopriv w in this.writepass(x) } }

class User signer user extends Object            {
  Sys s ... /* initialization elided */
  unit use(){ dopriv p in s.passwd("mypass") }
  unit try(){ dopriv w in s.writepass("mypass") }}
```

Assume that $\mathcal{A}(user) = \{p\}$ and $\mathcal{A}(sys) = \{p, w\}$. Invoking method `use` makes a frame $\langle user, \varnothing\rangle$, and then `use` enables p; from the resulting frame $\langle user, \{p\}\rangle$, a call is made to `passwd` which pushes $\langle sys, \varnothing\rangle$. Then `check` p succeeds and privilege w gets enabled. Finally, `writepass` checks w successfully, and calls the hardware `write`.

Invocation of user method `try` results in a security exception: Although w gets added to the frame for `try`, the check by `writepass` fails because w is not authorized for *user*.

As described, the stack inspection semantics is *lazy* in that tests $p \in \mathcal{A}(n)$ are only performed when p is actually needed to perform a guarded operation. But a stack S determines a set $\mathsf{privs}\ S$ of enabled, authorized privileges: $p \in \mathsf{privs}\ S \Leftrightarrow \mathsf{chk}(p, S)$. This gives rise to an equivalent *eager* [11, 42] semantics: Instead of evaluating an expression in the context of a stack S, it maintains the context $\langle n, \mathsf{privs}\ S\rangle$, where n is the principal in the top frame of S.

Actual implementations depend on facilities built in to the runtime environment, but they are largely written as library classes. In earlier work [3], we showed equivalence of the lazy and eager strategies, using simulations and built-in semantics for the security mechanism in an idealized language (like those of [38, 42], which also show the equivalence). It is an interesting exercise to write down, in our core language, lazy and eager implementations, and show their equivalence using the abstraction theorem. This involves rewriting the source code to add a security context parameter for methods, and to invoke suitable methods on that context. What is more important is to give an accurate model including the built-in features. This is what we do below, extending the language of Section 2 and using eager semantics. Then we show that the abstraction theorem holds.

By itself, the access control mechanism merely controls which methods can be called. To ensure an intrinsic security

property such as "user programs do not write the password file", visibility control can ensure that, e.g., the `write` operation is only called by `writepass` (by making `write` or even `writepass` private). Confinement can be used to protect access to data such as real disk addresses. We show parametricity for the language including access control, which lays a basis for proofs of intrinsic security properties, but such proofs are left for future work.

The extended language. The core language in Section 2 is extended as follows:

$$\begin{array}{rcl} CL & ::= & \texttt{class}\ C\ \texttt{signer}\ n\ \texttt{extends}\ C\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\ S & ::= & \ldots \mid \texttt{check}\ p \mid \texttt{dopriv}\ p\ \texttt{in}\ S \\ e & ::= & \ldots \mid \texttt{testpriv}\ p \end{array}$$

The new typing rules are simple: For any Γ, C, the type of `testpriv` p is `bool`, and the type of `check` p is `com`. The type of `dopriv` p `in` S is `com` just if the type of S is `com`. The only addition to semantic categories θ is Sec where $[\![Sec]\!] = (\mathsf{Principals} \times \mathcal{P}(\mathsf{Privs}))$ ordered by equality. The changes in the semantic domains occur in the domains of methods, expressions and commands:

$$\begin{array}{l} [\![C, (\overline{x} : \overline{T}) \to T]\!] = \\ \quad [\![\overline{x} : \overline{T}, \mathit{this} : C]\!] \to [\![\mathit{Heap}]\!] \to [\![\mathit{Sec}]\!] \to ([\![T]\!] \times [\![\mathit{Heap}]\!])_\perp \\ [\![\Gamma;\ C \vdash e : T]\!] \in [\![\mathit{MEnv}]\!] \to [\![\Gamma]\!] \to [\![\mathit{Heap}]\!] \to [\![\mathit{Sec}]\!] \to [\![T]\!]_\perp \\ [\![\Gamma;\ C \vdash S : \texttt{com}]\!] \in \\ \quad [\![\mathit{MEnv}]\!] \to [\![\Gamma]\!] \to [\![\mathit{Heap}]\!] \to [\![\mathit{Sec}]\!] \to ([\![\Gamma]\!] \times [\![\mathit{Heap}]\!] \times [\![\mathit{Sec}]\!])_\perp \end{array}$$

We give below the interesting changes in the semantics of expressions and commands; the rest of the semantics follow *mutatis mutandis* taking $[\![Sec]\!]$ into account. Let σ range over $[\![Sec]\!]$.

$$\begin{array}{l} [\![\Gamma;\ C \vdash e.m(\overline{e}) : T]\!]\mu\eta h\sigma \\ = \ \textbf{let}\ \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h\sigma\ \textbf{in} \\ \quad \textbf{if}\ \ell \notin \mathit{dom}\, h\ \textbf{then}\ \perp\ \textbf{else} \\ \quad \textbf{let}\ d = \mu(\mathit{loctype}\ \ell)m\ \textbf{in} \\ \quad \textbf{let}\ n = \mathit{signer}(\mathit{loctype}\ \ell)\ \textbf{in} \\ \quad \textbf{let}\ \langle \_, P\rangle = \sigma\ \textbf{in} \\ \quad \textbf{let}\ \sigma_0 = \langle n, P \cap \mathcal{A}(n)\rangle\ \textbf{in} \\ \quad \textbf{let}\ \overline{d} = [\![\Gamma;\ C \vdash \overline{e} : \overline{U}]\!]\mu\eta h\sigma\ \textbf{in} \\ \quad \textbf{let}\ (h_0, d_0) = d[\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]h\sigma_0\ \textbf{in}\ d_0 \\ [\![\Gamma;\ C \vdash \texttt{testpriv}\ p : \texttt{bool}]\!]\mu\eta h\sigma \\ = \textbf{let}\ \langle \_, P\rangle = \sigma\ \textbf{in}\ (p \in P) \\ [\![\Gamma;\ C \vdash \texttt{check}\ p : \texttt{com}]\!]\mu\eta h\sigma \\ = \ \textbf{let}\ \langle \_, P\rangle = \sigma\ \textbf{in} \\ \quad \textbf{if}\ p \in P\ \textbf{then}\ (\eta, h, \sigma)\ \textbf{else}\ \perp \\ [\![\Gamma;\ C \vdash \texttt{dopriv}\ p\ \texttt{in}\ S : \texttt{com}]\!]\mu\eta h\sigma \\ = \ \textbf{let}\ \langle n, P\rangle = \sigma\ \textbf{in} \\ \quad \textbf{if}\ (p \in \mathcal{A}(n))\ \textbf{then}\ [\![\Gamma;\ C \vdash S : \texttt{com}]\!]\mu\eta h\langle n, P \cup \{p\}\rangle \\ \quad \textbf{else}\ [\![\Gamma;\ C \vdash S : \texttt{com}]\!]\mu\eta h\sigma \end{array}$$

The statement of Theorem 4.1 can be interpreted in the new semantics, taking into account the revised semantic domains. And it can be proved the same way, using the following lemma to take $[\![Sec]\!]$ into account.

LEMMA 6.1. *Suppose a basic simulation is given, such that CT and CT' are confined. Then, for all non-rep classes $C \neq A$, all constituent expressions $\Gamma;\ C \vdash e : T$ and commands $\Gamma;\ C \vdash S :$ `com` in methods declared in C, and all confined μ, μ', the following holds. For all confined heaps h (resp. h') and environments η confined for C, h (resp. η' for C, h'), and all $\sigma \in [\![Sec]\!]$, if $\mathcal{R}$ MEnv $\mu\ \mu'$, $\mathcal{R}$ Heap $h\ h'$, and $\mathcal{R}\ \Gamma\ \eta\ \eta'$ then*

$$\begin{array}{l} \mathcal{R}\ (T_\perp)\ (\ [\![\Gamma;\ C \vdash e : T]\!]\mu\eta h\sigma)([\![\Gamma;\ C \vdash' e : T]\!]'\mu'\eta' h'\sigma) \\ \mathcal{R}\ (\Gamma \times \mathit{Heap} \times \mathit{Sec})_\perp\ ([\![\Gamma; C \vdash S]\!]\mu\eta h\sigma)([\![\Gamma; C \vdash' S]\!]'\mu'\eta' h'\sigma) \end{array}$$

7. DISCUSSION

We have used instance-based confinement to show representation independence for a Java-like language including references and access control based on principals and privileges. Although we focused on Java, similar features are found in a number of other languages. As remarked in [13], there do not seem to be comparable results in the literature. Cavalcanti and Naumann [6] prove representation independence for language very similar to our core language except that they use copy semantics for assignment and do not model shared references, a central feature of conventional languages. We exploit their insights on how simple semantic domains are adequate. Their work addresses refinement calculus, and uses predicate transformer semantics in order to model specifications as constituents of method bodies.

Confinement figures heavily in the verification logics of Müller and Poetzsch-Heffter [26] and in some work by the group of Nelson and Leino [18, 9]. Whereas other formalizations of semantics for Java-like languages are oriented to verification of individual programs [14] or proofs of metatheorems such as type safety [41], these works are expressly concerned with modular checking and verification. They address encapsulation with specifications based on Leino's notion of dependency which relates abstraction to representation. Aspects of parametricity have doubtless been confronted in soundness proofs by these researchers (especially in [18, 25]). But they are ambitiously tackling many issues at once, including specifications and more sophisticated forms of confinement; we have found it difficult to glean results and proof techniques that can be used in other settings. Certainly there are no explicit results like the abstraction theorem.

Our semantics does not depend on confinement or specifications, and is presented using widely known semantic notions. We believe that our formulation of confinement and abstraction is transparent enough to invite independent confirmation and to be adapted for other uses. In ongoing work we are assessing various notions of confinement and their use for checking of security and other properties. The capability semantics of [5] might complement our work, in that it offers a means for formalizing and thus comparing various fine-grained notions of confinement.

Our result uses confinement expressed in terms of types declared in the program; one direction for future work is to explore confinement based on principals.

Acknowledgements. Peter O'Hearn, David Schmidt, and the anonymous referees offered improvements and encouragement. Hongseok Yang pointed out a faulty proof step, which led us to correct the definition of confinement.

8. REFERENCES

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, 1999.

[2] A. Banerjee and D. A. Naumann. A static analysis for instance-based confinement in Java. In preparation.

[3] A. Banerjee and D. A. Naumann. A simple semantics and static analysis for Java security. Technical Report CS Report 2001-1, Stevens Institute of Technology.

[4] J. Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6), 2001.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.

[6] A. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. Submitted, 2001.

[7] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP* 2001.

[8] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[9] D. Detlefs and K. R. M. Leino and G. Nelson. Wrestling with rep exposure. Research Report 156, COMPAQ Systems Research Center, July 1998.

[10] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE*, Berlin, 1996.

[11] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[12] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1998.

[13] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Trans. Prog. Lang. Syst.*, 22(6), 2000.

[14] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *FASE*, 2000.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *OOPSLA*, 1999.

[16] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[17] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000.

[18] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, COMPAQ Systems Research Center, Nov. 2000. To appear in *ACM Trans. Prog. Lang. Syst.*

[19] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6), 1994.

[20] N. Lynch and F. Vaandrager. Forward and backward simulations part I. *Inf. Comput.*, 121(2), 1995.

[21] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *POPL*, 1988.

[22] J. C. Mitchell. Representation independence and data abstraction. In *POPL*, 1986.

[23] J. C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, 1991.

[24] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[25] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Available from `www.informatik.fernuni-hagen.de/pi5/publications`.

[26] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[27] P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. *ECOOP Workshop on Formal Techniques for Java Programs*, Technical Report 269, Fernuniversität Hagen, 2000.

[28] D. A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Comput. Sci.*, 2001. To appear.

[29] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3), 1995.

[30] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes)*. Birkhäuser, Boston, 1997.

[31] G. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, University of Edinburgh, School of Artificial Intelligence, 1973.

[32] U. S. Reddy. Objects and classes in Algol-like languages. In *FOOL*, 1998.

[33] J. C. Reynolds. Towards a theory of type structure. In *Colloques sur la Programmation, LNCS 19*, pages 408–425, 1974.

[34] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*. North-Holland, 1981.

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

[36] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2001.

[37] C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA*, 2000.

[38] C. Skalka and S. Smith. Static enforcement of security with types. In *ICFP*, 2000.

[39] C. Strachey. Fundamental concepts in programming languages. *Higher Order and Symbolic Computation*, 13(1), 2000. Originally appeared in 1967 Lecture notes, International Summer School in Computer Programming, Copenhagen.

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

[41] D. von Oheimb, T. Nipkow, and C. Pusch. muJava: Embedding a programming language in a theorem prover. In *Proceedings of the International Summer School Marktoberdorf*, 1999.

[42] D. Wallach, A. Appel, and E. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Software Eng. Method.*, 9(4), 2000.

APPENDIX

A. PROOF OF LEMMA 3.1

LEMMA. For nonrep C, B, if $C \leq B$ and *conf* $C\ \eta\ h$ then *conf* $B\ \eta\ h$, for all environments η for parameters of methods of B.

PROOF. In the cases $C \leq B \leq A$ and $C \not\leq A$, the result holds because *conf* $C\ \eta\ h \Leftrightarrow$ *conf* $B\ \eta\ h$ by definition of *conf* .

In the case of $C \leq A < B$, we have to take care because *conf* $A\ \eta\ h$ allows $rng\,\eta$ to include *Rep* locations whereas *conf* $B\ \eta\ h \Leftrightarrow rng\,\eta \cap locs(Rep\downarrow) = \varnothing$. This is why the lemma includes a hypothesis that η is for the parameters of some method of B. The method is inherited in A, so item (iii)(a) in the definition of confinement for CT disallows parameters of type $\leq$ *Rep* or $\geq$ *Rep*. Thus $rng\,\eta$ can have no *Rep* locations.

B. PROOF OF LEMMA 3.2

LEMMA. Suppose CT is confined. Then the method environment $\hat{\mu}$ obtained as a fixpoint is confined, as is each μ_i in the approximation chain.

PROOF. Confinement of $\hat{\mu}$ follows from confinement of each μ_i in the approximation chain. Thus, by definition of confinement for method environments, it suffices to show that for all $i \geq 0$, all non-rep C, and all m, h, η

$$\begin{array}{l} \mathit{conf}\ h \wedge \mathit{conf}\ C\ \eta\ h \wedge \mu_i C m \eta h \neq \bot \\ \Rightarrow \textbf{let}\ (d, h_0) = \mu_i C m \eta h\ \textbf{in}\ \mathit{conf}\ h_0 \wedge d \notin locs(Rep\downarrow) \end{array}$$

This we prove by induction on i.

Case $i = 0$. By definition of μ_0 we have $\mu C m \eta h = \bot$ which falsifies the antecedent of the implication.

Case $i > 0$. The definition of $\mu_i\, C\, m$ is by cases on whether m is inherited or declared in C.

Suppose m is declared in C as $M = Tm(\overline{x} : \overline{T})\{S; \texttt{return}\ e\}$, so that $\mu_i\, C\, m = [\![M]\!]\mu_{i-1}$. By induction, μ_{i-1} is confined. Suppose *conf* h, *conf* $C\ \eta\ h$, and $\mu C m \eta h \neq \bot$. Let $(\eta_0, h_0) = [\![S]\!]\mu_{i-1}$. Because μ_{i-1} is confined, by the induction hypothesis, and S is confined, by the hypothesis that CT is confined, we have *conf* h_0 and *conf* $C\ \eta_0\ h_0$. Let $d = [\![e]\!]\mu_{i-1}\eta_0 h_0$. If $C \neq A$ we have $d \notin locs(Rep\downarrow)$ by confinement of e. If $C = A$ we have $d \notin locs(Rep\downarrow)$ owing to the syntactic constraint (iii)(a) on return types for methods of A in the definition of confinement for CT.

Suppose m is inherited in C from class $B > C$, so that $\mu_i C m = \mu_i B m$. Suppose *conf* h, *conf* $C\ \eta\ h$, and $\mu C m \eta h \neq \bot$. By Lemma 3.1 we have *conf* $B\ \eta\ h$, so the result follows by confinement of $\mu_i B m$. (Strictly speaking we are using a secondary induction on inheritance chains.)

C. ADDITIONAL CASES FOR LEMMA 4.2

LEMMA. Suppose a basic simulation is given, such that CT and CT' are confined. Then, for all non-rep classes $C \neq A$, all constituent expressions $\Gamma;\ C \vdash e : T$ and commands $\Gamma;\ C \vdash S : \texttt{com}$ in methods declared in C, and all confined μ, μ', the following holds. For all confined heaps h (resp. h') and environments η confined for C, h (resp. η' for C, h'), if $\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu'$, $\mathcal{R}\ \mathit{Heap}\ h\ h'$, and $\mathcal{R}\ \Gamma\ \eta\ \eta'$ then

$$\mathcal{R}\ (T_\perp)\ ([\![\Gamma;\ C \vdash e : T]\!]\mu\eta h)\ ([\![\Gamma;\ C \vdash' e : T]\!]'\mu'\eta'h')$$
$$\mathcal{R}\ (\Gamma \times \mathit{Heap})_\perp\ ([\![\Gamma;\ C \vdash S]\!]\mu\eta h)\ ([\![\Gamma;\ C \vdash' S]\!]'\mu'\eta'h')$$

PROOF. For expressions, the proof goes by induction on $\Gamma;\ C \vdash e : T$, written "induction on e" in the sequel. For commands, the proof goes by induction on $\Gamma;\ C \vdash S : \texttt{com}$, using the result for expressions. We adopt the convention that primed variables will refer to semantic objects in the primed semantics, $[\![.]\!]'$. For clarity, we recall both the unprimed and primed semantics for the case of field update only.

Field access: For the unprimed semantics, $[\![.]\!]$, the semantics of $e.f$ is

$$\begin{array}{ll} & [\![\Gamma;\ C \vdash e.f : T]\!]\mu\eta h \\ = & \textbf{let}\ \ell = [\![\Gamma;\ C \vdash e : C]\!]\mu\eta h\ \textbf{in} \\ & \quad \textbf{if}\ \ell \notin dom\,h\ \textbf{then}\ \bot\ \textbf{else}\ h\ell f \end{array}$$

For the primed semantics, $[\![.]\!]'$, the semantics of $e.f$ is

$$\begin{array}{ll} & [\![\Gamma;\ C \vdash' e.f : T]\!]'\mu'\eta'h' \\ = & \textbf{let}\ \ell' = [\![\Gamma;\ C \vdash' e : C]\!]'\mu'\eta'h'\ \textbf{in} \\ & \quad \textbf{if}\ \ell' \notin dom\,h'\ \textbf{then}\ \bot\ \textbf{else}\ h'\ell' f \end{array}$$

By induction on e and definition of $\mathcal{R}\ C_\perp$, either $[\![\Gamma;\ C \vdash e : C]\!]\mu\eta h = \bot$ and $[\![\Gamma;\ C \vdash' e : C]\!]'\mu'\eta'h' = \bot$ or neither denotation of e is $\bot$. If both are $\bot$ then we have the result, $\mathcal{R}\ T_\perp\ \bot\ \bot$, by definition of $\mathcal{R}$. If neither is $\bot$ then we have $\mathcal{R}\ C_\perp\ \ell\ \ell'$, hence $\ell = \ell'$ by Fact 3.4. Now, e is confined, by $C \neq A$ and (i) in the definition of confinement for CT. By confinement of e we have $\ell \notin locs(Rep\downarrow) \cup locs(Rep'\downarrow)$. Thus by Fact 3.3 we have $\ell \in dom\,h \Leftrightarrow \ell \in dom\,h'$. If $\ell \notin dom\,h$ then both semantics of $e.f$ are $\bot$ and then $\mathcal{R}\ (T_\perp)\ \bot\ \bot$.

For $\ell \in dom\,h$, we consider cases on whether $C < A$. Consider admissible partitions $(hOut * hA_1 * hRep_1 \ldots) = h$ and $(hOut' * hA'_1 * hRep'_1 \ldots) = h'$ that correspond as defined for $\mathcal{R}\ \mathit{Heap}$. In the case $C < A$, we have $\ell \in locs(A\downarrow)$ and hence ℓ in some $dom(hA_i)$. From $\mathcal{R}\ \mathit{Heap}\ h\ h'$ we have

$$R\ (hA_i * hRep_i)\ (hA'_i * hRep'_i)$$

and thus $\ell \in dom(hA'_i)$ by basic simulation. Since $C \neq A$, we know by visibility that f is not in the private fields $\overline{g}$ of A. Thus, as $type(f, loctype\ \ell)) = T$, we have $\mathcal{R}\ T\ (h\ell f)\ (h'\ell f)$ by basic simulation.

Finally, in the case $C \not\leq A$ (recall that C is non-rep and $C \neq A$ in the Lemma), we have $\ell \in dom(hOut)$ and hence $\ell \in dom(hOut')$ by definition $\mathcal{R}\ \mathit{Heap}$. Hence

$$\mathcal{R}\ ((loctype\ \ell)\ state)\ (h\ell)\ (h'\ell)$$

and thus $\mathcal{R}\ T\ (h\ell f)\ (h'\ell f)$ by definition of $\mathcal{R}\ ((loctype\ell)\ state)$.

Variable x**:** We have $[\![\Gamma;\ C \vdash x : T]\!]\mu\eta h = \eta x$. By hypothesis $\mathcal{R}\ \Gamma\ \eta\ \eta'$ we have $\mathcal{R}\ (\Gamma\, x)\ (\eta\, x)\ (\eta'\, x)$ and thus, as $\Gamma\, x = T$ by typing, $\mathcal{R}\ T_\perp\ (\eta\, x)\ (\eta'\, x)$.

Null: We have $[\![\Gamma;\ C \vdash \texttt{null} : B]\!]\mu\eta h = \mathsf{nil}$, and $\mathcal{R}\ B_\perp\ \mathsf{nil}\ \mathsf{nil}$ by definition.

Equality test $e_1 == e_2$**:** Letting d_1, d_2 be as in the semantic definition, we have $\mathcal{R}\ T_\perp\ d_1\ d'_1$ by induction on e_1 and $\mathcal{R}\ T_\perp\ d_2\ d'_2$ by induction on e_2. If either d_1 or d_2 is $\bot$ then that is the denotation of $e_1 == e_2$ and we have $\mathcal{R}\ \texttt{bool}_\perp\ \bot\ \bot$. Otherwise, from $\mathcal{R}\ T_\perp\ d_1\ d'_1$ and $\mathcal{R}\ T_\perp\ d_2\ d'_2$ we have $d_1 = d'_1$ and $d_2 = d'_2$ by Fact 3.4. Hence $d_1 = d_2$ iff $d'_1 = d'_2$ and thus, since $\mathcal{R}\ \texttt{bool}$ is the identity by Fact 3.4, we have $\mathcal{R}\ \texttt{bool}_\perp\ (d_1 = d_2)\ (d'_1 = d'_2)$.

(Note that the semantic definition is given using a harmless confusion between the data type $[\![\texttt{bool}]\!]$ and booleans in the meta-theory. The same is true for semantics of type casts and tests.)

Method call as an expression: First we recall the semantics:

$$\begin{array}{l}[\![\Gamma;\ C \vdash e.m(\overline{e}) : T]\!]\mu\eta h\\ =\ \textbf{let}\ \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h\ \textbf{in}\\ \quad \textbf{if}\ \ell \notin dom\, h\ \textbf{then}\ \bot\ \textbf{else}\\ \quad \textbf{let}\ (\overline{x} : \overline{T}) \to T = mtype(m, D)\ \textbf{in}\\ \quad \textbf{let}\ d = \mu(loctype\ \ell)m\ \textbf{in}\\ \quad \textbf{let}\ \overline{d} = [\![\Gamma;\ C \vdash \overline{e} : \overline{U}]\!]\mu\eta h\ \textbf{in}\\ \quad \textbf{let}\ (d_0, h_0) = d[\overline{x} \mapsto \overline{d}, this \mapsto \ell]h\ \textbf{in}\ d_0\end{array}$$

By induction on e, either
$[\![\Gamma;\ C \vdash e : D]\!]\mu\eta h = \bot = [\![\Gamma;\ C \vdash' e : D]\!]\mu'\eta'h'$ in which case both semantics of $e.m(\overline{e})$ yield $\bot$, or neither denotes $\bot$. In the latter case, we have by induction on e that $\mathcal{R}\ D\ \ell\ \ell'$, so $\ell = \ell'$ by Fact 3.4. And, by $C \neq A$ and confinement condition (i) for CT and CT', e is confined and hence $\ell \notin locs(Rep\downarrow) \cup locs(Rep'\downarrow)$. Now, if $\ell \notin dom\, h$ then, by Fact 3.3, $\ell \notin dom\, h'$ and both semantics are $\bot$. Otherwise, $\ell \in dom\, h$ and $\ell \in dom\, h'$. Let

$$\eta_0 = [\overline{x} \mapsto \overline{d}, this \mapsto \ell] \qquad \eta_0' = [\overline{x} \mapsto \overline{d'}, this \mapsto \ell]$$

From $\mathcal{R}\ MEnv\ \mu\ \mu'$ we have

$$\mathcal{R}\ ((loctype\ \ell), (\overline{x} : \overline{T}) \to T)\ (\mu(loctype\ \ell)m)\ (\mu'(loctype\ \ell)m)$$

By induction on $\overline{e}$, we get $\mathcal{R}\ \overline{U}\ \overline{d}\ \overline{d'}$ and thus $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$ by the corollary of Fact 3.4. Moreover, we have $\mathcal{R}\ Heap\ h\ h'$ by hypothesis. We claim that $conf\ (loctype\ \ell)\ \eta_0\ h$ and $conf\ (loctype\ \ell)\ \eta_0'\ h'$. Then, by the displayed relation we get

$$\mathcal{R}\ (T \times Heap)_\bot\ (\mu(loctype\ \ell)m\eta_0 h)\ (\mu'(loctype\ \ell)m\eta_0'h')$$

So, either $\mu(loctype\ \ell)m\eta_0 h = \bot = \mu'(loctype\ \ell)m\eta_0'h'$ or neither is $\bot$. In the latter case, let $(d_0, h_0) = \mu(loctype\ \ell)m\eta_0 h$; we have $\mathcal{R}\ (T \times Heap)_\bot\ (d_0, h_0)\ (d_0', h_0')$ and hence $\mathcal{R}\ T_\bot\ d_0\ d_0'$.

It remains to prove the claim. The argument is very similar to the one in the case of method call as command, in the main body of the paper.

Cast expression:

$$\begin{array}{l}[\![\Gamma;\ C \vdash (B)\ e : B]\!]\mu\eta h\\ =\ \textbf{let}\ \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h\ \textbf{in}\\ \quad \textbf{if}\ \ell \in dom\, h \wedge loctype\ \ell \leq B\ \textbf{then}\ \ell\ \textbf{else}\ \bot\end{array}$$

Induction on e yields that $\mathcal{R}\ D_\bot\ \ell\ \ell'$ or else both denotations of e are $\bot$. In the latter case, we have the result, as $\mathcal{R}\ B_\bot\ \bot\ \bot$. Otherwise, from confinement condition (i) on class tables, e is confined. Thus, as $C \neq A$, $\ell \notin locs(Rep\downarrow)$, which implies by Fact 3.3 that $\ell \in dom h \Leftrightarrow \ell' \in dom h'$. Moreover, $\ell' = \ell$ by Fact 3.4. Hence either both semantics yield ℓ, whence $\mathcal{R}\ B_\bot\ \ell\ \ell$, or both yield $\bot$ and again $\mathcal{R}\ B_\bot\ \bot\ \bot$.

Type test:

$$\begin{array}{l}[\![\Gamma;\ C \vdash e\ \texttt{instanceof}\ B : \texttt{bool}]\!]\mu\eta h\\ =\ \textbf{let}\ \ell = [\![\Gamma;\ C \vdash e : D]\!]\mu\eta h\ \textbf{in}\\ \quad \ell \in dom\, h \wedge loctype\ \ell \leq B\end{array}$$

The argument is similar to that for type cast. In the non-$\bot$ case, we have $\ell \in dom\, h \wedge loctype\ \ell \leq B$ iff $\ell' \in dom\, h' \wedge loctype\ \ell' \leq B$ and then the result follows because $\mathcal{R}\ \texttt{bool}$ is the identity.

Assignment:

$$\begin{array}{l}[\![\Gamma;\ C \vdash x := e : \texttt{com}]\!]\mu\eta h\\ =\ \textbf{let}\ d = [\![\Gamma;\ C \vdash e : T]\!]\mu\eta h\ \textbf{in}\ ([\eta \mid x \mapsto d], h)\end{array}$$

Induction on e yields that $\mathcal{R}\ T_\bot\ d\ d'$ unless both denotations of e are $\bot$. From $\mathcal{R}\ \Gamma\ \eta\ \eta'$ we get $\mathcal{R}\ \Gamma\ [\eta \mid x \mapsto d]\ [\eta' \mid x \mapsto d']$ and then, using the hypothesis for h, h',
$\mathcal{R}\ (\Gamma \times Heap)_\bot\ ([\eta \mid x \mapsto d], h)\ ([\eta' \mid x \mapsto d'], h')$.

Field Update:

$$\begin{array}{l}[\![\Gamma;\ C \vdash x.f := e : \texttt{com}]\!]\mu\eta h\\ =\ \textbf{let}\ \ell = \eta x\ \textbf{in}\ \textbf{if}\ \ell \notin dom\, h\ \textbf{then}\ \bot\ \textbf{else}\\ \quad \textbf{let}\ d = [\![\Gamma; C \vdash e : U]\!]\mu\eta h\ \textbf{in}\ (\eta, [h \mid \ell \mapsto [h\ell \mid f \mapsto d]])\end{array}$$

We consider the non-$\bot$ case. By assumption $\mathcal{R}\ \Gamma\ \eta\ \eta'$, we have $\mathcal{R}\ (\Gamma x)\ \ell\ \ell'$, hence $\ell = \ell'$ by Fact 3.4. By hypothesis $conf\ C\ \eta\ h$ we have $\ell \notin locs(Rep\downarrow)$, and by confinement of η' we have $\ell \notin locs(Rep'\downarrow)$. Hence by Fact 3.3 we get, $\ell \in dom\, h \Leftrightarrow \ell \in dom\, h'$, so either both semantics are $\bot$ due to memory fault or neither is. We consider the case $\ell \in dom\, h \wedge \ell \in dom\, h'$. By induction on e we have $\mathcal{R}\ U\ d\ d'$ and hence $\mathcal{R}\ T\ d\ d'$ by the corollary of Fact 3.3 (where $T\ f \in dfields\, C$ as in the typing rule).

To conclude the arguement it suffices to show
$\mathcal{R}\ Heap\ [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]\ [h' \mid \ell \mapsto [h'\ell \mid f \mapsto d']]$ because $\mathcal{R}\ \Gamma\ \eta\ \eta'$ holds by assumption. Consider admissible partitions $(hOut * hA_1 * hRep_1 \dots) = h$ and $(hOut' * hA_1' * hRep_1' \dots) = h'$ that correspond as in the definition of $\mathcal{R}\ Heap\ h\ h'$.

Case $C < A$: From typing we have $\Gamma\, x = C$ and hence there is some i with $\{\ell\} = dom(hA_i)$ and by $\mathcal{R}\ Heap\ h\ h'$

$$R\ (hA_i * hRep_i)\ (hA_i' * hRep_i')$$

and so $\{\ell\} = dom(hA_i')$. By typing and $C \neq A$, field f is not in the private fields $\overline{g}$ of A. So $\mathcal{R}\ Heap\ [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]\ [h' \mid \ell \mapsto [h'\ell \mid f \mapsto d']]$ follows from $\mathcal{R}\ Heap\ h\ h'$ and $\mathcal{R}\ T\ d\ d'$.

Case $C \not< A$: We have $\ell \in dom\, hOut$ and thus $\ell \in dom\, hOut'$ by hypothesis $\mathcal{R}\ Heap\ h\ h'$. Moreover,
$\mathcal{R}\ ((loctype\ \ell)\ state)\ (h\ell)\ (h'\ell)$ and so by $\mathcal{R}\ T\ d\ d'$ we get $\mathcal{R}\ ((loctype\ \ell)\ state)\ [h\ell \mid f \mapsto d]\ [h'\ell \mid f \mapsto d']$. So $\mathcal{R}\ Heap\ [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]\ [h' \mid \ell \mapsto [h'\ell \mid f \mapsto d']]$.

Assigning `new`:

$$\begin{array}{l}[\![\Gamma;\ C \vdash x := \texttt{new}\ B(\ ) : \texttt{com}]\!]\mu\eta h\\ =\ \textbf{let}\ \ell = fresh(B, h)\ \textbf{in}\\ \quad ([\eta \mid x \mapsto \ell], [h \mid \ell \mapsto [fields\, B \mapsto defaults]])\end{array}$$

Let $h_0 = [h \mid \ell \mapsto [fields\, B \mapsto defaults]]$, and $\eta_0 = [\eta \mid x \mapsto \ell]$. Because $x := \texttt{new}\ B(\ )$ is a constituent command in a method of non-rep class $C \neq A$, it is confined (by condition (i) for confinement of the class table). Thus we have $conf\ h_0$ and $conf\ C\ \eta_0\ h_0$. By definition, $conf\ C\ \eta_0\ h_0$ means $rng\, \eta_0 \cap locs(Rep\downarrow) = \varnothing$, so $\ell \notin locs(Rep\downarrow)$, and similarly $\ell' \notin locs(Rep'\downarrow)$. Then by Fact 3.3, we have $dom\, h \cap locs\, B = dom\, h' \cap locs\, B$, so by parametricity of $fresh$ we have $\ell = fresh(B, h) = fresh(B, h') = \ell'$. So, by Fact 3.4 and $\mathcal{R}\ \Gamma\ \eta\ \eta'$ we have $\mathcal{R}\ \Gamma\ \eta_0\ \eta_0'$. It remains to show $\mathcal{R}\ Heap\ h_0\ h_0'$, which we do by cases on B.

Case $B \not< A$: Writing $fields'$ for the fields given by CT', we have $fields\, B = fields'\, B$ and thus $\mathcal{R}\ B\ state\ [fields\, B \mapsto defaults]\ [fields'\, B \mapsto defaults]$. So, as B is non-rep and

$\neq A$, we can add ℓ to *hOut* and *hOut'* to get partitions that witness $\mathcal{R}$ *Heap* h_0 h_0'.

Case $B = A$**:** Let $hA_0 = [\ell \mapsto [\mathit{fields}\, A \mapsto \mathit{defaults}]]$ and $hA_0' = [\ell \mapsto [\mathit{fields}'\, A \mapsto \mathit{defaults}']]$. By basic simulation, we have $R\ h_1\ h_1'$. Thus to witness $\mathcal{R}$ *Heap* h_0 h_0' it suffices to let $hRep_0 = \varnothing$ and add $hA_0 * hRep_0$ (respectively $hA_0 * hRep_0$) to the given partition of h (respectively, h').

Case $B < A$**:** This is a combination of the preceding two cases.

Conditional:

$$\begin{aligned}&[\![\Gamma;\ C \vdash \texttt{if}\ e\ S_1\ \texttt{else}\ S_2 : \texttt{com}]\!]\mu\eta h\\ =\ &\textbf{let}\ b = [\![\Gamma;\ C \vdash e : \texttt{bool}]\!]\mu\eta h\ \textbf{in}\\ &\quad\textbf{if}\ b\ \textbf{then}\ [\![\Gamma;\ C \vdash S_1 : \texttt{com}]\!]\mu\eta h\\ &\quad\textbf{else}\ [\![\Gamma;\ C \vdash S_2 : \texttt{com}]\!]\mu\eta h\end{aligned}$$

By induction on e we have $\mathcal{R}\ \texttt{bool}_\perp\ b\ b'$. If $b = \perp$ then $b' = \perp$ and both semantics yield $\perp$. Otherwise, $b = b'$, and if $b = \mathsf{true}$ then the result follows directly by induction on S_1. And if $b = \mathsf{false}$ then the result follows by induction on S_2.

Sequence:

$$\begin{aligned}&[\![\Gamma;\ C \vdash S_1;\ S_2 : \texttt{com}]\!]\mu\eta h\\ =\ &\textbf{let}\ (\eta_0, h_0) = [\![\Gamma;\ C \vdash S_1 : \texttt{com}]\!]\mu\eta h\ \textbf{in}\\ &\quad[\![\Gamma;\ C \vdash S_2 : \texttt{com}]\!]\mu\eta_0 h_0\end{aligned}$$

Either both semantics of S_1 yield $\perp$ or neither does. In the latter case, by induction on S_1 we have
$\mathcal{R}\ (\Gamma \times \mathit{Heap})_\perp\ (\eta_0, h_0)\ (\eta_0', h_0')$. Moreover, as S_1 is a constituent of a method in CT and CT', by confinement of S_1 we have *conf* h_0, *conf* h_0', *conf* $C\ \eta_0\ h_0$, and *conf* $C\ \eta_0'\ h_0'$. So we can use induction on S_2 to obtain the result.

Local variable:

$$\begin{aligned}&[\![\Gamma;\ C \vdash \texttt{var}\ T\ x := e\ \texttt{in}\ S : \texttt{com}]\!]\mu\eta h\\ =\ &\textbf{let}\ d = [\![\Gamma;\ C \vdash e : U]\!]\mu\eta h\ \textbf{in}\\ &\quad\textbf{let}\ (\eta_0, h_0) = [\![(\Gamma, x : T);\ C \vdash S]\!]\mu[\eta \mid x \mapsto d]h\ \textbf{in}\\ &\quad(\eta_0 \lfloor x, h_0)\end{aligned}$$

By induction on e, $\mathcal{R}\ U_\perp\ d\ d'$. If $d = \perp$, then $d' = \perp$ and both semantics yield $\perp$. Otherwise, we have $\mathcal{R}\ T\ d\ d'$ by the corollary to Fact 3.4. Thus, from $\mathcal{R}\ \Gamma\ \eta\ \eta'$ we obtain $\mathcal{R}\ \Gamma, x : T\ [\eta \mid x \mapsto d]\ [\eta' \mid x \mapsto d']$. In order to use induction on S, we need to show *conf* $C\ [\eta \mid x \mapsto d]\ h$ and *conf* $C\ [\eta' \mid x \mapsto d']\ h'$. From condition (i) in the definition of confinement for CT, e is confined and hence $d \notin \mathit{locs}(\mathit{Rep}\downarrow)$. Because C is a non-rep class different from A, *conf* $C\ [\eta \mid x \mapsto d]\ h$ is equivalent to $\mathit{rng}\,[\eta \mid x \mapsto d] \cap \mathit{locs}(\mathit{Rep}\downarrow) = \varnothing$. This holds because $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}\downarrow) = \varnothing$ and $d \notin \mathit{locs}(\mathit{Rep}\downarrow)$. Similarly, we get *conf* $C\ [\eta' \mid x \mapsto d']\ h'$. Now, by induction on S we get that both semantics are $\perp$ or else
$\mathcal{R}\ (\Gamma, x : T \times \mathit{Heap})_\perp\ (\eta_0, h_0)\ (\eta_0', h_0')$. In the latter case, $\mathcal{R}\ (\Gamma \times \mathit{Heap})_\perp\ (\eta_0 \lfloor x, h_0)\ (\eta_0' \lfloor x, h_0')$ as required.

D. PROOF OF PROPOSITION 5.1

We also consider a third version that omits `g` entirely, as its effect is unobservable:

```
class A1 extends Object {
  unit callP(C y){ y.P(this); diverge }
  unit inc(){ skip }    }
```

Proof. We consider the first two implementations of `A1`. Choose *Rep*, *Rep'* to be arbitrary classes (unusable by clients but not needed by `A1`), as they are not relevant here. Let the basic simulation R relate h to h' just if they have the same singleton domain ℓ with $\mathit{loctype}(\ell) \leq \texttt{A1}$ and $h\,\ell\,g = h'\,\ell\,g' = 2 \times m$ for some $m \geq 0$. To deal with the third implementation of `A1`, the relation is just $h\,\ell\,g = 2 \times m$, independent from h', and the rest of the argument is the same. Let the method declarations for `A1.callP` be M, M', respectively, and the corresponding bodies be `S,S'` where

```
S= y.P(this);if(isEven(g))then diverge else skip
S'= y.P(this); diverge
```

To apply Theorem 4.1, we must show for confined μ, μ' that $\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu' \Rightarrow \mathcal{R}\ (A1, (y : C) \to \texttt{unit})\ ([\![M]\!]\mu)\ ([\![M']\!]'\mu')$ (and the same for the bodies of `inc`, which is easy). This boils down to proving that for confined and related μ, μ', h, h', and η, η', we have $\mathcal{R}$ *Heap* h_0 h_0' where h_0 (resp. h_0') is the heap returned by $[\![S]\!]\mu\eta h$ (resp. $[\![S']\!]'\mu'\eta' h'$). Now we use the semantics of S and S'; because μ, μ' are related, the meanings of P are related, so they yield related outcomes. (For example, in a state where y has exactly type C, we use that $\mu\,C\,P\,\eta_0\,h$ is related to $\mu'\,C\,P\,\eta_0'\,h'$ where η_0, η_0' are the input environments to the call $y.P(\mathit{this})$.) In short, the relation holds after the call $y.P$, hence the g fields are even and both S and S' diverge —so $(*)$ diverges for both semantics.

E. NOTES ON THE OBJECT ORIENTED MEYER-SIEBER EXAMPLE

Preceding Proposition 5.1, mentioned the need for behavioral subclassing. But there is another subtlety in the example which shows how our language differs from some idealized object oriented languages. For languages with instance-based visibility, encapsulation of private state can be formalized using existential types, and established parametricity results can be extended with some effort [32]. But Java visibility is class based. A method can access private fields of objects other than its target (i.e., `this`), if they are of the same class. In the example, class `C` must be different from `A1`, because the latter has no method named `P`. So in this example the encapsulation provided by privacy is enough to justify the argument for equivalence. One could modify the example to include a suitable method `P` in class `A1`. Then equivalence of the implementations would depend on that method also preserving the relation. Such a situation can still be analyzed in a modular way, because the implementation of `A1.P` must appear in `A1`. Consider, however, Java's `protected` fields, which are visible to method declarations in subclasses. If `g` is visible in a subclass, the equivalence becomes problematic, especially if it is to be dealt with in terms of an open system, i.e., we do not have a fixed collection of all the subclasses of `A1`. The issues raised by protected fields are beyond the scope of this paper. In practice, the issues are handled, if at all, by disciplined coding practices along with specifications and other annotations intended to ensure behavioral subclassing [37].