A Static Analysis for Instance-based Confinement in Java [Extended version, 30 Nov 01]

Anindya Banerjee⋆[1] and David A. Naumann⋆⋆[2]

[1] Department of Computing and Information Sciences, Kansas State University
Manhattan, KS 66506, USA
ab@cis.ksu.edu

[2] Department of Computer Science, Stevens Institute of Technology
Hoboken, NJ 07030 USA
naumann@cs.stevens-tech.edu

Abstract. A semantic definition is given for instance-based pointer confinement (alias control); this provides a form of encapsulation suited to many object-oriented designs. A syntax-directed static analysis is defined and proved, using a compositional semantics, to imply semantic confinement. Previous work by the authors, reviewed here, shows that this notion of confinement ensures a strong information-hiding property. The language studied here has features of sequential Java including mutable state, private fields, dynamic binding and inheritance, recursive classes, casts and type tests, and mutually recursive methods.

1 Introduction

One of the main benefits of object-oriented programming is information hiding and encapsulation: classes and visibility controls offer encapsulation without sacrificing extensibility and flexibility. But it is well known that the unfettered use of shared references to mutable objects is error-prone and can violate intended encapsulation. Quite a few proposals have been made for *confinement*, i.e., alias control (see, e.g., [CNP01,Boy01,VB01] and citations therein). Most proposals have been accompanied by an intuitive explanation of a property intended to be ensured, justified by examples and design patterns. Some proposals have included precise definitions of syntactic conditions necessary for the property; in some cases the conditions are impractically restrictive, which led to alternative proposals. No "obviously right" notion has emerged, partly because it is not easy to give precise arguments for a complex language and partly because there are several different, if related, properties of interest.

One useful confinement property is unique, i.e., unshared, references. These have strong properties (e.g., [Lea00,Boy01]); but for many purposes sharing is needed. Sharing can often be confined to the scope of a *module*, i.e., a sealed or closed package [Szy99], as shown in [VB01,GPV01]. This facilitates a coarse form

⋆ Supported by NSF grants EIA-9806835 and CCR-0093080.
⋆⋆ Supported by NSF grant INT-9813854.

of encapsulation that is quite useful, e.g., in debugging and security. Stronger and more fine-grained confinement is needed for reasoning about specifications using "modifies clauses" [MPH00,DLN98,LN00] and for justifying program transformations [BN02]. The present paper addresses confinement that is *instance-based* in that an object of public class is viewed as owning certain objects that constitute its internal representation [CPN98,CNP01,MPH00]. This form of confinement is applicable to many situations in practice, but our contribution is not just to treat this form; rather, we explore an approach, based on denotational semantics, that can be adapted to other forms.

The literature on confinement reflects the search for confinement notions that ensure strong properties but are not unduly restrictive or complicated. Example programs are an essential guide in this search, as are practical considerations to do with static checking. However, in order to find the most general and least restrictive conditions, a complementary approach is to proceed from a desired semantic property. In many cases there is a property that seems intuitively clear, e.g., "there is exactly one object with a reference to this object", or "objects of a module-scoped class are encapsulated in their defining module". But this is not the same as a precise semantic definition. Moreover, properties such as those just mentioned are not in themselves important. What matters are their consequences. For example, unique references help achieve non-interference between threads without the performance penalties of locking [Lea00]. Module- and instance-confined references provide numerous benefits of encapsulation, e.g., frame axioms needed for modular reasoning about mutable state [LN00,Mül01] and behavior-preserving replacement of components [Fow99,Szy99].

In previous work [BN02], we give a *semantic definition* of instance-based confinement closely related to the "ownership model" [CPN98,Mül01]. A typical application is for container classes: A public class A serves as an interface for an abstract collection of objects. It has private fields pointing to objects that constitute an internal data structure providing a concrete representation. The representation objects have some type *Rep* with module scope, and confinement means that references to *Rep* objects cannot be leaked to clients outside the module. Besides giving a precise notion of instance-based confinement, which has been done before (e.g., [CPN98,Mül01]), we formalize and prove a desired consequence: an *abstraction theorem* that says behavior of a client is independent of the internal representation of an object of class A, provided that the representation objects are confined. The theorem formalizes representation independence in terms of *simulation relations* which are widely used for reasoning about data abstraction [Rey84,LV95,dRE98], secure information flow [MS92,ABHR99], program analysis [NNH99], and program transformation [HHS93].

The present paper makes three contributions. First, we give a *static analysis* of instance-based confinement. That is, we define confinement as a property of program syntax. The definition is syntax-directed and does not require any code annotations or flow analyses. It can be checked in linear time, and is modular: the check for client classes is independent from the checks for A and *Rep* and their subclasses. The second contribution of the paper is a *proof of soundness* for the analysis: syntactic confinement implies semantic confinement.

Perhaps the most important contribution of our work is to show how a simple denotational semantics can be used in a straightforward way to study confinement and encapsulation. Prior to our work in [BN02], it was widely believed that small-step operational semantics was more suited to deal with the complicated interactions between features of rich languages like Java.

The semantics directly captures the operational semantics of dynamic binding and of the heap. But it abstracts from intermediate steps of computation and is compositional so proofs can be done by structural induction on syntax, without secondary inductions on computation steps. Compositional proofs facilitate extension to additional language features [HHS93] and may be beneficial for high-assurance applications involving mobile components [App01].

When we set out to prove the abstraction theorem [BN02], we believed that our initial definition of confinement was an accurate formalization of the intuitive notion suited to instance-based representation independence. But the construction of a detailed proof of the abstraction theorem was instrumental in uncovering significant errors in that definition. We also believed that the semantic formulations would be useful for other purposes, and this is supported by our experience with static analysis. The semantic definition of confinement only needed to be strengthened slightly, but without loss of generality, in order to serve as inductive hypothesis for the soundness proof. The analysis was defined hand in hand with construction of the proof.

The rest of the paper is organized as follows. Sections 2–5 summarize the relevant parts of our previous work [BN02]. Section 2 introduces the language and gives the formal syntax including typing rules. Section 3 gives the formal semantics. Section 4 discusses confinement and representation independence informally, mentioning several design patterns for which instance-based confinement is suitable. Section 5 gives the semantic definition of confinement. Section 6 gives the static analysis and Section 7 proves the soundness result. Section 8 assesses the work and remaining challenges.

On first reading, it may be preferable to skip the formal typing rules an Section 2, as well as the semantic definitions (Section 3). Readers familiar with [BN02] can skip Sections 2 and 3 and skim Sections 4 and 5.

2 Language Syntax and Typing

The 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 heap effects (`new` and field update) occur in commands rather than expressions. There are a couple of minor deviations from Java, e.g., the keyword `var` marks local variable declarations.

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. In subsequent examples we omit returns for the unit value; the singleton type, unit, corresponds to Java's "void". Object types are implicitly references: assignment creates aliases and == compares references.

A convenient simplification is to preclude side effects in expressions. No such restriction is imposed on the syntax, but the semantics discards effects of expressions, so our results are only interesting for programs that do not exploit expression effects.

To formalize the language, we adapt some notations from Featherweight Java [IPW99]. 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 the distinguished name $this$.

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

Other base types, such as integers, can be treated in the same way as bool, so we omit them for brevity.

Well formed class declarations are specified by rules [Car97] 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 typing environment Γ is a finite function from variable names to types. To simplify the typing rules and semantic definitions, we assume that variable names are not re-used and are distinct from parameter names.

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

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

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

$$\frac{\begin{array}{l} x \neq \mathit{this} \\ \Gamma;\ C \vdash e : T \quad T \leq \Gamma x \end{array}}{\Gamma;\ C \vdash x := e : \mathtt{com}} \qquad \frac{\begin{array}{l} \Gamma x = C \quad T\, f \in \mathit{dfields}\, C \\ \Gamma;\ C \vdash e : U \quad U \leq T \end{array}}{\Gamma;\ C \vdash x.f := e : \mathtt{com}} \qquad \frac{x \neq \mathit{this} \quad B \leq \Gamma x}{\Gamma;\ C \vdash x := \mathtt{new}\ B(\,) : \mathtt{com}}$$

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

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

Table 1. Typing rules for expressions and commands.

A complete program is given as a *class table* CT that associates each declared class name with its declaration. The typing rules make use of auxiliary notions that are defined in terms of CT, so the typing relation $\vdash$ depends on CT but this is elided in the notation. Because typing of each class is done in the context of the full table, methods can be mutually recursive, and so can field types. The rules for field access and update enforce privacy. Subsumption is built in to the rules using the subtyping relation $\leq$ on T specified as follows. For base types, $\mathtt{bool} \leq \mathtt{bool}$ and $\mathtt{unit} \leq \mathtt{unit}$. For classes C and D, we have $C \leq D$ iff either $C = D$ or the class declaration for C is $\mathtt{class}\ C\ \mathtt{extends}\ B\ \{\ \ldots\ \}$ for some $B \leq D$.

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

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

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.

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

$$
\begin{array}{lcl}
\eta \in [\![\Gamma]\!] & \Leftrightarrow & \mathit{dom}\,\eta = \mathit{dom}\,\Gamma \wedge \forall x \in \mathit{dom}\,\eta\ .\ \eta\, x \in [\![\Gamma\, x]\!] \\
s \in [\![C\ \mathit{state}]\!] & \Leftrightarrow & \mathit{dom}\, s = \mathit{fields}\, C \wedge \forall f \in \mathit{fields}\, C\ .\ sf \in [\![\mathit{type}(f, C)]\!] \\
h \in [\![\mathit{Heap}]\!] & \Leftrightarrow & \mathit{dom}\, h \subseteq \mathit{Loc} \wedge \forall \ell \in \mathit{dom}\, h\ .\ h\ell \in [\![(\mathit{loctype}\ \ell)\ \mathit{state}]\!] \\
[\![C, (\overline{x} : \overline{T}) \to T]\!] & = & [\![\overline{x} : \overline{T}, \mathit{this} : C]\!] \to [\![\mathit{Heap}]\!] \to ([\![T]\!] \times [\![\mathit{Heap}]\!])_{\perp} \\
[\![\mathit{MEnv}]\!] & = & (C : \mathit{ClassNames}) \nrightarrow (m : \mathit{MethodNames}) \nrightarrow [\![C, \mathit{mtype}(m, C)]\!]
\end{array}
$$

Table 2. Semantic domains.

A class table is well formed if each of its method declarations are well formed according to this rule:

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

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. A command denotes a function from initial state to either a final state or the error value $\perp$.

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. This could be done using an extra, immutable field, but for some definitions it is convenient to associate the class with the location. We assume given a function $\mathit{loctype} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{loctype}\ \ell = C$. We write $\mathit{locs}\ C$ for $\{\ell \mid \mathit{loctype}\,\ell = C\}$ and $\mathit{locs}(C{\downarrow})$ for $\{\ell \mid \mathit{loctype}\,\ell \leq C\}$. We assume given an allocator function *fresh* such that $\mathit{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$, for all C, h. 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 an allocator.

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

Table 2 gives the semantic domains. In addition to domains like $[\![T]\!]$ and $[\![\Gamma]\!]$ that correspond directly to syntactic notations, we use the following domains: $[\![Heap]\!]$ is the set of heaps, $[\![C\ state]\!]$ is the set of states of objects of class C, $[\![MEnv]\!]$ is the set of method environments, and $[\![(C, (\overline{x} : \overline{T}) \to T)]\!]$ is the set of meanings for methods of class C with result T and parameters $\overline{x} : \overline{T}$.

To cater for the fixpoint semantics of recursively defined methods, each domain is taken to be a partially ordered set with least upper bounds of ascending chains [DP90]. The sets $[\![Heap]\!]$, $[\![\texttt{bool}]\!]$, $[\![C]\!]$, and $[\![C\ state]\!]$ are ordered by equality. We write $\to$ for continuous function space, ordered pointwise, and $X_\perp$ for domain X with added bottom element $\perp$. We write $\rightharpoonup\!\!\!\!\!\shortmid\,$ for finite partial functions.

In the formalization of confinement, we use the relation $\nleqq$ on types defined by $T \nleqq U \Leftrightarrow T \nleq U \wedge T \ngeq U$. We often exploit the properties $T \leq U \Rightarrow [\![T]\!] \subseteq [\![U]\!]$ and $T \nleqq U \Rightarrow [\![T]\!] \cap [\![U]\!] = \varnothing$.

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

$$
\begin{array}{lcl}
[\![\Gamma;\ C \vdash e : T]\!] & \in & [\![MEnv]\!] \to [\![\Gamma]\!] \to [\![Heap]\!] \to [\![T]\!]_\perp \\
[\![\Gamma;\ C \vdash S : \texttt{com}]\!] & \in & [\![MEnv]\!] \to [\![\Gamma]\!] \to [\![Heap]\!] \to ([\![\Gamma]\!] \times [\![Heap]\!])_\perp
\end{array}
$$

The 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., $[\eta \mid x \mapsto d]$. In the semantics of local variables, we write $\downharpoonleft$ for domain restriction: if d_1 is in the domain of function d_2 then $d_2 \downharpoonleft d_1$ is the function like d_2 but without d_1 in its domain.

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

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

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

Table 3. Semantics of expressions.

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

$$
\begin{array}{l}
\mu_0\, C\, m = \lambda\eta.\ \lambda h.\ \bot \\
\mu_{j+1}\, C\, m = [\![M]\!]\mu_j \quad \text{if } m \text{ is declared as } M \text{ in } C \\
\mu_{j+1}\, C\, m = \mu_{j+1}\, B\, m \quad \text{if } m \text{ is inherited from } B \text{ in } C \\
[\![M]\!]\mu\eta h = \ \textbf{let } (\eta_0, h_0) = [\![(\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash S : \texttt{com}]\!]\mu\eta h \textbf{ in} \\
\qquad\qquad \textbf{let } d = [\![(\overline{x} : \overline{T}, \mathit{this} : C);\ C \vdash e : T]\!]\mu\eta_0 h_0 \textbf{ in } (d, h_0) \\
\qquad\quad \text{where, in class } C, \text{ we have } M = T\ m(\overline{T}\ \overline{x})\{S;\ \texttt{return}\, e\}.
\end{array}
$$

4 Confinement and Representation Independence

This expository section describes the range of application of our notion of confinement, and the encapsulation property that it ensures.

One way to exploit confinement is in *modular reasoning about a program*, via frame axioms for delimiting the part of state that can be modified by a command [LN00,Mül01]. Another way to exploit confinement, described below, is in *reasoning about the modularity of a program*: determining what constitutes an encapsulated representation that can be replaced by another. Put differently, we show how confinement can be used to achieve a sound but flexible notion of equivalence (or refinement [CN01]) of components.

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

Table 4. Semantics of commands.

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

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

Here is an alternative implementation of A0 that uses an isomorphic representation to achieve the same behavior.

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

The following command could be a constituent of a method of a client class C.

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

The behavior of this command is independent of the encapsulated representation of A0. Informally, this is because the client cannot access the private fields g and g' and the locations stored in those fields are not leaked to the client.

More formally, we argue using a simulation relation that the behavior is the same regardless of which implementation of `A0` is chosen. First, we give a relation between states of an object `o` for the first implementation and `o'` for the second. We say `o` and `o'` are related just if either $\texttt{o.g} = \texttt{null} = \texttt{o'.g'}$ or $\texttt{o.g} \neq \texttt{null} \neq \texttt{o'.g'}$ and $\texttt{o.g.f} = \neg(\texttt{o'.g'.f})$. If `o,o'` are newly constructed,[1] $\texttt{o.g} = \texttt{null} = \texttt{o'.g'}$ holds. Invocations of `setg` and `getg` can and must be shown to maintain the relation. Then it follows, by the abstraction theorem [BN02], that the relation is maintained by all client programs. Moreover, by the identity extension lemma [BN02], related states are identical (after garbage collection) if `A0` objects are not reachable. This implies that a client using `o` in a local variable[2] cannot be distinguished from one using `o'`.

The abstraction theorem requires that the class table be confined. For example, suppose we add to `A0` a method

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

which leaks a reference to the private field `g` (or `g'` in the second version of `A0`). A client class `C` can exploit the leak 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 second implementation of `A0` is chosen, but does not diverge with the first implementation. Due to subtyping, such problems can occur even if class `Boolean` has module scope and the client is outside the module. Another way for `g` to be leaked is for it to be passed as an argument in a callback to `C`, e.g., if we add to `A0` a method

```
unit badCallback(C x){ x.m(g) }
```

where $\mathit{mtype}(m, C) = (y : \texttt{Boolean}) \to \texttt{unit}$.

The simulation argument given above involves a relation between a pair of instances of class `A0`. In general there can be many instances, each with its own representation objects separate from the representations of other instances. The corresponding notion of confinement is depicted in Figure 1, which shows two instances of an interface class A.

Our formalization of confinement is based on classes in the sense that we use class names to distinguish between interface objects (class A), representations (class Rep, with $\mathit{Rep} \not\leq A$) and others. We allow subclasses, e.g., representation objects can have some subclass of Rep. This use of class names is partly an

[1] In practice the 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.

[2] If `A0` is a field of the client object, that object should itself be considered the interface and `A0` part of the representation.

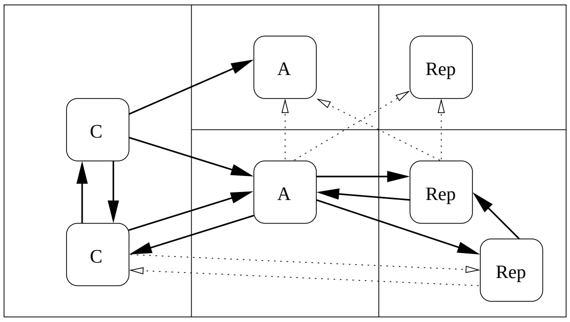

Fig. 1. Confinement example. Dotted references disallowed.

artifice of formalization. In practice, the interface object might have the same class as its representations (e.g., a simple linked list with header node), and representations may use library classes also used by clients. But class names are very convenient for formalization, and our formulation loses no generality because freshly named copies of classes can be used (as in [Mül01] and, in effect, [CPN98]). An alternative formalization could use Java interfaces to give a common super-type for all representation objects.

Module-based confinement is intended to achieve encapsulation at a larger granularity [VB01] that is quite useful for informal practice. The information-hiding property provided by module-based confinement is not fundamentally different from instance-based confinement, but it appears to be more complicated to formalize and we know of no such work for object-oriented languages.

Instance-based confinement is appropriate for many applications. The private fields of any public class are candidates to be considered representations. Collection classes provide examples, but many others can be found, e.g., in the literature on design patterns [GHJV95]. For example, in the Subscribe-publish (i.e., Observer) pattern, the set of currently subscribed clients is not accessible to a client. The Factory, Builder, and Memento patterns also involve a single interface object that encapsulates representation objects. In object-oriented telecommunications middleware [Coc00], proxy objects are used to isolate clients from services, e.g., to facilitate revocation by the provider of a shared service: The provider cannot prevent the client from having references to the proxy, but it can remove the proxy's reference to the actual service object.

The leading example of [VB01], a security flaw in JDK 1.1, fits our formulation of instance-based confinement. A public method of class `Class` was intended to give read access to the signers of a class; but by returning a reference to the mutable object referenced by private field `signers` it allowed a client class to add a trusted principal to its signers.

Collection classes have two features that do not fit our formal definition of confinement. First, generic collections of client objects are usually implemented with representation objects, like tree nodes, that point back to the collected client objects. Second, iterator objects are like interface objects, and they point to representations owned by another interface object. It is possible, though non-trivial, to admit such patterns [CPN98,Mül01,CNP01].

5 Semantic Definition of Confinement

This section gives the semantic definition of confinement[3] and some basic properties thereof. We use notation for heaps adapted from the pointer logic of [IO01,Rey01].

Confinement requires that in every state the heap can be partitioned as shown in Figure 1, for which we write $hOut * hA_1 * hRep_1 * hA_2 * hRep_2$. Each object of class A is in a heap hA_i and the corresponding $hRep_i$ contains its representation objects; $hOut$ 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.

Definition 1. *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* $size(dom\ hA_i) = 1$
- $dom\ hRep_i \subseteq locs(Rep{\downarrow})$
- $dom\ hOut \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) = \varnothing$

A heap may have several admissible partitions, e.g., if there are inaccessible *Rep* objects. The definitions and results do not depend on choice of partition.

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

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

To say that no objects in h_1 *except for the ones pointed to by fields $\overline{f}$* contain references to objects in h_2, we define

$$h_1 \not\rightsquigarrow^{\overline{f}} h_2 \Leftrightarrow \forall \ell \in dom\ h_1 \,.\, rng((h_1\ \ell)\lfloor\overline{f}) \cap dom\ h_2 = \varnothing$$

Definition 2 (Confined heap). *Let $\overline{g}$ be the list of private fields of class A. Heap h is* confined, *written $conf\ h$, iff h has an admissible partition such that for all i, j with $i \neq j$ we have*
- $hOut \not\rightsquigarrow hRep_i$
- $hRep_i \not\rightsquigarrow hOut$
- $hA_i * hRep_i \not\rightsquigarrow hA_j * hRep_j$
- $hA_i \not\rightsquigarrow^{\overline{g}} hRep_i$

[3] To obtain an inductive hypothesis for the soundness proof, we strengthen the definition from that needed in [BN02], as follows. Definition 2 restricts references from A to *Rep* objects to be via the private fields of A. Definition 3 adds constraint (c), and confinement for environments and expressions of *Rep* methods (which are ensured by (c)). For *conf* μ, we add that the initial environment is confined in the final heap.

The definition imposes the restrictions of Figure 1 and, because we are concerned with private fields of class A, it requires that references from an A object to its representation be via those fields only.

For a program to be confined, it must preserve confinement of heaps. To formalize this notion, we must say what it is for environments to be confined. This leads to corresponding conditions for method environments, expressions, and commands: essentially, they preserve confinement of the heap and environment. In the case of method call this is difficult to formalize in purely semantic terms so we appeal to a restriction on method signatures.

Definition 3 (Confined signature). *Suppose $mtype(m, C) = (\overline{x} : \overline{T}) \to T$ where m is a method declared or inherited in class C. We say the* signature of m is confined in C *provided*

$$\begin{array}{lrcl} \text{(a)} & C \leq A & \Rightarrow & \overline{T} \not\lesseqgtr Rep \wedge \overline{T} \not\lesseqgtr A \wedge T \not\lesseqgtr Rep \\ \text{(b)} & C \not\leq A \wedge C \not\leq Rep & \Rightarrow & \overline{T} \not\lesseqgtr Rep \\ \text{(c)} & C \leq Rep & \Rightarrow & (U \leq A \vee U \leq Rep) \text{ for all } U \in \overline{T}. \end{array}$$

The restrictions may appear rather strong, but similar constraints are imposed, in various ways, in the literature. We have not found non-trivial examples that are confined but do not meet these constraints.

Recall that we consider only public methods. Restriction (a) precludes examples like methods `bad` and `badCallback` in Section 4. Moreover, for a client to pass an A-reference to an A-object o would lead to a violation of heap confinement unless the reference is to o itself or o makes no use of it. In (b), restriction $\overline{T} \not\leq Rep$ is clearly necessary, as passing a *Rep*-reference to a client violates confinement. The reverse, $\overline{T} \not\geq Rep$, is more questionable; e.g., it disallows parameters of type `Object` for client methods. The restriction can probably be avoided by using more complicated semantic conditions and, for the static analysis, some form of flow analysis. But the syntactic restriction is in keeping with the use of types in the formulation of confinement. Many uses of `Object` are considered undesirable, as they relenquish the benefits of typing, but they are unavoidable in the absence of parametric polymorphism. Future versions of Java will include parametric polymorphism and preliminary work indicates that our results extend easily to polymorphism as it is found in GJ [BOSW98].

We say a class C is *non-rep* provided $C \not\lesseqgtr Rep$.

Definition 4 (Confined class table). *Class table CT is confined iff (a) the signature of every method in every class is confined, and (b) for every non-rep C and every $T\ m(\overline{T}\ \overline{x})\{S; \mathtt{return}\, e\}$ in $CT(C)$, it is the case that S, e, and all constituents thereof are confined (see Table 5).*

Proposition 1. *If CT is confined, and no methods are inherited in Rep from any $B > Rep$, then its semantics, $\hat{\mu}$, is confined; that is, conf $\hat{\mu}$.*

The proof is omitted. It is similar to one in [BN02] but also using Definition 3(c).

The condition on inheritance deals with a problem that can be handled using "anonymous methods" as in [VB01], but that appears to require code annotations which we prefer to avoid in this paper.

Environment η is *confined for C and h*, written $conf\ C\ \eta\ h$, iff the following holds.

$$
\begin{array}{l}
C \not\leq Rep \wedge C \not\leq A \ \Rightarrow\ rng\ \eta \cap locs(Rep{\downarrow}) = \varnothing \\
C < A \ \Rightarrow\ rng\ \eta \cap locs(Rep{\downarrow}) = \varnothing \wedge rng\ \eta \cap locs(A{\downarrow}) = \{\eta\ this\} \\
C = A \ \Rightarrow\ rng\ \eta \cap (locs(Rep{\downarrow}) \cup locs(A{\downarrow})) \subseteq dom(hRep_j * hA_j) \\
\qquad\qquad \text{for some partition and some } j \text{ with } dom(hA_j) = \{\eta\, this\} \\
C \leq Rep \ \Rightarrow\ rng\ \eta \cap Loc \subseteq dom(hRep_j * hA_j) \\
\qquad\qquad \text{for some partition and some } j \text{ with } \eta\, this \in dom(hRep_j)
\end{array}
$$

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

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

and (ii) for all $C \leq Rep$, and all m, h, η, if $conf\ h$ and $conf\ C\ \eta\ h$ then

$$
\begin{array}{rl}
\mu C m \eta h \neq \bot \ \Rightarrow & \textbf{let}\ (d, h_0) = \mu C m \eta h\ \textbf{in} \\
& conf\ h_0 \wedge conf\ C\ \eta\ h_0 \wedge (d \in Loc \ \Rightarrow\ d \in dom(hRep_j * hA_j)) \\
& \ \text{for some partition and } j \text{ with } \eta\, this \in dom(hRep_j).
\end{array}
$$

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

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

For expressions, for non-rep $C \not\leq A$ we say $\Gamma;\ C \vdash e : T$ is confined iff for any confined h, μ, and η confined in C, h:

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

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

$$
\begin{array}{rl}
[\![\Gamma;\ C \vdash e : T]\!]\mu\eta h \neq \bot \ \Rightarrow & \textbf{let}\ d = [\![\Gamma;\ C \vdash e : T]\!]\mu\eta h\ \textbf{in} \\
& d \notin locs(Rep{\downarrow}) \wedge (d \in locs(A{\downarrow}) \ \Rightarrow\ d = \eta\ this)
\end{array}
$$

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

$$
\begin{array}{rl}
[\![\Gamma;\ A \vdash e : T]\!]\mu\eta h \neq \bot \ \Rightarrow & \textbf{let}\ d = [\![\Gamma;\ A \vdash e : T]\!]\mu\eta h\ \textbf{in} \\
& d \in (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \ \Rightarrow\ d \in dom(hRep_j * hA_j) \\
& \ \text{for some partition and } j \text{ with } dom(hA_j) = \{\eta\, this\}.
\end{array}
$$

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

$$
\begin{array}{rl}
[\![\Gamma;\ C \vdash e : T]\!]\mu\eta h \neq \bot \ \Rightarrow & \textbf{let}\ d = [\![\Gamma;\ C \vdash e : T]\!]\mu\eta h\ \textbf{in} \\
& d \in Loc \ \Rightarrow\ d \in dom(hRep_j * hA_j) \\
& \ \text{for some partition and } j \text{ with } \eta\, this \in dom(hRep_j).
\end{array}
$$

Table 5. Confinement of environments, commands, and expressions.

6 Static Analysis

The static analysis is a syntactic specification of confinement for designated classes A, Rep. It is defined in a style similar to that of typing rules. The analysis is applied only to a well-formed class table, i.e., one whose class declarations are well-formed according to the rules in Section 2. A class table is syntactically confined iff all method declarations in all classes are syntactically confined, i.e., the bodies of the methods (comprising commands and expressions) are syntactically confined.

We write $\Gamma;\ C \rhd e : T$ to signify syntactic confinement for an expression e typable as $\Gamma;\ C \vdash e : T$. The analysis is defined in a compositional manner in Table 6. For example, the analysis declares $e.f$ confined provided e, a constituent of $e.f$, is. Likewise $e.m(\overline{e})$ is syntactically confined provided both constituents e and $\overline{e}$ are. A consequence of compositionality is that it is straightforward to implement a modular check for syntactic confinement. We write $\Gamma;\ C \rhd S$ to signify syntactic confinement for a command S typable as $\Gamma;\ C \vdash S : \texttt{com}$.

The crux of the analysis is the constraints on types for field update, `new`, and method call expressions. To confine $x.f := e$ in a class C we need to guarantee that if $C \leq A$ then the field f of the object in the heap is not updated to contain a reference to a different A-object as this will break confinement. So we force the type of e to be incomparable to A. To confine $x := \texttt{new}\ B()$ in a class C we need to guarantee that if C is a proper subclass of A, or C is an outsider (i.e, $C \not\leq A$ and $C \not\leq Rep$) then it has no direct access to Rep-objects; hence the requirement $B \not\leq Rep$. Moreover, if $C \leq A$, then it cannot create a reference to a different A object as this will break confinement; hence the requirement $B \not\leq A$. For method call expressions, if the caller is an A object, it is useless for the result to be any A object but itself. To impose this on the signature of all public methods would be too restrictive, so the analysis requires $T \not\lesseqgtr A$ for the result type of methods called from $C \leq A$.

Definition 5 (Syntactically confined class table). *Class table CT is syntactically confined, written $\rhd CT$, iff for all classes C and all method declarations in $CT(C)$ we have $C \rhd T\ m(\overline{T}\ \overline{x})\{S;\ \texttt{return}\ e\}$ as defined in Table 6.*

7 Soundness

The main theorem says that the static analysis is sound: that is, if the analysis declares a class table to be syntactically confined and the signature of every method is confined then the class table is semantically confined via the definitions in Section 5.

Theorem 1 (Soundness of static analysis for confinement). *If $\rhd CT$ and every method signature in CT is confined then CT is confined.*

$$\Gamma;\ C \rhd x : \Gamma x \qquad \Gamma;\ C \rhd \mathtt{null} : B \qquad \frac{\mathit{mtype}(m, D) = (\overline{x} : \overline{T}) \to T \quad C \leq A \Rightarrow T \not\gtrless A \quad \Gamma;\ C \rhd e : D \quad \Gamma;\ C \rhd \overline{e} : \overline{U}}{\Gamma;\ C \rhd e.m(\overline{e}) : T}$$

$$\Gamma;\ C \rhd e_1 {=}{=} e_2 : \mathtt{bool} \qquad \frac{\Gamma;\ C \rhd e : C}{\Gamma;\ C \rhd e.f : T}$$

$$\frac{\Gamma;\ C \rhd e : D}{\Gamma;\ C \rhd (B)\ e : B} \qquad \Gamma;\ C \rhd e\ \mathtt{instanceof}\ B : \mathtt{bool}$$

$$\frac{x \neq \mathit{this} \quad \Gamma;\ C \rhd e : T}{\Gamma;\ C \rhd x := e} \qquad \frac{C \leq A \Rightarrow U \not\gtrless A \quad \Gamma;\ C \rhd e : U}{\Gamma;\ C \rhd x.f := e} \qquad \frac{C \neq A \Rightarrow B \not\leq \mathit{Rep} \quad C \leq A \Rightarrow B \not\leq A}{\Gamma;\ C \rhd x := \mathtt{new}\ B(\,)}$$

$$\frac{\Gamma;\ C \rhd e : D \quad \Gamma;\ C \rhd \overline{e} : \overline{U}}{\Gamma;\ C \rhd e.m(\overline{e})} \qquad \frac{\Gamma;\ C \rhd S_1 \quad \Gamma;\ C \rhd S_2}{\Gamma;\ C \rhd \mathtt{if}\ e\ S_1\ \mathtt{else}\ S_2}$$

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

$$\frac{(\overline{x} : \overline{T}, \mathit{this} : C);\ C \rhd S \qquad (\overline{x} : \overline{T}, \mathit{this} : C);\ C \rhd e : U}{C \rhd T\ m(\overline{T}\ \overline{x})\{S;\ \mathtt{return}\ e\}}$$

Table 6. Static analysis for confined expressions, commands, and method declarations.

Proof. Condition (a) for confinement of CT requires that every method signature is confined, which we have by hypothesis. It remains to show (b), for which it suffices to show that for all non-rep C and all A, method bodies and constituents of method bodies occurring in these classes are confined.

Assume $\rhd CT$. Then for all classes C and all method declarations $T\ m(\overline{T}\ \overline{x})\{S;\ \mathtt{return}\ e\}$ in $CT(C)$ we have $C \rhd T\ m(\overline{T}\ \overline{x})\{S;\ \mathtt{return}\ e\}$. By the static analysis we have $(\overline{x} : \overline{T}, \mathit{this} : C);\ C \rhd S$ and $(\overline{x} : \overline{T}, \mathit{this} : C);\ C \rhd e : U$. By Lemma 1 and Lemma 2 below we get that e and S, respectively, are confined. By induction on the definition of $\rhd$, syntactic confinement of an expression or command implies syntactic confinement of its constituents. That all constituents of e and S are confined now follows from the definition of $\rhd$ by inductively applying Lemma 1 and Lemma 2.

Lemma 1. *If every method signature in CT is confined and $\Gamma;\ C \rhd e : T$ then $\Gamma;\ C \vdash e : T$ is confined, for all Γ, e, T and all non-rep C.*

Proof. Go by induction on $\Gamma;\ C \rhd e : T$ and by cases on C. Assume *conf* h, *conf* μ, *conf* $C\ \eta\ h$ and every method signature in CT is confined. Assume that an admissible partition of h is given. Also let $d = [\![\Gamma;\ C \vdash e : T]\!]$ and assume that $d \neq \bot$ (the case $d = \bot$ is immediate).

For $C \not\leq A$, we must show that $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. For $C < A$, we must show that $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and that $d \in \mathit{locs}(A{\downarrow}) \Rightarrow d = \eta\ \mathit{this}$. For $C = A$, we must show $d \in (\mathit{locs}(A{\downarrow}) \cup \mathit{locs}(\mathit{Rep}{\downarrow})) \Rightarrow d \in \mathit{dom}(hA_j * h\mathit{Rep}_j)$ for some j with $\mathit{dom}(hA_j) = \{\eta\ \mathit{this}\}$.

We consider a few cases below; the remaining cases are in Appendix A.

Cast expression: Recall the semantic definition.

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

As we assume $d \neq \bot$, we have $d = \ell$. If $C \not\leq A$, since $\Gamma;\ C \rhd (B)\, e : B$ we have $\Gamma;\ C \rhd e : D$, therefore, by induction on e we have $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. If $C < A$, then by induction on e we have $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $(\ell \in \mathit{locs}(A{\downarrow}) \Rightarrow \ell = \eta\, \mathit{this})$. If $C = A$, then by induction on e we have $(\ell \in (\mathit{locs}(A{\downarrow}) \cup \mathit{locs}(\mathit{Rep}{\downarrow})) \Rightarrow \ell \in \mathit{dom}(hA_j * h\mathit{Rep}_j)$ for some j with $\mathit{dom}(hA_j) = \{\eta\, \mathit{this}\}$. In each case, these are the conditions required for confinement of expression $(B)\, e$ in class C.

Method call as an expression:

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

We assume $\ell \in \mathit{dom}\, h$, as otherwise $d = \bot$, and $d = d_0 \neq \bot$.

First we consider the case of $C < A$ or $C \not\leq A$. Since $\Gamma;\ C \rhd e : D$, by induction on e we have $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. Let $\eta' = [\overline{x} \mapsto \overline{d}, \mathit{this} \mapsto \ell]$. We claim that $\mathit{conf}\, (\mathit{loctype}\, \ell)\, \eta'\, h$; then, we have by definition $\mathit{conf}\, \mu$ that $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. It remains to prove the claim, for which we go by cases on $\mathit{loctype}\ \ell$ (recall that $\mathit{loctype}\ \ell \not\leq \mathit{Rep}$).

(i) $\mathit{loctype}\ \ell < A$: By confined method signature, Definition 3(a), we have $\overline{T} \not\geq \mathit{Rep}$ and $\overline{T} \not\geq A$, so $\overline{d} \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $\overline{d} \notin \mathit{locs}(A{\downarrow})$. Now $\mathit{conf}\, (\mathit{loctype}\ \ell)\eta'\, h \Leftrightarrow (\mathit{rng}\ \eta' \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing) \wedge (\mathit{rng}\ \eta' \cap \mathit{locs}(A{\downarrow}) = \{\eta'\ \mathit{this}\})$. And, $\mathit{rng}\ \eta' \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \{\overline{d}, \ell\} \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$, since $\overline{d} \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. And, $(\mathit{rng}\ \eta' \cap \mathit{locs}(A{\downarrow})) = \{\overline{d}, \ell\} \cap \mathit{locs}(A{\downarrow}) = \{\ell\} = \{\eta'\ \mathit{this}\}$.

(ii) $\mathit{loctype}\ \ell \not\leq A$: By confined method signatures, Definition 3(b), we have $\overline{T} \not\geq \mathit{Rep}$, so $\overline{d} \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. Now $\mathit{conf}\, (\mathit{loctype}\ \ell)\, \eta'\, h \Leftrightarrow (\mathit{rng}\ \eta' \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing)$. And, $\mathit{rng}\ \eta' \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \{\overline{d}, \ell\} \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$, since $\overline{d} \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$.

(iii) $\mathit{loctype}\ \ell = A$: Then by confined method signatures, $\overline{T} \not\geq A$ and $\overline{T} \not\geq \mathit{Rep}$, so $\overline{d} \notin \mathit{locs}(A{\downarrow})$ and $\overline{d} \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. Now $\mathit{conf}\ (\mathit{loctype}\ \ell)\ \eta'\ h \Leftrightarrow \mathit{rng}\ \eta' \cap (\mathit{locs}(\mathit{Rep}{\downarrow}) \cup \mathit{locs}(A{\downarrow})) \subseteq \mathit{dom}(hA_j * h\mathit{Rep}_j)$ for some j where $\mathit{dom}(hA_j) = \{\eta'\ \mathit{this}\} = \{\ell\}$. And $\mathit{rng}\ \eta' \cap (\mathit{locs}(\mathit{Rep}{\downarrow}) \cup \mathit{locs}(A{\downarrow})) = \{\overline{d}, \ell\} \cap (\mathit{locs}(\mathit{Rep}{\downarrow}) \cup \mathit{locs}(A{\downarrow})) = \{\ell\} \subseteq \mathit{dom}(hA_j * h\mathit{Rep}_j)$, for some j and $\mathit{dom}(hA_j) = \{\eta'\ \mathit{this}\} = \{\ell\}$.

Having proved $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ in each case, we are done with the case $C \not\leq A$. For $C < A$ we also need to show $d \in \mathit{locs}(A{\downarrow}) \Rightarrow d = \eta\, \mathit{this}$. But the analysis for method call as expression stipulates $C \leq A \Rightarrow T \not\geq A$, so $d \notin \mathit{locs}(A{\downarrow})$ and we are done for $C \neq A$.

For $C = A$, we must show $d \in (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \Rightarrow d \in dom(hA_j * hRep_j)$ for some j with $dom(hA_j) = \{\eta\, this\}$. We go by cases on $loctype\,\ell$. For $loctype\,\ell \not\leq Rep$, we have $d \notin locs(A{\downarrow})$ by the analysis ($C \leq A \Rightarrow T \not\geq A$), and $d \notin locs(Rep{\downarrow})$ by the same argument as above.

It remains to consider $C = A$ and $loctype\,\ell \leq Rep$. We claim $conf\,(loctype\,\ell)\eta' h$. Then by $conf\ \mu$ and $loctype\,\ell \leq Rep$ we have $d \in dom(hRep_j * hA_j)$ for the j with $\eta'\, this \in dom(hRep_j)$. Now by confinement of e at $C = A$, this must be the same j for which $\{\eta\, this\} = dom(hA_j)$, so $d \in dom(hA_j * hRep_j)$ as required. Now we prove the claim $conf\ (loctype\,\ell)\ \eta'\ h$, that is, we show $rng\,\eta' \cap Loc \subseteq dom(hA_j * hRep_j)$ where j is as above. By confinement of the signature of m, Definition 3(b), $rng\,\eta' \cap Loc = rng\,\eta' \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})$. Now $rng\,\eta' = \overline{d}, \ell$ and $\ell \in dom(hRep_j)$. By confinement of $\overline{e}$ and induction, we have $\overline{d} \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA_j * hRep_j)$.

Lemma 2. *If every method signature in CT is confined and $\Gamma;\ C \rhd S$ then $\Gamma;\ C \vdash S : \mathtt{com}$ is confined, for all Γ, S and non-rep C.*

Proof. Go by induction on $\Gamma;\ C \rhd S$ and by cases on C. Assume $conf\ h$, $conf\ \mu$, $conf\ C\ \eta\ h$ and every method signature in CT is confined. Also assume $[\![\Gamma;\ C \vdash S : \mathtt{com}]\!] \neq \bot$ and let $(\eta_0, h_0) = [\![\Gamma;\ C \vdash S : \mathtt{com}]\!]$. We must show $conf\ h_0$ and $conf\ C\,\eta_0\,h_0$. We consider a few cases below; remaining cases are in Appendix B.

Field Update:

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

As we consider the non-$\bot$ case, we have $\eta_0 = \eta$ and $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$.

If $C \not\leq A$, then since $conf\ C\ \eta\ h$ we have $rng\ \eta \cap locs(Rep{\downarrow}) = \varnothing$. Hence $\ell \notin locs(Rep{\downarrow})$. Also, $\ell \notin locs(A{\downarrow})$ since $\Gamma x = C$. By the analysis, $\Gamma;\ C \rhd e : U$, hence by Lemma 1, $d \notin locs(Rep{\downarrow})$.

We proceed to show $conf\ h_0$ for the case $C \not\leq A$. Since h is confined, there exists an admissible partition of h such that $(hOut * hA_1 * hRep_1 * \ldots) = h$. Clearly, $\ell \in dom(hOut)$; and, by confinement of h, $hOut \not\mapsto hRep_i$. So we can partition h_0 exactly as we partitioned h: $hOut \not\mapsto hRep_i$ still holds because $d \notin locs(Rep{\downarrow})$. Finally, note that $conf\ C\ \eta_0\ h_0$ follows from $conf\ C\ \eta\ h$..

If $C < A$, then since $conf\ C\ \eta\ h$ we have, $rng\ \eta \cap locs(Rep{\downarrow}) = \varnothing$ and $rng\ \eta \cap locs(A{\downarrow}) = \{\eta\ this\}$. Hence $\ell \notin locs(Rep{\downarrow})$ and if $\ell \in locs(A{\downarrow})$ then $\ell = \eta\ this$. By the analysis, $\Gamma;\ C \rhd e : U$, hence by Lemma 1, $d \notin locs(Rep{\downarrow})$ and $(d \in locs(A{\downarrow}) \Rightarrow d = \eta\ this)$. By analysis, however, $U \not\geq A$, hence $d \notin locs(A{\downarrow})$.

We proceed to show $conf\ h_0$ in the case $C < A$. Since h is confined, there exists an admissible partition of h such that $(hOut * hA_1 * hRep_1 * \ldots) = h$. Clearly, $\ell \in dom(hOut)$ or $\ell \in dom(hA_i)$ for some i. In the former case, we proceed as in the case for $C \not\leq A$ and, using $rng\ \eta \cap locs(Rep{\downarrow}) = \varnothing$, show $conf\ h_0$ and $conf\ C\eta_0\,h_0$. In the latter case, since $rng\ \eta \cap locs(A{\downarrow}) = \{\eta\ this\}$, we have $\ell = \eta\ this$. We argue that h_0 can be partitioned the same as h. Since h is confined, we have, $i \neq j \Rightarrow (hA_i * hRep_i) \not\hookrightarrow (hA_j * hRep_j)$. This implication still holds for h_0 because $d \notin locs(Rep{\downarrow})$ and $d \notin locs(A{\downarrow})$. Moreover, the condition $hA_i \not\hookrightarrow^{\overline{g}} hRep_i$ is still maintained by h_0, since by visibility, f is not in the private fields $\overline{g}$ of A. Finally, note that $conf\ C\ \eta_0\ h_0$ follows from $conf\ C\ \eta\ h$.

If $C = A$, then since $conf\ A\ \eta\ h$ we have, $rng\ \eta \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA_j * hRep_j)$ for some j where $dom(hA_j) = \{\eta\ this\}$. Since $\Gamma\ x = A$, therefore, $\ell \in locs(A{\downarrow})$; hence $\ell \notin locs(Rep{\downarrow})$ and $\ell \in dom(hA_j)$ for some j with $\ell = \eta\ this$. Since $\Gamma;\ A \rhd e : U$, by induction on e we have, $d \in (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \Rightarrow d \in dom(hA_k * hRep_k)$ for some k where $dom(hA_k) = \{\eta\ this\}$.

By the analysis, $U \not\lesseqgtr A$. Hence $d \notin locs(A{\downarrow})$. Suppose $d \in locs(Rep{\downarrow})$. Then $d \in dom(hRep_k)$ for some k; choose the corresponding partition hA_k so that $d \in dom(hA_k * hRep_k)$ with $dom(hA_k) = \{\eta\ this\}$. Hence $k = j$. So to show $conf\ h_0$, it suffices to partition the heap exactly as h; because $C = A$, by visibility, $f \in \overline{g}$, so $hA_j \not\hookrightarrow^{f} hRep_j$ follows for any $f \in \overline{g}$. Now $conf\ A\ \eta\ h_0$ by definition of confinement for η.

Now suppose $d \notin locs(Rep{\downarrow})$. Then to show $conf\ h_0$, it suffices to partition the heap exactly as h, noting that $d \in dom(hOut)$. And, $conf\ A\ \eta\ h_0$ follows by definition of confinement for η.

Assigning `new`:

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

Then $\eta_0 = [\eta \mid x \mapsto \ell]$ and $h_0 = [h \mid \ell \mapsto [fields\, B \mapsto defaults]]$. In each case we construct a partition $h_0 = hOut' * hA'_1 * hRep'_1 \ldots$.

If $C \not\leq A$, then we have by the analysis that $B \not\leq Rep$. Because $rng\ (h_0\ell) \cap Loc = \varnothing$ (by definition of $defaults$), and h is confined, it follows that h_0 is confined using $hOut' = [hOut \mid \ell \mapsto [fields\, B \mapsto defaults]]$ and the rest of the partition unchanged. Now $conf\, C \eta_0\, h_0 \Leftrightarrow rng\ \eta_0 \cap locs(Rep{\downarrow}) = \varnothing$. And, $rng\ \eta_0 \cap locs(Rep{\downarrow}) = (\{\ell\} \cap locs(Rep{\downarrow})) \cup (rng\ \eta \cap locs(Rep{\downarrow})) = \{\ell\} \cap locs(Rep{\downarrow}) = \varnothing$ by assumption $conf\ C\ \eta\ h$ and since $\ell \notin locs(Rep{\downarrow})$ from $B \not\leq Rep$.

If $C < A$, then we have by the analysis that $B \not\leq A$. Because $rng\ (h_0\ell) \cap Loc = \varnothing$, and h is confined, it follows that h_0 is confined, where again the new partition adds ℓ to $hOut$. Now $conf\ C\ \eta_0\ h_0 \Leftrightarrow (rng\ \eta_0 \cap locs(Rep{\downarrow}) = \varnothing) \wedge (rng\ \eta_0 \cap locs(A{\downarrow}) = \{\eta_0\ this\})$. And, $rng\ \eta_0 \cap locs(Rep{\downarrow}) = \varnothing$ using the same reasoning as for case $C \not\leq A$. Now $(rng\ \eta_0 \cap locs(A{\downarrow})) = \{\ell\} \cap locs(A{\downarrow}) \cup (rng(\eta \downharpoonright x) \cap locs(A{\downarrow}))$. But since $B \not\leq A$, we have $\{\ell\} \cap locs(A{\downarrow}) = \varnothing$ and since $conf\ C\ \eta\ h$, we have $rng\ \eta \cap locs(A{\downarrow}) = \{\eta\ this\}$. And $\eta\, this = \eta_0\, this$ because $x \neq this$ by typing.

If $C = A$, let j be such that $\{\eta\, this\} = dom(hA_j)$. Define the new partition by cases on B. If $B \leq Rep$, define $hRep'_j = [hRep_j \mid \ell \mapsto [fields\, B \mapsto defaults]]$ and the rest the same as for the given partition of h. If $B \not\leq Rep$, define $hOut' = [hOut \mid \ell \mapsto [fields\, B \mapsto defaults]]$ and again the rest the same. (Note that $B \not\leq A$ by the analysis.) Now $conf\ h_0$ follows since $conf\ h$ and $rng\ (h_0\ell) \cap Loc = \varnothing$. To show $conf\ A\ \eta_0\ h_0$, first note that $conf\ A\ \eta\ h$ holds by assumption, so $rng\ \eta \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA_j * hRep_j)$. Now $conf\ A\ \eta_0\ h_0 \Leftrightarrow rng\ \eta_0 \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA'_k * hRep'_k)$ for some k and $dom(hA'_k) =$

$\{\eta_0\ \mathit{this}\}$. By typing rule, $x \neq \mathit{this}$, hence $\eta_0\ \mathit{this} = \eta\ \mathit{this}$. Choose $k = j$. Now $rng\ \eta_0 \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) = (\{\ell\} \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow}))) \cup (rng(\eta \restriction x) \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow}))) = (\{\ell\} \cap locs(Rep{\downarrow})) \cup (rng(\eta \restriction x) \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})))$, since by the analysis $B \not\leq A$. Now if $\ell \notin locs(Rep{\downarrow})$, the result $conf\ A\ \eta_0\ h_0$ follows from assumption $conf\ A\ \eta\ h$ as we chose $hRep'_j = hRep_j$. If $\ell \in locs(Rep{\downarrow})$, then $rng\ \eta_0 \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA_j * hRep'_j)$ and $dom(hA_j) = \{\eta_0\ \mathit{this}\}$, as we chose $hRep'_j = [hRep_j \mid \ell \mapsto [fields\ B \mapsto defaults]]$.

8 Discussion

We have given a static analysis, in the style of a typing system, for a form of instance-based confinement. The language studied includes mutable state, private fields, dynamic binding and inheritance, subsumption, casts and type tests, mutually recursive classes and mutually recursive methods.

Although earlier proposals for instance-based confinement were not formally justified and were unacceptably restrictive, recent proposals have been presented with precise typing systems [CPN98,CNP01,Mül01] and are even more general than ours, e.g., they allow controlled references out of representation objects. Unlike other work, our analysis does not require code annotations, but some annotations are likely needed for more refined and flexible notions of confinement.

We have proved that the static analysis ensures a semantically defined confinement property. Such a result is also proved in [CPN98], and argued as well in [CNP01,Mül01]. The semantic property is somewhat implicit in [Mül01], as that work is primarily concerned with a verification logic.

Although our proofs are somewhat intricate, complete details can be given in the space of a few pages despite the richness of the language. Operational semantics seems to require more complicated and less modular proofs. However, the importance of confinement has become quite clear and its formal study is immature so a variety of approaches need to be explored. We are particularly interested in modular checking of self-certifying mobile code [App01], for which purpose compositionality is important and annotation requirements should be minimized.

The central novelty of our work is that our analysis ensures a semantic confinement property that in turn has been proved to ensure an encapsulation property that is an end in itself: representation independence. A very different property, soundness of frame axioms, is proved in [LN00] and [Mül01] using forms of instance-based confinement. Our semantics and proof techniques appear robust and suited to proving such results as well.

In future work we plan to address additional language features such as concurrency, protected fields, private and protected methods, interface types, parametric polymorphism and modules. We also plan to treat more sophisticated notions of confinement, and to make a connection with capabilities or privileges as in [BNR01,CNP01]. The results in [BN02] already encompass the privilege-based access control system of Java.

References

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, 1999.

[App01] A. W. Appel. Foundational proof-carrying code. In *Proceedings of LICS*, 2001.

[BN] A. Banerjee and D. A. Naumann. A static analysis for instance-based confinement in Java, Extended version. http://www.cs.stevens-tech.edu/˜naumann/staticE.ps.

[BN02] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. To appear in *POPL*, 2002.

[BNR01] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, 1998.

[Boy01] J. Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[Car97] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[CN01] A. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. Submitted, 2001.

[CNP01] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP*, 2001.

[Coc00] M. Cochinwala. Using objects for next generation communication services. In E. Bertino, editor, *ECOOP*, 2000.

[CPN98] D. G. Clarke, J.M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.

[DLN98] D. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report 156, COMPAQ Systems Research Center, July 1998.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge, 1990.

[dRE98] W-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge, 1998.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GPV01] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, 2001.

[HHS93] C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[IO01] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[IPW99] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *OOPSLA*, 1999.

[Lea00] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000.

[LN00] K. R. M. Leino and G Nelson. Data abstraction and information hiding. Technical Report 160, COMPAQ Systems Research Center, November 2000. To appear in TOPLAS.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2), 1995.

[MPH00] P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[MS92] M. Mizuno and D. A. Schmidt. A security flow control algorithm and its denotational semantics-based correctness proof. *Formal Aspects of Computing*, 4:727–754, 1992.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Rey84] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.

[Rey01] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2001.

[Szy99] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1999.

[VB01] J. Vitek and B. Bokowski. Confined types in java. *Software Practice and Experience*, 31(6):507–532, 2001.

A Remaining cases for proof of Lemma 1

Variable x: Then $d = \eta x$. In each case, i.e., $C \not\leq A$, $C < A$, and $C = A$, the condition on d required for confinement of expression x is a direct consequence of the definition of confinement for η. For example, in the case $C \not\leq A$, we have by $conf\, C\, \eta\, h$ that $rng(\eta) \cap locs(Rep{\downarrow}) = \varnothing$, which implies the required condition $d \notin locs(Rep{\downarrow})$.

Null, type test, and equality test: In each of these cases, the value d is not in *Loc*. The confinement conditions hold for all non-*Loc* values.

Field access: $[\![\Gamma;\ C \vdash e.f : T]\!]\mu\eta h = \mathbf{let}\ \ell = [\![\Gamma;\ C \vdash e : C]\!]\mu\eta h\ \mathbf{in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ \ell \notin dom\, h\ \mathbf{then}\ \bot\ \mathbf{else}\ h\ell f$

If $C \not\leq A$, then since $\Gamma;\ C \rhd e : C$, by induction on e we have, $\ell \notin locs(Rep{\downarrow})$. It suffices to consider the case $\ell \in dom\ h$, so $d = h\ell f$. Since h is confined, consider admissible partition $(hOut * hA_1 * hRep_1 \ldots) = h$. Because $\ell \in [\![C]\!]$, we have $\ell \in dom(hOut)$. So, because $hOut \not\rightsquigarrow hRep_i$ by confinement of h, we get $d \notin dom(hRep_i)$, hence $d \notin locs(Rep{\downarrow})$.

If $C < A$ then $\ell \in locs(A{\downarrow})$ because $\ell \in [\![C]\!]$. Since $\Gamma;\ C \rhd e : C$, by induction on e we have $\ell = \eta\ this$. It suffices to consider the case $\ell \in dom\ h$, so $d = h\ell f$. Consider admissible partition $(hOut * hA_1 * hRep_1 \ldots) = h$. By visibility, we know that f is not in the private fields $\overline{g}$ of A; hence by confinement of h we have $d \notin dom(hRep_i)$, so $d \notin locs(Rep\downarrow)$. Moreover, by confinement of h again, $hA_i * hRep_i \not\leftrightarrow hA_j * hRep_j$ for $i \neq j$. Hence if $d \in locs(A\downarrow)$, it must be that $d \in dom(hA_i)$. That is, $d = \eta\, this$.

If $C = A$, then since $\Gamma;\ A \rhd e : A$, by induction on e we get, $\ell \in (locs(A\downarrow) \cup locs(Rep\downarrow)) \Rightarrow \ell \in dom(hA_j * hRep_j)$ for some j with $dom(hA_j) = \{\eta\ this\}$. It suffices to consider the case $\ell \in dom\, h$. Now, by typing, $\ell \in locs(A\downarrow)$ so $\ell \in dom(hA_j)$ for some j with $dom(hA_j) = \{\eta\ this\} = \{\ell\}$. Now $d = hA_j \ell f$, and we need to show $d \in (locs(A\downarrow) \cup locs(Rep\downarrow)) \Rightarrow d \in dom(hA_k * hRep_k)$ for some k with $dom(hA_k) = \{\eta\, this\}$. By confinement of heap h, we know that $j \neq i \Rightarrow (hA_j * hRep_j) \not\leftrightarrow (hA_i * hRep_i)$. If $d \in (locs(A\downarrow) \cup locs(Rep\downarrow))$ then, as $d \in rng(hA_j \ell)$, we have $d \in dom(hA_j * hRep_j)$. So it suffices to choose $k = j$.

B Remaining cases for proof of Lemma 2

Assignment:

$$[\![\Gamma;\ C \vdash x := e : \mathtt{com}]\!]\mu\eta h \ = \ \mathbf{let}\ d = [\![\Gamma;\ C \vdash e : T]\!]\mu\eta h\ \mathbf{in}\ ([\eta \mid x \mapsto d], h)$$

As $h_0 = h$ and $conf\ h$ by hypothesis, it suffices to show $conf\ C\ \eta_0\ h$ where $\eta_0 = [\eta \mid x \mapsto d]$.

If $C \not\leq A$, then since $\Gamma;\ C \rhd e : T$, by Lemma 1 we have, $d \notin locs(Rep\downarrow)$. Now $conf\ C\ \eta_0\ h \Leftrightarrow rng\ \eta_0 \cap locs(Rep\downarrow) = \varnothing$. And, $rng\ \eta_0 \cap locs(Rep\downarrow) = (\{d\} \cup rng\ \eta) \cap locs(Rep\downarrow) = (\{d\} \cap locs(Rep\downarrow)) \cup (rng\ \eta \cap locs(Rep\downarrow)) = \varnothing$, since $d \notin locs(Rep\downarrow)$ and $conf\ C\ \eta\ h$.

If $C < A$, then since $\Gamma;\ C \rhd e : T$, by Lemma 1 we have, $d \notin locs(Rep\downarrow)$ and $(d \in locs(A\downarrow) \Rightarrow d = \eta\ this)$. Now $conf\ C\ \eta_0\ h \Leftrightarrow rng\ \eta_0 \cap locs(Rep\downarrow) = \varnothing$ and $rng\ \eta_0 \cap locs(A\downarrow) = \{\eta_0\ this\}$. So $conf\ C\ \eta_0\ h$ holds by the above for $\eta_0\ x$ and by hypothesis $conf\ C\ \eta\ h$ for the rest of η_0 —recall that x is distinct from $this$ by typing.

If $C = A$, then since $\Gamma;\ A \rhd e : T$, by Lemma 1 we have, $d \in (locs(A\downarrow) \cup locs(Rep\downarrow)) \Rightarrow d \in dom(hA_j * hRep_j)$ for some j with $dom(hA_j) = \{\eta\ this\}$. Now $conf\ A\ \eta_0\ h \Leftrightarrow rng\ \eta_0 \cap (locs(A\downarrow) \cup locs(Rep\downarrow)) \subseteq dom(hA_k * hRep_k)$ for some k where $dom(hA_k) = \{\eta\, this\}$. As $x \neq this$ by typing, we can take $k = j$ and get $conf\ A\ \eta_0\ h$ by the above for $\eta_0\ x$ and by hypothesis $conf\ A\ \eta\ h$ for the rest of η_0.

Method call as command:

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

Then $\eta_0 = \eta$. It suffices to consider the case $\ell \in dom\ h$, otherwise the semantics is $\bot$. If $C \not\leq A$, then since $\Gamma;\ C \rhd e : D$, by induction on e we have, $\ell \notin locs(Rep\downarrow)$. Let $\eta' = [\overline{x} \mapsto \overline{d}, this \mapsto \ell]$. Below we show the claim $conf\ (loctype\, \ell)\ \eta'\ h$; from this it follows by assumption $conf\ \mu$ that $conf\ h_0$ and $conf\ C\,\eta_0\,h_0$. We defer the argument for $conf\ (loctype\ \ell)\ \eta'\ h$ because it also applies for the following case.

If $C < A$, then since $\Gamma;\ C \rhd e : D$, by induction on e we have, $\ell \notin locs(Rep{\downarrow})$ and $(\ell \in locs(A{\downarrow}) \Rightarrow \ell = \eta\ this)$. Let $\eta' = [\overline{x} \mapsto \overline{d}, this \mapsto \ell]$. We claim that $conf\ (loctype\ \ell)\ \eta'\ h$; then, we have by assumption $conf\ \mu$ that h_0 is confined and $conf\ C\ \eta_0\ h_0$.

Now we show $conf\ (loctype\ \ell)\ \eta'\ h$ for which we go by cases on $loctype\ \ell$. Recall that $\ell \notin locs(Rep{\downarrow})$, so $(loctype\ \ell) \not\leq Rep$.

(i) $loctype\ \ell < A$: By confined method signatures, $\overline{T} \not\geq Rep$ and $\overline{T} \not\geq A$, so $\overline{d} \notin locs(Rep{\downarrow})$ and $\overline{d} \notin locs(A{\downarrow})$. Now $conf\ (loctype\ \ell)\ \eta'\ h \Leftrightarrow (rng\ \eta' \cap locs(Rep{\downarrow}) = \varnothing) \wedge (rng\ \eta' \cap locs(A{\downarrow}) = \{\eta'\ this\})$. And, $rng\ \eta' \cap locs(Rep{\downarrow}) = \{\overline{d}, \ell\} \cap locs(Rep{\downarrow}) = \varnothing$, since $\overline{d} \notin locs(Rep{\downarrow})$ and $\ell \notin locs(Rep{\downarrow})$. And, $(rng\ \eta' \cap locs(A{\downarrow})) = \{\overline{d}, \ell\} \cap locs(A{\downarrow}) = \{\ell\} = \{\eta'\ this\}$.

(ii) $loctype\ \ell \not\leq A$: By confined method signatures, $\overline{T} \not\geq Rep$, so $\overline{d} \notin locs(Rep{\downarrow})$. Now $conf\ (loctype\ \ell)\ \eta'\ h \Leftrightarrow (rng\ \eta' \cap locs(Rep{\downarrow}) = \varnothing)$. And, $rng\ \eta' \cap locs(Rep{\downarrow}) = \{\overline{d}, \ell\} \cap locs(Rep{\downarrow}) = \varnothing$, since $\overline{d} \notin locs(Rep{\downarrow})$ and $\ell \notin locs(Rep{\downarrow})$.

(iii) $loctype\ \ell = A$: Then by confined method signatures, $\overline{T} \not\geq A$ and $\overline{T} \not\geq Rep$, so $\overline{d} \notin locs(A{\downarrow})$ and $\overline{d} \notin locs(Rep{\downarrow})$. Now $conf\ (loctype\ \ell)\ \eta'\ h \Leftrightarrow rng\ \eta' \cap (locs(Rep{\downarrow}) \cup locs(A{\downarrow})) \subseteq dom(hA_j * hRep_j)$ for some j where $dom(hA_j) = \{\eta'\ this\} = \{\ell\}$. And $rng\ \eta' \cap (locs(Rep{\downarrow}) \cup locs(A{\downarrow})) = \{\overline{d}, \ell\} \cap (locs(Rep{\downarrow}) \cup locs(A{\downarrow})) = \{\ell\} \subseteq dom(hA_j * hRep_j)$, for some j and $dom(hA_j) = \{\eta'\ this\} = \{\ell\}$.

Finally we consider the case $C = A$. From assumption $conf\ A\ \eta\ h$ we have $rng\ \eta \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA_j * hRep_j)$ for some j where $dom(hA_j) = \{\eta\ this\}$. Let $\eta' = [\overline{x} \mapsto \overline{d}, this \mapsto \ell]$. We claim $conf\,(loctype\ \ell)\eta' h$. Then by $conf\ \mu$ we get, $conf\ h_0$ and $conf\ A\ \eta_0\ h_0$.

It remains to prove the claim, which we do by cases on $loctype\ \ell$. For the three cases **(i–iii)** above, the argument is unchanged. That leaves the following case.

(iv) $loctype\ \ell \leq Rep$: Then $\ell \in locs(Rep{\downarrow})$ hence $\ell \in dom(hRep_j)$ for some j. Now $conf\ (loctype\ \ell)\ \eta'\ h \Leftrightarrow rng\ \eta' \cap Loc \subseteq dom(hA_j * hRep_j)$ for some j. And $rng\ \eta' \cap Loc = \{\overline{d}, \ell\} \cap Loc = \{\ell\} \cup (\overline{d} \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})))$, since by confined method signatures, $T \leq A \vee T \leq Rep$ for all $T \in \overline{U}$. Then $rng\ \eta' \cap Loc \subseteq dom(hA_j * hRep_j)$ for some j provided $\overline{d} \cap (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \subseteq dom(hA_j * hRep_j)$ for some j. This is what we now show.

Since $\Gamma;\ A \rhd \overline{e} : \overline{U}$, by Lemma 1 on $\overline{e}$ we have $\overline{d} \in (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \Rightarrow \overline{d} \in dom(hA_k * hRep_k)$ for some k with $dom(hA_k) = \{\eta\ this\}$. So the argument is complete if we choose $j = k$.

Local variable:

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

We claim *conf* $C\,[\eta \mid x \mapsto d]\,h$. From this it follows, using $(\Gamma, x : T);\ C \rhd S$ and induction on S, that *conf* h_0 and *conf* $C\,\eta_0\,h_0$. Hence *conf* $C\,(\eta_0 \lfloor x)\,h_0$.

It remains to prove the claim *conf* $C\,[\eta \mid x \mapsto d]\,h$ which we do by cases on C.

If $C \not\leq A$, then since $\Gamma;\ C \rhd e : U$, by Lemma 1 we have $d \notin locs(Rep{\downarrow})$ and hence, using hypothesis *conf* $C\,\eta\,h$, we have *conf* $C\,[\eta \mid x \mapsto d]\,h$.

If $C < A$, then since $\Gamma;\ C \rhd e : U$, by Lemma 1 we have, $d \notin locs(Rep{\downarrow})$ and $(d \in locs(A{\downarrow}) \Rightarrow d = \eta\ this)$. Hence we obtain *conf* $C\,[\eta \mid x \mapsto d]\,h$ using assumption *conf* $C\,\eta\,h$.

If $C = A$, then since $\Gamma;\ A \rhd e : U$, by Lemma 1, $d \in (locs(A{\downarrow}) \cup locs(Rep{\downarrow})) \Rightarrow d \in dom(hA_j * hRep_j)$ for some j with $dom(hA_j) = \{\eta\ this\}$. Hence we obtain *conf* $A\,[\eta \mid x \mapsto d]\,h$ using assumption *conf* $A\,\eta\,h$.

Sequence:

$$\begin{aligned}\llbracket \Gamma;\ C \vdash S_1;\ S_2 : \texttt{com} \rrbracket \mu\eta h = {} & \textbf{let}\ (\eta', h') = \llbracket \Gamma;\ C \vdash S_1 : \texttt{com} \rrbracket \mu\eta h\ \textbf{in} \\ & \llbracket \Gamma;\ C \vdash S_2 : \texttt{com} \rrbracket \mu\eta' h'\end{aligned}$$

Then $(\eta_0, h_0) = \llbracket \Gamma;\ C \vdash S_2 : \texttt{com} \rrbracket \mu\eta' h'$. Since $\Gamma;\ C \rhd S_1$ we have, by induction on S_1, *conf* h' and *conf* $C\,\eta'\,h'$. Since $\Gamma;\ C \rhd S_2$ we have, by induction on S_2, *conf* h_0 and *conf* $C\,\eta_0\,h_0$.

Conditional:

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

If $b = \mathsf{true}$, then $(\eta_0, h_0) = \llbracket \Gamma;\ C \vdash S_1 : \texttt{com} \rrbracket \mu\eta h$. Since $\Gamma;\ C \rhd S_1$, by induction on S_1 we have, h_0 is confined and *conf* $C\,\eta\,h_0$. The argument is symmetric for $b = \mathsf{false}$.