A Logic for Information Flow in Object-Oriented Programs

Torben Amtoft Sruthi Bandhakavi Anindya Banerjee

Department of Computing and Information Sciences
Kansas State University, Manhattan KS 66506, USA
{tamtoft,sruthi,ab}@cis.ksu.edu

Abstract

This paper specifies, via a Hoare-like logic, an interprocedural and flow sensitive (but termination insensitive) information flow analysis for object-oriented programs. Pointer aliasing is ubiquitous in such programs, and can potentially leak confidential information. Thus the logic employs *independence assertions* to describe the noninterference property that formalizes confidentiality, and employs *region assertions* to describe possible aliasing. Programmer assertions, in the style of JML, are also allowed, thereby permitting a more fine-grained specification of information flow policy.

The logic supports local reasoning about state in the style of separation logic. Small specifications are used; they mention only the variables and addresses relevant to a command. Specifications are combined using a frame rule. An algorithm for the computation of postconditions is described: under certain assumptions, there exists a *strongest* postcondition which the algorithm computes.

1. Introduction

An information flow policy, concerned with protecting confidentiality of data, must ensure that during program execution, data does not flow to a channel unauthorized to receive the data [10]. The typical setting for checking confidentiality of data involves channels with different clearance levels[1], e.g., High for sensitive/private channels and Low for public channels, and a program that manipulates data arriving at input channels (with different clearance levels) and produces results that may flow into output channels (with different clearance levels). In this setting, confidentiality of data can be assured provided that, during program execution, data meant for High output channels do not flow into Low output channels. Cohen [14] advanced an equivalent, deductive formulation for assuring confidentiality: from the text of the program, and by observing only the data in Low output channels (hereafter called Low outputs) an attacker cannot deduce any information about the data in High input channels (hereafter called High inputs). In other words, for confidentiality to hold, Low outputs must not depend on High inputs in any way. It is this notion of *independence* that is explored in this paper in the context of object-oriented programs.

Here are some simple examples that illustrate whether or not a program satisfies confidentiality. In each example, the variable l is a Low output and the variable h is a High input. First, the assignment $l := h$ violates confidentiality directly due to the data flow from h to l. Second, the conditional $\mathbf{if}\ h > 0\ \mathbf{then}\ l := 1\ \mathbf{else}\ l := 0$ violates confidentiality indirectly due to control flow: while neither assignment by itself violates confidentiality, information as to whether or not $h > 0$ is revealed by whether or not l is 1 after the execution. In contrast, the command $l := h\ ;\ l := 0$ satisfies confidentiality although it has a subpart that does not: no deductions can be made about the input value of h from the output value of l, since the latter is always 0.

Information flow analysis has been used to statically certify[15] that confidentiality holds in all possible execution paths of a program. Typical information flow analyses, surveyed by Sabelfeld and Myers [24], are often specified using security type systems [26, 19, 22, 6, 18]. The security guarantee provided by a well-typed program is this: no High inputs will flow to Low outputs either directly, via data flow, or indirectly, via control flow, during program execution. The type systems mentioned above, except for the recent [18], are flow insensitive, and this is a source of imprecision. Indeed, such type systems reject all the example programs above, including the benign one, for they require every subprogram be well-typed whether or not it contributes to the final answer. The subprogram, $l := h$, in the benign example, fails to type.

Extant security type systems for object-oriented programs [6, 19] have yet another source of imprecision that arises due to the way aliasing is handled. In object-oriented programs, fields of a class – in addition to program variables – are annotated with security levels. However, if an object is assigned to a High variable, then the Low fields of the object cannot be updated [6, 19]. Thus the field update, $z.info := 42$, is rejected by the security type system in case $info$ has level Low and z has level High. The reasoning is as follows: consider two Low variables, p and q, which are assigned objects o_1 and o_2 respectively. Now consider the command $\mathbf{if}\ h > 7\ \mathbf{then}\ z := p\ \mathbf{else}\ z := q$ which appears secure since a High variable is updated under a High guard. However, depending on h, either z and p are aliases of o_1, or z and q are aliases of o_2. A subsequent update of z's $info$ field will reveal information about h: if $q.info$ is not 42 after the field update, we know that $h > 7$ holds. A similar reasoning requires a method call like $x.m(y)$ to update only High fields in the body of method m, in case the receiver x is High. Such reasoning, while sound, is imprecise: aliasing may not be present at all, in which case, both the field update and the method call is benign.

Our challenges are twofold. First, we prefer a flow sensitive specification of information flow analysis. We also want to handle pointer aliasing in a manner that is more precise than extant approaches which do not perform any alias analysis.

The second challenge is to obtain a *modular* specification for an interprocedural information flow analysis. (Ideally, this would allow us to obtain a static checker for information flow). To be

[1] In general, these levels form a security lattice, with Low $\leq$ High.

specific, we want our analysis to be compositional in the *state*.[2] We want local reasoning about the *heap* where aliasing happens; this means that when we analyze a command, we are only allowed to consider the *footprint* of the command on the state, i.e., we can only consider the variables and parts of the heap that are used by the command [21, 23] – nothing else.

Contributions. The primary contribution of this paper is to meet both of the above challenges by specifying an interprocedural information flow analysis using a Hoare-like logic. Assertions in the logic are stateful and describe aliasing properties – *region assertions* – as well as information flow properties – *independence assertions*. To reason about outgoing method calls in method bodies, we require method summaries to provide a contract about assertions that must be met before a call and assertions that must hold after a call.

Importantly, the logic uses fundamental ideas from separation logic [21, 23] to provide local reasoning about state. As we clarify in the sequel, specifications in the logic are *small* or *local*: the intuition is that these specifications convey the "bare essence" of reasoning about a command. The reasoning can be elaborated in different contexts, and larger specifications may be obtained by way of a *frame rule*. Indeed, with region and independence assertions, our specification yields an interprocedural static checker for information flow.

Our second contribution is to extend the logic with programmer assertions so that a more fine-grained specification of information flow policy can be obtained. Programmer assertions can take the form "x is a constant", or "variables x and y are equal", or "$x = k(y)$", where k is a mathematical function: such assertions are also allowed, e.g., in JML [12]. In contrast to region and independence assertions, however, programmer assertions may require runtime checking or verification by a theorem prover. We show examples of the use of programmer assertions in concert with region and independence assertions for verifying observational purity [7] and for demonstrating selective dependency [14]. Nevertheless, we do not have an automatic checker in the presence of programmer assertions. At some points in the checking process, "logical implications" need to be decided. We do not know whether there exists a useful proof system to decide the logical implications. But we provide a few simple heuristics to ease the burden of checking.

A minor contribution of the paper is concerned with completeness issues for the logic with assertions restricted to region and independence assertions only. For this sub-logic, we give an algorithm that computes postconditions from preconditions and show that, under certain extra assumptions, the sub-logic is complete: there exists a *strongest* postcondition that the algorithm computes[3]. Alas, the algorithm is non-modular. The main difficulty lies with interprocedural analysis, for which the procedure summaries must be discovered and updated on the fly. We leave this issue for a future paper.

2. Examples

Local Reasoning about Aliasing. Recall that local reasoning about a command entails reasoning only about the *footprint* of the command. In the command, $z.info := 42$, for example, reasoning is permitted only with variable z, the location in the heap that z denotes, and the contents of the $info$ field – nothing else. Since we are interested in static checking, we need to abstract the concrete heap location denoted by z.

Abstract locations (as in, e.g., [20]) are used to abstract sets of concrete heap locations. A region assertion $x \rightsquigarrow L$, read "x at L", asserts that L abstracts the concrete location denoted by x.

Suppose two abstract locations L_1 and L_2 are disjoint, i.e., they abstract two disjoint sets of concrete locations. Then, if $x \rightsquigarrow L_1$ and $y \rightsquigarrow L_2$ hold, we infer that x, y *must not* alias a concrete location. (In contrast, if L_1, L_2 are not disjoint, then x, y *may* alias).

Region assertions may also take the form $L_1.f \rightsquigarrow L_2$, so as to deal with aliasing caused by heap-allocated values, e.g, $x.f$. The intuition is that for any concrete location ℓ_1 that is abstracted by L_1, if field f of ℓ_1 contains concrete location ℓ_2, then ℓ_2 is abstracted by L_2.

We now show two examples in which region assertions are used to reason locally about aliasing. Consider a method $getNode$ which, given the head of a linked list and an integer i, returns the node at position i in the list. Each node has two fields: $data$ denoting the value in the node, and $next$ denoting the next node in the list. We consider two implementations of $getNode$: in the first, a pointer to the ith node is returned, creating an alias; in the second, a copy of the ith node is returned – this does not create an alias. The bodies of $getNode$ for the two implementations are shown below; the distinguished variable, $\mathbf{result}$, holds the return result of a method.

$$
\begin{array}{l}
n := head;\ j := 0; \\
\mathbf{while}\ (n \neq \mathbf{null})\ \&\&\ (j < i)\ \mathbf{do} \\
\qquad \{n := n.next;\ j := j + 1;\} \\
\mathbf{result} := n
\end{array}
$$

Example 1: Node i is aliased

$$
\begin{array}{l}
n := head;\ j := 0; \\
\mathbf{while}\ (n \neq \mathbf{null})\ \&\&\ (j < i)\ \mathbf{do} \\
\qquad \{n := n.next;\ j := j + 1;\} \\
\mathbf{if}\ n \neq \mathbf{null\ then} \\
\qquad \{newNode := \mathbf{new}\ Node; \\
\qquad newNode.data := n.data;\ newNode.next := \mathbf{null}; \\
\qquad \mathbf{result} := newNode;\} \\
\mathbf{else}\ \{\mathbf{result} := \mathbf{null}\}
\end{array}
$$

Example 2: Node i is not aliased

Consider the first two commands of Example 1, where we assume that L is the abstract location in which the list is allocated. Because $head$ points to the first node in the list, $head \rightsquigarrow L$ is part of the precondition of the program, which also contains the assertion $L.next \rightsquigarrow L$. For the command $n := head$, we get the small specification:

$$\{head \rightsquigarrow L\}\ n := head\ \{n \rightsquigarrow L\}$$

The specification says that from precondition $head \rightsquigarrow L$, the postcondition $n \rightsquigarrow L$ can be asserted. Note how the region assertions in the specification mention facts about $head$ and n, nothing else. Next, for the command $j := 0$, we get the small specification[4] $\{true\}\ j := 0\ \{j \rightsquigarrow \mathbf{int}\}$. To combine the specifications for the two commands above, we use, in a manner similar to separation logic, a frame rule (also see [11]): because n is not modified by $j := 0$, the frame rule allows us to add $n \rightsquigarrow L$ as conjunct to both its pre- and postconditions. To wit:

$$\{n \rightsquigarrow L\}\ j := 0\ \{j \rightsquigarrow \mathbf{int}, n \rightsquigarrow L\}$$

[2] It is not compositional reasoning per se we are interested in, since it is "perfectly possible to be compositional and global (in the state) at the same time, as was the case in early denotational models of imperative languages" [21].

[3] By "strongest" postcondition we mean the strongest among the assertions accepted by our logic, rather than the strongest among the assertions which are "semantically correct" (a larger set).

[4] The assertion $j \rightsquigarrow \mathbf{int}$, expressing that j has an integer value, is strictly speaking redundant, since we shall assume that we are dealing with "well-typed" programs where a variable/field may contain an integer iff it has been assigned the type $\mathbf{int}$. Therefore such assertions may be omitted.

Now the two specifications can be combined to obtain the following specification for the sequential composition, $n := head;\ j := 0$.

$$\{head \rightsquigarrow L\}\ n := head;\ j := 0\ \{j \rightsquigarrow \mathbf{int},\ n \rightsquigarrow L\}$$

The invariant for the while loop is $\{n \rightsquigarrow L, L.next \rightsquigarrow L\}$, which we may write in abbreviated form as $\{(n, L.next) \rightsquigarrow L\}$. To show that the preamble establishes this invariant from the program's precondition, we may apply the frame rule once more on the above specification, adding $L.next \rightsquigarrow L$ to both pre- and postcondition; this is valid since no $next$ field is modified by the preamble. Thus:

$$\{(head, L.next) \rightsquigarrow L\}\ n := head;\ j := 0\ \{(n, L.next) \rightsquigarrow L\}$$

To show that the invariant is maintained by the while loop, we show the stronger property that each assignment in the loop body maintains the invariant. For $n := n.next$ the small specification is $\{(n, L.next) \rightsquigarrow L\}\ n := n.next\ \{n \rightsquigarrow L\}$. Now the frame rule (applicable since no $next$ field is modified) gives us

$$\{(n, L.next) \rightsquigarrow L\}\ n := n.next\ \{(n, L.next) \rightsquigarrow L\}$$

In a similar (but simpler) way, we can show $\{(n, L.next) \rightsquigarrow L\}\ j := j + 1\ \{(n, L.next) \rightsquigarrow L\}$. Finally, for $\mathbf{result} := n$, the small specification is

$$\{n \rightsquigarrow L\}\ result := n\ \{result \rightsquigarrow L\}$$

By a few more applications of the frame rule, we obtain the following specification for the body, B_1, of $getNode$.

$$\{(head, L.next) \rightsquigarrow L\}\ B_1\ \{(n, L.next, result) \rightsquigarrow L\}$$

As expected, n and $result$ *may alias* the same location in the heap.

In Example 2, the precondition for the entire method body, B_2, of $getNode$ is the same as that of B_1, namely, $(head, L.next) \rightsquigarrow L$. The crucial difference is the occurrence of the command $newNode := \mathbf{new}\ Node$ where we may choose an arbitrary abstract location to abstract the concrete location being created. Choosing L_1, we get the small specification

$$\{true\}\ newNode := \mathbf{new}\ Node\ \{newNode \rightsquigarrow L_1\}\ .$$

Applying the frame rule repeatedly, we can derive postcondition[5]

$$\{(n, L.next) \rightsquigarrow L, (result, newNode) \rightsquigarrow L_1, L_1.next \rightsquigarrow \bot\}$$

for B_2. The key observation is that provided L and L_1 are disjoint, n and $result$ *must not alias* the same location in the heap.

Information Flow Analysis and Independences. A baseline correctness property for information flow analysis is noninterference [17] (the negation of Cohen's notion of dependency [14]) which is formalized via an "indistinguishability" relation on states. Two states are indistinguishable if they agree on values of their Low variables (but may differ on values of High variables). Noninterference holds if any two runs of a program starting in two initially indistinguishable states, yield two final states that are also indistinguishable. In other words, a program is noninterfering, if for any pair of runs, changes to its High input variables are unobservable via its Low output variables; hence, reverting to a point made in the introduction, Low outputs are *independent* of High inputs.

The small specifications of our analysis are designed to answer the following question, encompassing noninterference as a special case[6]: *given two runs which initially agree on variables* $x_1 \ldots x_n$*, will they at the end agree on variables* $y_1 \ldots y_m$*?* Accordingly, we introduce *independence assertions* of the form $x\ltimes$, such that a positive answer to the above question amounts to the specification $\{x_1\ltimes, \ldots, x_n\ltimes\}\ \_\ \{y_1\ltimes, \ldots, y_m\ltimes\}$. In general, we shall consider assertions of the form $a\ltimes$, where a is an *abstract address*: either a variable, or a field access of the form $L.f$.

Leveraging the above reading of noninterference, Amtoft and Banerjee specified, as a Hoare-like logic, a termination insensitive information flow analysis for simple imperative programs [2] (later extended to a termination *sensitive* analysis [3]). This paper extends that logic to handle programs written in a core, Java-like, object-oriented language. Also, unlike [2, 3], this paper employs a standard style semantics.

Aliasing, Independences and Local Reasoning. We consider the following example adapted from Askarov's master's thesis [4].

```
class X{
  int q;
  int getQ(){result := self.q};
  unit setQ(int n){self.q := n}}
```

What can we say about the body of $getQ$? First, we consider region assertions. Suppose assertions $self \rightsquigarrow \rho_1$ and $\rho_1.q \rightsquigarrow \mathbf{int}$ hold for the precondition of $getQ$. Then we can assert that $result \rightsquigarrow \mathbf{int}$ holds in the postcondition of $getQ$. Think about ρ_1 as a metavariable which will be instantiated by abstract locations at the point of call. For instance, if the receiver in the call to $getQ$ is at abstract location L, then ρ_1 will be substituted by L.

Next, we consider independence assertions. Given that $self \rightsquigarrow \rho_1$ holds for the precondition of $getQ$, we want to check whether the postcondition contains $result\ltimes$. That is, under which conditions will two runs agree on the final value of $result$? For that to be the case, the runs must agree on the initial value of $self.q$, a sufficient condition for which is that $\rho_1.q\ltimes$ holds in the precondition; also (since $self.q$ depends on $self$), the runs must agree on $self$. A convenient *method summary* for $getQ$ is thus the following

$$\{self \rightsquigarrow \rho_1, self\ltimes, \rho_1.q\ltimes\}\ getQ\ \{result\ltimes\}\ .$$

On the other hand, if the independence assertions in the precondition do not hold at the point of call, we are unable to conclude $result\ltimes$ in the postcondition.

In a similar manner, we can compute the following method summary for $setQ$:

$$\{self \rightsquigarrow \rho_1, self\ltimes, n\ltimes, \rho_1.q\ltimes\}\ setQ\ \{\rho_1.q\ltimes\}$$

This says that in order for two runs to agree on the final value of the q fields of "corresponding" (as formalized in Sec. 4) objects abstracted by ρ_1, they must agree on the initial value of n, and on the initial value of $self$ (as otherwise, the two runs would update non-corresponding objects). Also, because there may be other objects abstracted by ρ_1 than the one which $self$ points to (and these objects did not have their q field updated), the runs must agree on the initial value of all q fields; this requirement can be omitted in the case where ρ_1 abstracts one concrete location only, i.e., in the case of "strong update".

Now consider the program

```
X x1;  X x2 := new X;
x1 := x2; //alias created
x1.setQ(secret);
z := x2.getQ()
```

where, because x_1 and x_2 are aliases, the value of $secret$ is leaked to z. Let us see how checking independences might help detect the leak. We recall what noninterference means: two runs that initially agree on all variables except for $secret$, must agree on the final value of z. A proof of noninterference, in our framework, would thus amount to establishing a specification where $z\ltimes$ is in the postcondition, *without* having to assume that $secret\ltimes$ is in the precondition. Below, we argue that this is impossible.

[5] The abstract location $\bot$ abstracts null pointers only.

[6] As can be seen by letting $x_1 \ldots x_n$, and $y_1 \ldots y_m$, be the Low variables.

First assume that the location allocated by **new** is abstracted by L_2; then we have $x_2 \leadsto L_2$ and $x_1 \leadsto L_2$. With the aim of proving that $z\ltimes$ holds after the call to $getQ$, we consult the method summary for $getQ$ where we substitute $self$ by x_2, and $result$ by z, and ρ_1 by L_2. Looking at the resulting precondition, we see that we must show that $x_2\ltimes$ and $L_2.q\ltimes$ holds before the call to $getQ$, that is, after the call to $setQ$. We therefore consult the summary for $setQ$ where we substitute $self$ by x_1, n by $secret$, and ρ_1 by L_2. Looking at the resulting precondition, we see that we must at least show that $secret\ltimes$ holds. But this yields the desired contradiction.

Suppose the aliasing were removed in a slight modification of the above program, where z is once again the output variable:

$$X\ x_1 := \mathbf{new}\ X;\ \ X\ x_2 := \mathbf{new}\ X; \text{//no alias}$$
$$x_1.setQ(secret);\ \ z := x_2.getQ()$$

Now x_1 and x_2 do not alias the same heap location. The postcondition for the first assignment asserts $\{x_1 \leadsto L_1, x_1\ltimes\}$, and that for the second asserts $\{x_2 \leadsto L_2, x_2\ltimes\}$, where L_1 and L_2 are assumed disjoint to reflect the absence of aliasing. As before, to establish that $z\ltimes$ holds after the call to $getQ$, we must show that $x_2\ltimes$ and $L_2.q\ltimes$ holds after the call to $setQ$. But since locations abstracted by L_2 are not modified by the call to $setQ$, this follows from the frame rule (since we may assume that $L_2.q\ltimes$ holds before the call). In summary, because of the absence of aliasing, the assertion $z\ltimes$ does hold finally, even if $secret\ltimes$ does not hold initially. This is in contrast to the previous example.

It is instructive to see how an existing type-based information flow analysis system, like Jif [19], handles the above programs. Assume that the variables $secret$ and x_1 are typed High, and x_2 and field q are typed Low. Since q is Low, the method $setQ$ has a *begin label* of Low, which says that the method can only be called if the program counter of the caller is no more restrictive than Low. But the level of the receiver (x_1) is High. This is one reason why Jif rejects this program. In general, the above check ensures that if there are any low aliases of x_1 in the future – e.g., x_2 in the first program – they should not be able to read the value of q assigned by $setQ$. In the second example there is no aliasing. Yet, Jif rejects this example also, because the call to $setQ$ is untypable.

Programmer assertions. As noted earlier, apart from region assertions and independence assertions, we also allow programmer assertions in code. For example, for the trivial program $\mathbf{if}\ x > 0\ \mathbf{then}\ w := 7\ \mathbf{else}\ w := 7$, clearly $w\ltimes$ holds (two runs will *always* agree on the final value of w), although a naïve analysis cannot prove the assertion. However, armed with the programmer assertion that w is a specific constant after the conditional, the following reasoning is sound in our framework: w being constant "logically implies" (defined in Sec. 4) that $w\ltimes$ holds.

We show two more examples of programmer assertions. The first concerns *observational purity* [7]. Assume we repeatedly need to apply a function $expensive(z)$, the computation of which is very expensive. To save time, we decide to memoize the most recent call[7]. For that purpose, we introduce a class M, with fields $marg$ and res obeying the invariant

$$(marg \neq 0)\ \Rightarrow\ (res = expensive(marg))$$

and with a method

```
int cexp(int z){
  if z = self.marg
  then result := self.res
  else //compute expensive(z) and store the value in result
    result := expensive(z); self.marg := z; self.res := result
  assert (result = expensive(z))}
```

[7] The generalization to full memoization appears in Sec. 6.

Obviously, the last assertion should not be checked at runtime (this would defy the purpose of memoization), but might instead be verified by a theorem prover, using the above-mentioned invariant.

Suppose we know that for $cexp$: (a) its *result* depends *only* on z, not on memo data ($marg$ or res) and (b) its *computation* affects *only* an abstract location L_1. If L_1 is not used elsewhere, we can consider calls to $cexp$ "observationally pure" [7]; this notion of purity is under consideration for extending JML [12] which currently disallows effectful method calls in assertions.

It remains to show (a) and (b). Indeed, in Sec. 4.1, we will see that from $z\ltimes$ and the programmer assertion, $\mathbf{result} = expensive(z)$, we can derive $result\ltimes$. Hence it is easy to see that if $self \leadsto L_1$ and $z\ltimes$ are preconditions for $cexp$, then $result\ltimes$ is a valid postcondition for $cexp$. We also observe that $L_1.marg$ and $L_1.res$ are the only abstract addresses that may be modified by $cexp$. This information appears in the following method summary for $cexp$:

$$\{self \leadsto L_1,\ z\ltimes\}\ \_\ \{result\ltimes\}\ [L_1.marg, L_1.res].$$

Our second example with programmer assertions deals with *selective dependency* and we consider an example due to Cohen [14]: the command $b := x + a \bmod 4$ where, clearly, b is not independent of a. However, only the lower order two bits of a are revealed to b; nothing else is revealed. Suppose we fix the lower order two bits of a to 3, i.e., $a \bmod 4 = 3$. Then we can prove that the "rest of a is protected from b", by means of the derivation[8]

$$\begin{array}{ll}
\{x\ltimes\} & \\
\quad \mathbf{assert}\ a \bmod 4 = 3; & \\
\{a \bmod 4 = 3, x\ltimes\} & \\
\{(a \bmod 4)\ltimes,\ x\ltimes\} & \text{(by logical implication)} \\
\quad b := x + a \bmod 4; & \\
\{b\ltimes\} &
\end{array}$$

That is, $b\ltimes$ is in the postcondition, under the assumption that $x\ltimes$ is in the precondition, but *without* assuming that $a\ltimes$ is too.

The Rest of the Paper

Sec. 3 formalizes the language. Sec. 4 gives the syntax and semantics of assertions. Sec. 5 specifies the logic. The full memoization example, illustrating reasoning in the logic, appears in Sec. 6. Sec. 7 is about the computing of assertions and strongest postcondition. Sec. 8 concludes. All proofs appear in the companion technical report [1].

3. Language: syntax and semantics

Syntax. Our core language (Figure 1) is a class-based object-oriented language with recursive classes, methods and field update. To avoid clutter, unlike the technical report [1] we do not consider subclassing (and thus neither dynamic dispatch, nor cast, nor type test). The grammar is based on given sets of class names (with typical element C), expressions (E), constants ranging over integers (c), field names (f), and method names (m). The names x, y, z, w are used for program variables, and k is used for mathematical functions (e.g., mod).

The BNF is self-explanatory. One difference from usual security-typed languages is that programmer assertions are allowed via the command $\mathbf{assert}\ \theta$. Conjunctions and disjunctions of programmer assertions are also allowed. A type is either a base type **int**, or a "class type", i.e., a class name C; like Java, we have nominal (by name) typing. We assume a function, $type$, that assigns a type to all program variables and to all fields. We also assume the existence of a class table, CT, that maps a class name to the corresponding class declaration. A class declaration consists of a class name, e.g.,

[8] The technical development in this paper does not allow assertions $E\ltimes$ with E an expression, but it is straightforward to add them.

T	$::=$	$\mathbf{int} \mid C$ data type
CL	$::=$	$\mathbf{class}\ C\ \{\ \overline{T}\ \overline{f}; \overline{M}\ \}$ class declaration
M	$::=$	$T\ m(U\ u)\ \{S\}$ method declaration
S	$::=$	$x := E \mid x.f := y$ assign to variable, to field
	$\mid$	$x := \mathbf{new}\ C \mid x := y.f$ object construction, field access
	$\mid$	$x := y.m(z) \mid S\ ; S$ method call, sequence
	$\mid$	$\mathbf{if}\ x\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{while}\ x\ \mathbf{do}\ S$ conditional, while
	$\mid$	$\mathbf{assert}\ \theta$ programmer assertion
E	$::=$	$x \mid c \mid \mathbf{null} \mid E\ \mathsf{op}\ E \mid k(E)$
		variable, constants, arith. operations, arith. functions
θ	$::=$	$x = c \mid x = y \mid x = k(E) \mid \ldots$ primitive assertions
	$\mid$	$\theta \wedge \theta \mid \theta \vee \theta$

Figure 1. BNF of language

C, together with a list of *public* field declarations, e.g., $\overline{T}\ \overline{f}$, and a list of method declarations, e.g., $\overline{M}$. Consider a method m declared as $T\ m(U\ u)\ \{S\}$ in class C; such a method has return type T, and formal parameter type U, and body S where S is a command. We employ a distinguished variable $result$ such that the effect of an explicit return expression, $\mathtt{return}\ E$, can be achieved by letting the last assignment of S be $result := E$. We will assume that only well-typed programs are checked.

Semantics. We specify the semantics in relational style; such a semantics fits well with a Hoare-style partial correctness specification and eases the proofs, especially since our analysis is termination insensitive. After a brief description of the semantic domains involved, we define the semantics of commands and finally the semantics of well formed class tables.

The state of a method in execution comprises a store, s, and a heap, h. A store s (in semantic domain $Store$) assigns values to local variables and parameters, where values are integer constants or locations or the distinguished entity nil (which is *not* a location). We use v to range over values, and assume that Val, the set of all values, is partitioned into two disjoint parts, True and False. For locations, we assume given a countable set Loc ranged over by ℓ. We assume each location ℓ has a class C associated with it, and write $type\ \ell = C$. For all constants c we write $type\ c = \mathbf{int}$. For each type, we define a default value of that type: $default(\mathbf{int}) = 0$ and $default(C) = nil$.

A heap h (in semantic domain $Heap$) is a finite partial function from locations to object states, where an object state is a total mapping from field names to values. With abuse of notation, we say that location ℓ is in the range of heap h if there exists location ℓ_0 in $dom(h)$ and a field f such that $\ell = h\,\ell_0\,f$. We will work with *self-contained* states: say that state (s, h) is self-contained iff (a) for all ℓ in the range of s, ℓ is in the domain of h; and (b) for all ℓ in the range of h (c.f. above), ℓ is in the domain of h.

The meaning, $[\![E]\!]$, of an expression, E, is a function from $Store$ to Val; its definition is standard and thus elided. Pointer arithmetic is disallowed: in an expression $E_1\ \mathsf{op}\ E_2$, each $[\![E_i]\!]s$ has to evaluate to an integer. The meaning of an assertion θ is a predicate on states: $[\![\theta]\!] \in Store \times Heap \rightarrow Bool$.

The semantics of a class table is a method environment μ which provides a relational meaning, $\mu(C, m)$, for each method m declared in class C. The method environment μ is computed using a fixpoint construction. For each class C and method name m, $\mu(C, m) \subseteq (Store \times Heap) \times (Val \times Heap)$.

Because a command S may contain method calls as constituents, the meaning of S is with respect to a method environment μ. More precisely, $[\![S]\!]\mu$ is a relation on input and output states: $[\![S]\!]\mu \subseteq (Store \times Heap) \times (Store \times Heap)$. The relational semantics of commands appears in Table 1. We explain the cases [$\mathsf{FieldUpd}$], [New] and [$\mathsf{MethodCall}$] below.

In field update, $x.f := y$, the heap h_0 is updated with the value of y at field f of location ℓ, where ℓ is the meaning of x. (We use the notation $[h_0 \mid \ell.f \mapsto v]$ to denote the update of the object state $h_0\,\ell$ at field f by v).

In object allocation, $x := \mathbf{new}\ C$, a fresh location ℓ of type C is allocated in the heap; the resulting store maps x to ℓ. The resulting heap, h, is the old heap, h_0, with its domain extended with ℓ. Each field f of C in the object state $h\,\ell$ is initialized to the default value of $type(f)$; this is captured by the notation $[h_0 \mid \ell \mapsto defaults]$.

For a method call, $x := y.m(z)$, suppose that y denotes a location ℓ with $type\ \ell = C$, where class C contains a method m with formal parameter u (written $pars(m, C) = u$). Let the initial state be (s_0, h_0), and suppose that the meaning of the method m is looked up in method environment μ, using a state whose heap component is h_0 but whose store component is a "local store", s_0', that binds $self$ to ℓ and u to $s_0(z)$. Let the method meaning relate (s_0', h_0) to (v, h), where v is the return result of the method, and h the updated heap. Upon return, local store s_0' is discarded, and the resulting state is heap h together with the initial store, s_0, with x updated to v.

Observe that for some (s_0, h_0) there may be no (s, h) with $(s_0, h_0)\,[[\![S]\!]\mu]\,(s, h)$. This will be the case in the event of an infinite computation, a run-time error (like dereferencing a null pointer), or a failed programmer assertion.

We are now ready for the semantics of a class table, CT. The semantics makes explicit the fixpoint computation alluded to earlier.

DEFINITION 3.1 (**Semantics of class table,** CT). $[\![CT]\!]$ *is the least upper bound (wrt. subset inclusion) of the ascending chain μ_n ($n \in \mathsf{Nats}$) of method environments, defined as follows (where class C contains method m with body S):*

$$\begin{array}{l} \mu_0(C, m) = \varnothing \\ (s_0, h_0)\,(\mu_{n+1}(C, m))\,(v, h) \iff \\ \qquad \exists s \cdot (s_0, h_0)\,[[\![S]\!]\mu_n]\,(s, h) \wedge\ (v = s(result)) \end{array}$$

Letting $\mu = [\![CT]\!]$, a key lemma is that for all self-contained states (s_0, h_0), if $(s_0, h_0)\,[[\![S]\!]\mu]\,(s, h)$ holds then (s, h) is self-contained with $dom(s_0) \subseteq dom(s)$ and $dom(h_0) \subseteq dom(h)$. In the sequel, we will tacitly assume that all states (s, h) are self-contained.

Modification of state. Sec. 2 presented several examples of local reasoning that were justified by the frame rule. Such reasoning is sound because a side condition holds for the frame rule: when the small specification of a command is extended with other assertions, the abstract addresses mentioned in the assertions are *disjoint* from the corresponding abstract addresses *modified by* the command. Both notions are made precise in Sec. 4. But first Definition 3.2 states precisely what it means to modify concrete locations occurring in heaps and stores.

DEFINITION 3.2. *For a location ℓ of type C, and for a field f of C, say that $\ell.f$ is* modified from heap h to heap h' *if $\ell \in dom(h')$ and either of the following conditions hold: (a) $\ell \in dom(h)$, and $h'\ell f \neq h\ell f$; (b) $\ell \notin dom(h)$, and $h'\ell f \neq default(type\ f)$.*

Variable x is modified from store s to store s' if $x \in dom(s')$ and either of the following conditions hold: (a) $x \in dom(s)$, and $s(x) \neq s'(x)$; (b) $x \notin dom(s)$.

4. Assertions

This section formalizes abstract locations, and provides the syntax and semantics of assertions. It also makes precise the two main

[Assert]	$(s_0,h_0)\,[\![\![\mathbf{assert}\ \theta]\!]\mu]\,(s,h) \iff [\![\theta]\!](s_0,h_0) \wedge\ s=s_0 \wedge\ h=h_0$	[Assign]	$(s_0,h_0)\,[[\![x:=E]\!]\mu]\,(s,h) \iff (\exists v\cdot v=[\![E]\!]s_0 \wedge\ s=[s_0\mid x\mapsto v]) \wedge\ h=h_0$
[FieldAcc]	$(s_0,h_0)\,[[\![x:=y.f]\!]\mu]\,(s,h) \iff \exists \ell\in Loc\cdot(s_0(y)=\ell \wedge s=[s_0\mid x\mapsto h_0\,\ell\,f]) \wedge h=h_0$	[FieldUpd]	$(s_0,h_0)\,[[\![x.f:=y]\!]\mu]\,(s,h) \iff s=s_0 \wedge\ \exists\ell\in Loc\cdot(s_0(x)=\ell \wedge h=[h_0\mid \ell.f\mapsto s_0(y)])$
[New]	$(s_0,h_0)\,[[\![x:=\mathbf{new}\ C]\!]\mu]\,(s,h) \iff \exists\ell\cdot(\mathit{type}\,\ell=C\ \wedge\ \ell\notin rng(s_0)\wedge \ell\notin dom(h_0)\ \wedge\ \ell\notin rng(h_0)\wedge s=[s_0\mid x\mapsto\ell]\wedge h=[h_0\mid \ell\mapsto \mathit{defaults}])$	[MethodCall]	$(s_0,h_0)\,[[\![x:=y.m(z)]\!]\mu]\,(s,h) \iff (s_0',h_0)\,\mu(C,m)\,(v,h) \wedge\ s=[s_0\mid x\mapsto v]$ where $\ell=s_0(y)$ and $C=\mathit{type}\ \ell$ and $s_0'=[pars(m,C)\mapsto s_0(z),\mathbf{self}\mapsto\ell]$

Table 1. Semantics of commands: excerpts

ingredients of the frame rule alluded to in Sec. 2, namely, the modification of abstract addresses, and disjointness. The frame rule can only be applied when an assertion is disjoint from the set of abstract addresses that may be modified by a command.

Abstract Locations. We let L range over the set of abstract locations, $AbsLoc$. Think of L as a token that stands for a set of concrete heap locations. We will consider the following relations on $AbsLoc$: a partial ordering relation, $L_1 \preceq L_2$, conveys that L_2 contains at least those concrete heap locations that L_1 contains. We also need a symmetric relation, $L_1 \diamond L_2$, pronounced "L_1 is disjoint from L_2", to convey that L_1 and L_2 have no concrete heap locations in common. We add a special element $\perp$ to $AbsLoc$ so that for all $L \in AbsLoc$ we have $\perp \preceq L$ and $\perp \diamond L$. One can think of $\perp$ as the counterpart of the concrete value nil.

We assume that if $L_1 \preceq L_2$ and $L \diamond L_2$ then also $L \diamond L_1$. We let LI range over $AbsLoc \cup \{\mathbf{int}\}$. And, we let X range over sets of abstract addresses.

Syntax of assertions As noted in Sec. 2, we have three kinds of primitive assertions, namely, region assertion, independence assertions, and programmer assertion. The BNF of assertions is this:

$$\phi \quad ::= \quad \theta \mid x \rightsquigarrow LI \mid L.f \rightsquigarrow LI \mid x\ltimes \mid L.f\ltimes \mid \mathit{true} \mid \phi\wedge\phi$$

An assertion is thus a (possibly empty) conjunction of primitive assertions. Recall from Sec. 2 that we shall often use the set notation to denote conjunctions of assertions. For simplicity, this paper allows disjunction only in programmer assertions, although the technical report allows arbitrary disjunctions of assertions.

Roughly, the meaning of $x \rightsquigarrow L$ in a state (s,h) is that the concrete heap location denoted by x is abstracted by L. The meaning of $a\ltimes$ is that the *two* current states in question, say (s,h) and (s_1,h_1), agree on the value of a; agreement implies that there is no leak of information via a. This intuition leads to the one-state and two-state semantics for assertions in the sequel.

One-state Semantics of Assertions. To give a precise meaning to assertions, we need to assume the existence of an extraction relation, η, (similar to the extraction functions described in [20, p.235]) that relates locations to abstract locations. We require that η satisfy the following properties: (a) If $L_1 \preceq L_2$ and $\ell\,\eta\,L_1$ then $\ell\,\eta\,L_2$; (b) If $L_1 \diamond L_2$ then for no ℓ we have $\ell\,\eta\,L_1$ and $\ell\,\eta\,L_2$; (c) $\ell\,\eta\perp$ holds for no ℓ. For convenience, we extend η to Val, so that $c\,\eta\,\mathbf{int}$ and $nil\,\eta\perp$ – thus $nil\,\eta\,L$ holds for all L. But $c\,\eta\,L$ holds for no L, and $\ell\,\eta\,\mathbf{int}$ holds for no ℓ, and $nil\,\eta\,\mathbf{int}$ does not hold.

We say that η *is over* h if $\ell\,\eta\,L$ implies $\ell\in dom(h)$. For η over h, we are now in a position to define the semantics of an assertion ϕ in state (s,h), written, $(s,h)\models_\eta \phi$.

$$\begin{array}{lcl}
(s,h)\models_\eta \theta & \iff & [\![\theta]\!](s,h)\\
(s,h)\models_\eta x\rightsquigarrow LI & \iff & s(x)\,\eta\,LI\\
(s,h)\models_\eta L.f\rightsquigarrow LI & \iff & \forall\ell\in dom(h)\cdot \ell\,\eta\,L\Rightarrow\\
 & & \quad (h\ell f)\,\eta\,(LI)\\
(s,h)\models_\eta x\ltimes & \iff & \mathit{true}\\
(s,h)\models_\eta L.f\ltimes & \iff & \mathit{true}\\
(s,h)\models_\eta \mathit{true} & \iff & \mathit{true}\\
(s,h)\models_\eta \phi_1\wedge\phi_2 & \iff & (s,h)\models_\eta\phi_1 \text{ and } (s,h)\models_\eta \phi_2
\end{array}$$

Two-state Semantics of Assertions. Consider, e.g., the assertion $x\ltimes$ and consider two states (s,h) and (s',h') for which we want the values of x to agree. If x denotes a location then, because of different allocation behavior in h and h', we cannot expect $s(x)$ and $s'(x)$ to be equal. Rather we expect the former to yield location ℓ and the latter to yield location ℓ', so that the agreement can be enforced by a bijection β that relates ℓ and ℓ'. On the other hand, not all locations need to be related to some other location, similar to what is the case for type-based information flow analysis [6]. There, the indistinguishability relation on states (s,h) and (s',h') is formalized using a bijection between those locations in $dom(h)$ and $dom(h')$ that are visible to a "low observer".

We formalize the above intuition. Let β range over bijections from a subset of Loc to a subset of Loc. That is, if $\ell\,\beta\,\ell_1$ and $\ell\,\beta\,\ell_2$ then $\ell_1=\ell_2$, but for some ℓ_0 there might not be any ℓ' such that $\ell_0\,\beta\,\ell'$; and if $\ell_1\,\beta\,\ell$ and $\ell_2\,\beta\,\ell$ then $\ell_1=\ell_2$, but for some ℓ_0 there might not be any ℓ' such that $\ell'\,\beta\,\ell_0$. In addition, with abuse of notation, for all integer constants c we shall assume that $c\,\beta\,c$, and also assume that $nil\,\beta\,nil$. We say that β *is over* $h\&h_1$ if $\ell\,\beta\,\ell_1$ implies $\ell\in dom(h)$ and $\ell_1\in dom(h_1)$.

We can now define the two-state semantics of assertion ϕ, written $(s,h)\&(s_1,h_1)\models_{\beta,\eta,\eta_1}\phi$. Here β is over $h\&h_1$, and η is over h, and η_1 is over h_1; further, if $\ell\,\beta\,\ell_1$ then $\ell\,\eta\,L$ iff $\ell_1\,\eta_1\,L$. The last condition simply says that concrete locations ℓ and ℓ_1 related by β are abstracted to the *same* abstract location L by both η and η_1.

$$\begin{array}{l}
(s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1} x\ltimes \iff (s\ x)\,\beta\,(s_1\ x)\\
(s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1} L.f\ltimes \iff\\
\quad\forall\ell\in dom(h),\,\ell_1\in dom(h_1)\cdot\\
\quad\ell\,\beta\,\ell_1\wedge\ell\,\eta\,L\Rightarrow(h\,\ell\,f)\,\beta\,(h_1\,\ell_1\,f)\\
(s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1}\phi\iff\\
\quad(s,h)\models_\eta\phi \text{ and } (s_1,h_1)\models_{\eta_1}\phi,(\phi\ \text{ is }\ \theta, x\rightsquigarrow L, L.f\rightsquigarrow LI)\\
(s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1}\mathit{true}\iff\mathit{true}\\
(s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1}\phi_1\wedge\phi_2\iff\\
\quad(s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1}\phi_1 \text{ and } (s,h)\,\&\,(s_1,h_1)\models_{\beta,\eta,\eta_1}\phi_2
\end{array}$$

Modification of Abstract Addresses. We now specify the conditions under which an abstract address X is modified from state (s,h) to state (s',h') under extraction relation η over heap h'. This is written, $(s,h)\to(s',h')\models_\eta X$. The abstract address X overapproximates the set of concrete locations that may be modified from (s, h) to (s', h').

DEFINITION 4.1 (**Modifying an abstract address**). *Say that* $(s, h) \rightarrow (s', h') \models_\eta X$ *iff*

(a) *for all* y *modified from* s *to* s', $y \in X$.
(b) *for all* $\ell.f$ *modified from* h *to* h', *there exists* L *with* $\ell\,\eta\,L$ *such that* $L.f \in X$.

Disjointness. Recall that $L_1 \diamond L_2$ denotes that L_1 and L_2 are disjoint. We define disjointness in two stages. In the first stage, we lift $\diamond$ to a relation between an abstract address and a set of abstract addresses as follows: (a) $x \diamond X$ iff $x \notin X$; (b) $L.f \diamond X$ iff for all $L_1.f \in X$, we have $L \diamond L_1$.

Second, we define what it means for an assertion ϕ to be disjoint from a set of abstract addresses, X. This relation, written $\phi \diamond X$, holds provided $a \diamond X$ for all abstract addresses a occurring on "the left hand side" of assertions in ϕ, i.e.: (a) For all $x \rightsquigarrow LI$ occurring in ϕ, $x \diamond X$; (b) For all $L.f \rightsquigarrow LI$ occurring in ϕ, $L.f \diamond X$; (c) For all $x\ltimes$ occurring in ϕ, $x \diamond X$; (d) For all $L.f\ltimes$ occurring in ϕ, $L.f \diamond X$; and (e) For all θ occurring in ϕ: if x occurs in θ then $x \notin X$. As we shall see later (Sec. 5), $\phi \diamond X$ is exactly the form of the side condition of the frame rule.

An Invariance. The main result of this section is an invariance result, intuitively stating that an assertion which is valid *before* executing a command, also remains valid *after*, provided it is *disjoint* from any abstract address modified by the command.

To precisely state this result, we need the following notion of "extension" of η and β: Say that η' over h' *extends* η over h, if $dom(h) \subseteq dom(h')$ and for all $\ell \in dom(h)$, for all L: $\ell\,\eta\,L$ iff $\ell\,\eta'\,L$.

Let $dom(h) \subseteq dom(h')$ and $dom(h_1) \subseteq dom(h_1')$. Say that β' over $h' \& h_1'$ *extends* β over $h \& h_1$ if $\beta = \{(\ell, \ell_1) \in \beta' \mid (\ell \in dom(h)) \vee (\ell_1 \in dom(h_1))\}$. (Therefore, if $\ell\,\beta'\,\ell_1$ and $\ell \in dom(h)$ then $\ell_1 \in dom(h_1)$, and vice versa).

LEMMA 4.2 (**Invariance**). *Suppose* $\phi \diamond X$. *Further, suppose* $(s, h) \rightarrow (s', h') \models_{\eta'} X$, *and* $(s_1, h_1) \rightarrow (s_1', h_1') \models_{\eta_1'} X$, *where* η' *over* h' *extends* η *over* h, *and* η_1' *over* h_1' *extends* η_1 *over* h_1. *Also, let* β' *over* $h' \& h_1'$ *extend* β *over* $h \& h_1$. *Suppose* $(s, h) \& (s_1, h_1) \models_{\beta,\eta,\eta_1} \phi$. *Then*

$$(s', h') \& (s_1', h_1') \models_{\beta',\eta',\eta_1'} \phi.$$

4.1 Logical implication

The purpose of this section is to define a notion of implication of assertions; this permits the deduction of more independences than can be obtained by tracking data and control flow only.

DEFINITION 4.3 (**Logically implies**). *Say that* ϕ_0 *logically implies* ϕ, *written* $\phi_0 \blacktriangleright \phi$, *iff* $(s, h) \& (s_1, h_1) \models_{\beta,\eta,\eta_1} \phi_0$ *implies* $(s, h) \& (s_1, h_1) \models_{\beta,\eta,\eta_1} \phi$.

The above definition allows us to show that the following logical implications are valid.

- Let θ be the programmer assertion $x = c$. Then $\theta \blacktriangleright x\ltimes$.
- Let θ be the assertion $(x = y)$. Then $(\theta \wedge y\ltimes) \blacktriangleright x\ltimes$.
- Let θ be the assertion $x = k(y)$, with k an arithmetic function. Then $(\theta \wedge y\ltimes) \blacktriangleright x\ltimes$.

Several other such logical implications are possible. For applications, recall Sec. 2, and see Sec. 6.

We can define $\blacktriangleright$ on abstract addresses in a manner similar to Def. 4.3. Say that $X \blacktriangleright X'$ iff $(s, h) \rightarrow (s', h') \models_\eta X$ implies $(s, h) \rightarrow (s', h') \models_\eta X'$. Clearly, if $X \subseteq X'$ then $X \blacktriangleright X'$.

5. Statically Checking Assertions via a Logic

To statically check assertions we define, in Table 2, a Hoare-like logic whose judgements take the form

$$\Pi \vdash \{\phi_0\}\ S\ \{\phi\}\ [X].$$

In the judgement, X is a set of abstract addresses that overapproximates the abstract addresses modified by S, ϕ_0 are the assertions that hold before execution of S, and ϕ are the assertions that hold after execution of S. Π is a summary environment for methods, such that $\Pi(C, m)$ is a (set of) summaries of the form $\{\psi_0\}\ \_\ \{\psi\}\ [X']$, where the only program variables mentioned in ψ_0 are *self* and the formal parameter of m, where the only program variable mentioned in ψ is $result$, and where X' does not contain program variables. The reason for having a *set* of summaries is polyvariance: at different call sites of the same method, different pre-and postconditions may hold. We will often omit Π in rules other than the rule for method call. Each judgement in Table 2 is a small specification.

Before discussing the small specifications in more detail, we shall define, for a judgement $\{\phi_0\}\ S\ \{\phi\}\ [X]$, its intended *meaning*, of which our logic will be a sound (but necessarily not complete) approximation.

5.1 Semantics of Judgements

DEFINITION 5.1. *We say that* $\mu \models \{\phi_0\}\ S\ \{\phi\}\ [X]$ *iff the following holds for all* $s, h, s', h', s_1, h_1, s_1', h_1', \beta, \eta, \eta_1$. *Assume*

$(s, h)\,[\![S]\!]\mu]\,(s', h')$ *and* $(s_1, h_1)\,[\![S]\!]\mu]\,(s_1', h_1')$ *and*
$(s, h) \& (s_1, h_1) \models_{\beta,\eta,\eta_1} \phi_0$.

Then there exists η' *over* h' *extending* η, *there exists* η_1' *over* h_1' *extending* η_1, *and there exists* β' *over* $h' \& h_1'$ *extending* β *over* $h \& h_1$, *such that*

(1a) $(s, h) \rightarrow (s', h') \models_{\eta'} X$
(1b) $(s_1, h_1) \rightarrow (s_1', h_1') \models_{\eta_1'} X$
(2) $(s', h') \& (s_1', h_1') \models_{\beta',\eta',\eta_1'} \phi$

Conditions **(1a)** and **(1b)** say that X is a sound overapproximation of the abstract addresses modified in S when its execution changes the state from (s, h) to (s', h'), or from (s_1, h_1) to (s_1', h_1'). Condition **(2)** says, under the assumption that precondition ϕ_0 holds for the initial pair of states (s, h) and (s_1, h_1), that the postcondition ϕ holds for the modified states (s', h') and (s_1', h_1'). Note that these conditions hold vacuously in case of non-termination, or run-time error (since then, states (s', h') and (s_1', h_1') would not exist).

Conjunction Rule not Sound in Semantic Model. It may be the case that $\mu \models \{\phi_0\}\ S\ \{\phi_1\}\ [X]$ and $\mu \models \{\phi_0\}\ S\ \{\phi_2\}\ [X]$ hold separately, but $\mu \models \{\phi_0\}\ S\ \{\phi_1 \wedge \phi_2\}\ [X]$ does not hold. For a concrete example, consider the following program S:

$$\textbf{if}\ z\ \textbf{then}\ x := \textbf{new}\ C;\ y := x\ \textbf{else}\ x := \textbf{new}\ C;\ y := \textbf{new}\ C$$

Using Def. 5.1, we can semantically establish $x\ltimes$ and $y\ltimes$ separately, but *not* $x\ltimes \wedge y\ltimes$. To see this, consider the initial states (s, h) and (s_1, h_1), evolving into states (s', h') and (s_1', h_1'). Our goal is to find β' extending β such that $(s'\ x)\beta(s_1'\ x)$ and $(s'\ y)\beta(s_1'\ y)$; this is trivial if $s(z)$ and $s_1(z)$ assume the same truth value, so assume that $s(z) \in \mathsf{True}$ but $s_1(z) \in \mathsf{False}$. Then there exists fresh location ℓ such that $s'(x) = s'(y) = \ell$, and there exists fresh locations $\ell_x \neq \ell_y$ such that $s_1'(x) = \ell_x$ and $s_1'(y) = \ell_y$. To establish $x\ltimes$, we define β' such that $\ell\ \beta'\ \ell_x$; similarly, to establish $y\ltimes$, we define β' such that $\ell\ \beta'\ \ell_y$. But to establish *both* $x\ltimes$ and $y\ltimes$, we would need $\ell\ \beta'\ \ell_x$ *and* $\ell\ \beta'\ \ell_y$, which conflicts with β' being a bijection.

5.2 Syntax-directed Rules

Table 2 gives the details of some small specifications. First note that ordinary assignment, $x := E$, is split into three cases – pure assignment, where E is an arithmetic expression; pointer assignment, where E is a variable z denoting a location; and null assignment, where E is $\mathbf{null}$.

Next note that for a given small specification, its region assertions are always relevant, in that those occurring in the precondition must be established by the context, whereas the independence assertions may or may not be relevant, depending on whether those occurring in the precondition are established by the context. Therefore certain specifications should be read as *two* specifications (for space reasons, we do not show both), with the "optional" independence assertions being listed right of a semicolon. For example, [PointerAssign] should be read as the two rules: $\{z \leadsto \rho\}\ x := z\ \{x \leadsto \rho\}\ [\{x\}]$ and $\{z \leadsto \rho,\ z\ltimes\}\ x := z\ \{x \leadsto \rho,\ x\ltimes\}\ [\{x\}]$.

Many of the rules in Table 2 have already been motivated by means of examples in Sec. 2, so below we shall discuss only a few, and also give the rule for method calls.

The postcondition of [New] asserts that x will be at some abstract address L with $L \neq \bot$; furthermore, $x\ltimes$ always holds and x is modified. The rule mirrors the concrete semantics of $\mathbf{new}$, where a fresh location is allocated in the heap, except that we do not require freshness of L.

Next we discuss [If], which is similar to the rule for conditionals in Hoare logic, except that in the presence of independences, some side conditions may be needed. Two cases:

(a) If ϕ_0 logically implies $x\ltimes$, then we know that in states (s, h) and (s_1, h_1), both $s(x)$ and $s_1(x)$ will have the *same* (integer) value, so the *same branch* of the conditional will be taken during evaluation. Hence, there is no indirect control flow, and thus no need for any side conditions. (In the context of security, this case amounts to the guard of the conditional being "low").

(b) Alternatively, in states (s, h) and (s_1, h_1), $s(x)$ and $s_1(x)$ may differ, causing different branches of the conditional to be taken. In this case, in order to assert $w\ltimes$ at the end of the conditional, it does not suffice to assert $w\ltimes$ at the end of each branch, since this merely says that two runs choosing the *same* branch will agree on the value of w. What we need is that:

1. w is not modified in any branch. (In the context of security, this amounts to "no write down" under a "high guard" [8]).
2. the two runs agree on the value of w *before* the conditional.

The first demand can be encoded as $\mathcal{I}(\phi) \diamond X$; the second, as $\phi_0 \blacktriangleright \mathcal{I}(\phi)$. Here, the notation $\mathcal{I}(\phi)$ denotes ϕ's projection to its independence assertions.

Concerning the specification of a method call, $x := y.m(z)$, assume that $type\ y = C$ and that $\Pi(C, m)$ contains the summary $\{\psi_0\}\ \_\ \{\psi\}\ [X]$. Then, with $\phi_0 = \psi_0[y/self, z/pars(m, C)]$ and $\phi = \psi[x/result]$,[9] we have

$$\text{[MethodCall]} \quad \Pi \vdash \{\phi_0\}\ x := y.m(z)\ \{\phi\}\ [X \cup \{x\}]$$

5.3 Structural Rules

There are two structural rules: [Conseq], which extends the rule of consequence in Hoare logic, and the frame rule, [Frame].

$$\text{[Conseq]} \quad \frac{\{\phi_1\}\ S\ \{\phi_2\}\ [X]}{\{\phi_1'\}\ S\ \{\phi_2'\}\ [X']} \quad \begin{array}{l} \text{if } \ \phi_1' \blacktriangleright \phi_1 \\ \text{and } \phi_2 \blacktriangleright \phi_2' \\ \text{and } X \blacktriangleright X' \end{array}$$

[9] The notation, e.g., $\psi[x/result]$ denotes substitution of x for $result$ in ψ.

$$\text{[Frame]} \quad \frac{\{\phi_1\}\ S\ \{\phi_2\}\ [X]}{\{\phi_1 \wedge \phi\}\ S\ \{\phi_2 \wedge \phi\}\ [X]} \text{ if } \phi \diamond X.$$

The frame rule is used to reason with small specifications in a larger context. For example, for a command $S_1 ; S_2$, rule [Seq] requires the postcondition of S_1 to be the same as the precondition of S_2. As the examples in Sec. 2 depict, such a match may not always be achievable by small specifications themselves: extra assertions must be added by invoking [Frame]. This is sound provided the added assertions are disjoint from the modified abstract addresses.

As suggested by the semantic considerations in Sec. 5.1, we do *not* have a rule of conjunction like the one in Hoare logic (without heaps), i.e., we cannot derive $\{\phi_0 \wedge \phi_0'\}\ S\ \{\phi \wedge \phi'\}\ [X \cup X']$ from $\{\phi_0\}\ S\ \{\phi\}\ [X]$ and $\{\phi_0'\}\ S\ \{\phi'\}\ [X']$. To see why this would be unsound (at least in our semantic model), let S be the command $x := \mathbf{new}\ C$. Then, for all L_1 and L_2, we would have $\{true\}\ S\ \{x \leadsto L_1\}\ [\{x\}]$ and $\{true\}\ S\ \{x \leadsto L_2\}\ [\{x\}]$ and by the proposed conjunction rule therefore $\{true\}\ S\ \{x \leadsto L_1 \wedge x \leadsto L_2\}\ [\{x\}]$. But this is clearly a semantic impossibility if $L_1 \diamond L_2$.

Remarks. **(a)** One may think that the small specifications lose information and may not be precise. For example, in [PointerAssign], why did $z\ltimes$ disappear in the postcondition? But that independence can be recovered by [Frame], since z is not modified. **(b)** Similarly, the rule for field update does not lose precision: if $y \leadsto L_1$ holds before then it also holds after, despite the use of [Conseq] to "unify" L_1 with the region of f (details are in [1]). **(c)** The technical report also shows one way to handle *strong update* in the rule [FieldUpd]. An example of strong update appeared in Sec. 2 but we do not discuss this issue any further due to lack of space. **(d)** In [If] and [Conseq], $\blacktriangleright$ has a semantic definition. However, when all assertions are restricted to region and independence assertions, $\blacktriangleright$ has a syntactic characterization given by Definition 7.4 and asserted by Theorem 7.6.

5.4 Soundness

DEFINITION 5.2 (**Consistent summary environment**). *Say that summary environment* Π *is consistent wrt. class table* CT *if whenever* $\Pi(C, m)$ *contains the summary* $\{\psi_0\}\ \_\ \{\psi\}\ [X]$*, and* S *is the body of a declaration of* m *in* C*, then* $\Pi \vdash \{\psi_0\}\ S\ \{\psi\}\ [X']$ *where* $X = \{L.f \mid L.f \in X'\}$.

The idea is that even if a local variable is modified by S and hence occurs in X', it should not occur in X since it is not visible outside m. On the other hand, all field updates[10] are globally visible.

THEOREM 5.3 (**Soundness**). *Let* Π *be a summary environment consistent wrt. class table* CT*. For a method* m *with body* S*, suppose* $\Pi \vdash \{\phi_0\}\ S\ \{\phi\}\ [X]$*. Then* $[\![CT]\!] \models \{\phi_0\}\ S\ \{\phi\}\ [X]$.

6. A Larger Example

We consider the following example due to Barnett et al. [7].

```
class C{
  private Hashtable ht := new Hashtable;  //cache
  public U m(T x){
    Hashtable t := self.ht;
    bool present := t.contains(x);
    if (!present){
      U y := costly(x);
      t.put(x, y); }
    U res := (U)t.get(x);
    assert (res = costly(x));
    result := res; } }
```

[10] Since we are handling only public fields. In future work, we hope to explore issues involving information hiding through private fields.

$$\text{[Assert]}\quad \{true\}\ \mathbf{assert}\ \theta\ \{\theta\}\ [\emptyset]$$

$$\text{[PureAssign]}\quad \frac{\{z_1,\ldots,z_n\} = \mathrm{free}(E)}{\{true;\ z_1\ltimes,\ldots,z_n\ltimes\}\ x := E\ \{x \leadsto \mathbf{int};\ x\ltimes\}\ [\{x\}]}$$

$$\text{[NullAssign]}\quad \{true\}\ x := \mathbf{null}\ \{x \leadsto \bot,\ x\ltimes\}\ [\{x\}]$$

$$\text{[PointerAssign]}\quad \{z \leadsto \rho;\ z\ltimes\}\ x := z\ \{x \leadsto \rho;\ x\ltimes\}\ [\{x\}]$$

$$\text{[FieldAcc]}\quad \begin{array}{l} \{y \leadsto \rho, \rho.f \leadsto \varrho;\ y\ltimes, \rho.f\ltimes\} \\ \quad x := y.f \\ \{x \leadsto \varrho;\ x\ltimes\} \\ [\{x\}] \end{array}$$

$$\text{[FieldUpd]}\quad \begin{array}{l} \{x \leadsto \rho, y \leadsto \varrho, \rho.f \leadsto \varrho;\ x\ltimes, y\ltimes, \rho.f\ltimes\} \\ \quad x.f := y \\ \{\rho.f \leadsto \varrho;\ \rho.f\ltimes\} \\ [\{\rho.f\}] \end{array}$$

$$\text{[New]}\quad \{true\}\quad x := \mathbf{new}\ C\quad \{x \leadsto \rho,\ x\ltimes\}\ [\{x\}]\quad \text{where } \rho \neq \bot$$

$$\text{[Seq]}\quad \frac{\{\phi_0\}\ S_1\ \{\phi_1\}\ [X_1] \qquad \{\phi_1\}\ S_2\ \{\phi\}\ [X_2]}{\{\phi_0\}\ S_1\ ;S_2\ \{\phi\}\ [X_1 \cup X_2]}$$

$$\text{[If]}\quad \frac{\{\phi_0\}\ S_1\ \{\phi\}\ [X] \qquad \{\phi_0\}\ S_2\ \{\phi\}\ [X]}{\{\phi_0\}\ \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{\phi\}\ [X]} \quad \begin{array}{ll} \text{where} & \phi_0 \blacktriangleright x\ltimes \\ \text{or} & \mathcal{I}(\phi) \diamond X \text{ and} \\ & \phi_0 \blacktriangleright \mathcal{I}(\phi) \end{array}$$

Table 2. Small specifications: excerpts. A ρ (a ϱ) is a metavariable to be instantiated by an L (an LI).

The method m is an efficient implementation of the method $costly$, employing memoization: argument-result pairs are cached in a hash table t, with the argument as key. A call to m with some argument, x, first checks if a value exists for key x in t (lines 4, 5); if not, it is computed (line 6) and stored in t (line 7). At that point, we know that the result can be retrieved from the hash table (line 8) and returned (line 10).

We shall now argue that m is observationally pure (and hence can be used in specifications). As in Sec. 2 (for $cexp$), this involves showing (i) that $result$ depends on x only; (ii) that m modifies only locations not visible to the caller.

For (i), we must show that two runs which agree on the initial value of x also agree on the final value of $result$. So let $x\ltimes$ be in the precondition, then – due to the frame rule, as x is not modified along the way – $x\ltimes$ holds after line (9), where by [Assert] we also have $res = costly(x)$. By [Conseq], this entails that $res\ltimes$ holds before line (10). By [PointerAssign], this entails that $result\ltimes$ is in the postcondition, as desired.

For (ii), assume that $Hashtable$ has two fields, key and val, and that it is in abstract location L_0. The only abstract addresses modified by m are $L_0.key$ and $L_0.val$ (as well as certain local variables which are not visible to the caller, c.f. Definition 5.2). The desired invisibility can then be obtained by assuming that L_0 is disjoint from all abstract locations used outside of m.

For the above to work out formally, we need method summaries such as the ones below:

$$\begin{array}{c} \{self \leadsto \rho_0, \\ \rho_0.key \leadsto \rho_1, x \leadsto \rho_1, \\ \rho_0.val \leadsto \rho_2, y \leadsto \rho_2\} \\ put \\ \{\} \\ [\rho_0.key, \rho_0.val] \end{array} \qquad \begin{array}{c} \{self \leadsto \rho_0, \\ \rho_0.key \leadsto \rho_1, x \leadsto \rho_1, \\ \rho_0.val \leadsto \rho_2\} \\ get \\ \{result \leadsto \rho_2\} \\ [\emptyset] \end{array}$$

Note that we do *not* need the summaries to contain independence assertions. It is interesting, however, to consider how such assertions could be added in the summary for, e.g., the method get. Naïvely, we would expect (c.f. the method $getQ$ described in Sec. 2) that if the precondition ψ_0 contains the assertions $x\ltimes$, $self\ltimes$, $\rho_0.key\ltimes$, and $\rho_0.val\ltimes$, then the postcondition ψ_1 will contain the assertion $result\ltimes$. But in general, get cannot be implemented so as to satisfy this summary. To see this we assume, in order to arrive at a contradiction, that S_g is the body of such an implementation. By Theorem 5.3 (and Definition 5.2), we have $\mu \models \{\psi_0\}\ S_g\ \{\psi_1\}\ \_$. Now consider two states, (s,h) and (s_1,h_1), where the key $s(x)$ (which due to $x\ltimes$ equals $s_1(x)$) is mapped by the hash table to *different* integer values. With β chosen such that β relates $s(self)$ to $s_1(self)$ but relates no other locations, we have $(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \psi_0$ (since, e.g., $\rho_0.key\ltimes$ vacuously holds). But with S_g transforming (s,h) into (s',h') and (s_1,h_1) into (s'_1,h'_1), it is *not* possible (since the integers $s'(result)$ and $s'_1(result)$ are different) to define β' such that $(s',h')\,\&\,(s'_1,h'_1) \models_{\beta',\eta',\eta'_1} result\ltimes$. This yields the desired contradiction.

To fix the above situation, we need to be more concrete about how the (hash) table is implemented. Suppose that it is a linked list, with each record containing not only a key and a val field (both integers), but also a $next$ field. Then, we can implement get such that $result\ltimes$ is in the postcondition, provided we include $\rho_0.next\ltimes$ in the precondition ψ_0. To see this, consider as above two states, (s,h) and (s_1,h_1), with $(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \psi_0$. Since ψ_0 contains $x\ltimes$, there exists an integer k such that $s(x) = s_1(x) = k$. Wlog., we can assume that in the first state, k occurs as the third key in the list. That is, there exists locations ℓ, ℓ_1, and ℓ_2 such that $s(self) = \ell$, $h\,\ell\,next = \ell_1$, $h\,\ell_1\,next = \ell_2$, and $h\,\ell_2\,key = k$. Since ψ_0 contains $self\ltimes$, with $s_1(self) = \ell'$ we have $\ell\,\beta\,\ell'$. This entails, since ψ_0 contains $\rho_0.next\ltimes$ and we can assume $\ell\,\eta\,\rho_0$, that with $h_1\,\ell'\,next = \ell'_1$ we have $\ell_1\,\beta\,\ell'_1$; similarly we then infer that $\ell_2\,\beta\,\ell'_2$ with $h_1\,\ell'_1\,next = \ell'_2$. Since ψ_0 contains $\rho_0.key\ltimes$ and $\rho_0.val\ltimes$, we now infer that $h_1\,\ell'_2\,key = h\,\ell_2\,key = k$, and that there exists v such that $h_1\,\ell'_2\,val = h\,\ell_2\,val = v$. With (s',h') and (s'_1,h'_1) the final states, this shows the desired $s'_1(result) = v = s'(result)$.

7. Computing Postconditions

It is time to address how to decide, and implement, our logic. For that purpose, we shall along the way introduce several simplifying assumptions, two of which we state already now.

ASSUMPTION 7.1. *Abstract locations form a finite complete lattice, with $\bot$ the least element and $\top$ the greatest element, where $\sqcup$ "corresponds to" set union and $\sqcap$ "corresponds to" set intersection. That is, we require that*

- *if $L = L_1 \sqcup L_2$ then $\ell\,\eta\,L$ iff $\ell\,\eta\,L_1$ or $\ell\,\eta\,L_2$*
- *if $L = L_1 \sqcap L_2$ then $\ell\,\eta\,L$ iff $\ell\,\eta\,L_1$ and $\ell\,\eta\,L_2$.*

Accordingly, we also require that if $L = L_1 \sqcup L_2$ then for all L': $L' \diamond L$ iff $L' \diamond L_1$ and $L' \diamond L_2$.

Recall from Sec. 4 that $\bot$ approximates nil but no concrete heap locations; on the other hand, $\top$ approximates all concrete locations.

The next assumption is motivated by the fact that if $L = L_1 \sqcup L_2$, then any information about $L.f$ can be deduced from information about $L_1.f$ and $L_2.f$.

ASSUMPTION 7.2. *Among the abstract locations are some "irreducible" elements (we write* $\mathsf{irr}(L)$ *for irreducible* L*) such that*

- *if* $L_1 \neq L_2$ *are irreducible then* $L_1 \diamond L_2$*;*
- *for each abstract location* L*, there are unique irreducible elements* $L_1, \ldots, L_n$ *(*$n \geq 0$*) such that* $L = L_1 \sqcup \ldots \sqcup L_n$.

Recall from Sec. 4 that all disjunctions in assertions occur only within programmer assertions θ. Thus, we can view an assertion ϕ as a set (implicitly a conjunction) of primitive assertions α. It is convenient to work with assertions where all abstract locations (on the "left hand side") are irreducible and occur at most once:

DEFINITION 7.3. *Say that* ϕ *is normalized iff (a) if* $L.f \rightsquigarrow L' \in \phi$ *then* L *is irreducible; (b) if* $L.f\ltimes \in \phi$ *then* L *is irreducible; (c) if* $L.f \rightsquigarrow L_1 \in \phi$ *and* $L.f \rightsquigarrow L_2 \in \phi$ *then* $L_1 = L_2$*; (d) if* $x \rightsquigarrow L_1 \in \phi$ *and* $x \rightsquigarrow L_2 \in \phi$ *then* $L_1 = L_2$*; (e)* ϕ *does not contain any assertions of the form* $\_ \rightsquigarrow \top$ *or* $\_ \rightsquigarrow \mathbf{int}$*; (f)* ϕ *contains exactly one programmer assertion.*

For a normalized assertion ϕ, the region part gives rise to a function as follows: (a) $\phi(x) = \mathbf{int}$, if *type* $x = \mathbf{int}$; (b) $\phi(x) = L$, if $x \rightsquigarrow L \in \phi$; (c) $\phi(x) = \top$, otherwise. And, given $\mathsf{irr}(L_0)$, we define: (d) $\phi(L_0.f) = \mathbf{int}$, if *type* $f = \mathbf{int}$; (e) $\phi(L_0.f) = L$, if $L_0.f \rightsquigarrow L \in \phi$; (f) $\phi(L_0.f) = \top$, otherwise.

It is possible to write a function $norm$ that converts an assertion ϕ into a normalized assertion $norm(\phi)$ which is logically equivalent. (That is, $\phi \blacktriangleright norm(\phi)$ and $norm(\phi) \blacktriangleright \phi$; also we have $\phi \diamond X$ iff $norm(\phi) \diamond X$). For example, if L can be written as $L_1 \sqcup \ldots \sqcup L_n$ where $\mathsf{irr}(L_1) \ldots \mathsf{irr}(L_n)$, then $norm$ will transform an assertion $L.f \rightsquigarrow LI$ into $\{L_1.f \rightsquigarrow LI, \ldots, L_n.f \rightsquigarrow LI\}$.

7.1 Checking Logical Implication

In Sec. 4.1, we gave a semantic definition (4.3) of logical implication. We shall show that *without* programmer assertions, that definition is equivalent to a syntactic characterization which is readily implementable.

DEFINITION 7.4. *For normalized* ψ *and* ψ'*, we write* $\psi \preceq \psi'$ *iff the following holds:*

(a) *if* $x \rightsquigarrow L' \in \psi'$ *there exists* L *with* $L \preceq L'$ *such that* $x \rightsquigarrow L \in \psi$
(b) *it* $L_1.f \rightsquigarrow L' \in \psi'$ *there exists* L *with* $L \preceq L'$ *such that* $L_1.f \rightsquigarrow L \in \psi$
(c) $x\ltimes \in \psi'$ *implies* $x\ltimes \in \psi$
(d) $L.f\ltimes \in \psi'$ *implies* $L.f\ltimes \in \psi$*;*
(e) $\theta' \in \psi'$ *implies that there exists* $\theta \in \psi$ *such that* $\theta \blacktriangleright \theta'$.

For arbitrary ϕ *and* ϕ'*, we shall – with abuse of notation – write* $\phi \preceq \phi'$ *iff* $norm(\phi) \preceq norm(\phi')$.

Now consider the case with no programmer assertions. Then clause (e) above is trivial (as $\theta' = \theta = true$), so it is easy to decide $\preceq$. As shown by the results below, this amounts to deciding $\blacktriangleright$.

FACT 7.5. *If* $\phi \preceq \phi'$ *then* $\phi \blacktriangleright \phi'$.

THEOREM 7.6. *If* ϕ *and* ϕ' *contains no programmer assertions, then* $\phi \blacktriangleright \phi'$ *is equivalent to* $\phi \preceq \phi'$.

To see why we need to assume the absence of programmer assertions, observe that $x = c$ logically implies $x\ltimes$ whereas $(x = c) \preceq x\ltimes$ does not hold. For that assumption to be removed, we would need a much stronger version of $norm$ that finds all instances of logical implication hidden in programmer assertions.

Concerning how to decide $X \blacktriangleright X'$, we proceed in a similar (but much simpler) way: we say that a set X of abstract addresses is normalized if L is irreducible for all $L.f \in X$; we write a function $norm$ that converts a set of abstract addresses into an equivalent normalized set; finally, we establish

FACT 7.7. $X \blacktriangleright X'$ *iff* $norm(X) \subseteq norm(X')$.

7.2 A Sound Algorithm

We shall define, inductively on S, a function $sp(S, \phi_0)$ that given a command S and a precondition ϕ_0 (which could be "global") computes a pair (ϕ, X); here we want ϕ to be a postcondition of S, and X to be the abstract addresses that may be modified by S. With

ASSUMPTION 7.8. *We assume that a consistent summary environment* Π *is given in advance*

we can show soundness of *sp* wrt. to the logic:

THEOREM 7.9. *If* $sp(S, \phi_0) = (\phi, X)$ *then* $\Pi \vdash \{\phi_0\}\ S\ \{\phi\}\ [X]$.

The full definition of *sp* is[11] in [1]; below we shall list the most interesting cases which are for conditional, assignment, and method call. (For *programmer assertions*, $sp(\mathbf{assert}\ \theta, \phi_0) = (\theta \wedge \phi_0, \emptyset)$).

Conditionals. We call *sp* recursively on the two branches and then combine, via a least upper bound operator, the resulting assertions.

DEFINITION 7.10. *For normalized* ψ_1 *and* ψ_2*, we define* $\psi = \psi_1 \sqcup \psi_2$ *(which is itself normalized) as follows:*

- $x \rightsquigarrow L \in \psi$ *iff there exists* L_1 *and* L_2 *with* $L = L_1 \sqcup L_2 \neq \top$ *such that* $x \rightsquigarrow L_1 \in \psi_1$ *and* $x \rightsquigarrow L_2 \in \psi_2$
- $L_0.f \rightsquigarrow L \in \psi$ *iff there exists* L_1 *and* L_2 *with* $L = L_1 \sqcup L_2 \neq \top$ *such that* $L_0.f \rightsquigarrow L_1 \in \psi_1$ *and* $L_0.f \rightsquigarrow L_2 \in \psi_2$
- $x\ltimes \in \psi$ *iff* $x\ltimes \in \psi_1$ *and* $x\ltimes \in \psi_2$
- $L.f\ltimes \in \psi$ *iff* $L.f\ltimes \in \psi_1$ *and* $L.f\ltimes \in \psi_2$
- $\theta \in \psi$ *iff there exists* $\theta_1 \in \psi_1$*,* $\theta_2 \in \psi_2$ *such that* $\theta = \theta_1 \vee \theta_2$.

For arbitrary ϕ_1 *and* ϕ_2*, we shall – with abuse of notation – write* $\phi_1 \sqcup \phi_2$ *for* $norm(\phi_1) \sqcup norm(\phi_2)$.

Let ϕ_{12} be the least upper bound of the analyses of the branches. Looking at the side conditions for [If] in the logic, we see that if ϕ_0 logically implies $x\ltimes$ (with x the test), we can just return ϕ_{12}. Otherwise, in order to satisfy the second side condition, we must remove from ϕ_{12} all independences which either are not in the precondition, or whose abstract addresses have been modified in S_1 or in S_2. The resulting code is

```
sp(if x then S1 else S2, φ0) =
    let (φ1, X1) = sp(S1, φ0) in
    let (φ2, X2) = sp(S2, φ0) in
    let X = norm(X1 ∪ X2) in
    let φ12 = φ1 ⊔ φ2 in
    let φ =if φ0 ⪯ x⋉
            then φ12
            else φ12 \ (C1 ∪ C2)
    where C1 = {y⋉ | (y ∈ X) ∨ (y⋉ ∉ norm(φ0))}
    and C2 = {L.f⋉ | (L.f ∈ X) ∨ (L.f ∉ norm(φ0))}
  in (φ, X)
```

Assignments. Assume that S is an assignment A which is not a method call, i.e., A is either a pure assignment, a pointer assignment, a null assignment, a field access, a field update, or an

[11] Except that we do not yet handle $\mathbf{while}$ loops; such would require some kind of fixed point iteration.

object creation. Assume that we have a nondeterministic function $Choose(A, \phi_0)$ which returns a triple (ψ_0, ψ, X) such that $\{\psi_0\}\ A\ \{\psi\}\ [X]$ is an instance of a rule for A in the logic where $\phi_0 \preceq \psi_0$. Then define

$$
\begin{aligned}
&\mathit{sp}(S, \phi_0) = \\
&\quad \mathsf{let}\ (\psi_0, \psi, X) = Choose(A, \phi_0)\ \mathsf{in} \\
&\quad \mathsf{let}\ \phi = \psi \wedge disj(\phi_0, X)\ \mathsf{in}\ (\phi, X)
\end{aligned}
$$

Here, the function $disj$ extracts the parts of an assertion *not modified* by the assignment, thus incorporating the frame rule. It is defined by $disj(\phi, X) = \{\alpha \in norm(\phi) \mid \alpha \diamond X\}$.

So far, the above definition is very non-deterministic; it will be concretized in the next section when we consider *strongest* postconditions.

Method calls. Assume that S is a method call $x := y.m(w)$, with $type\, y = C$ where C contains a method m with formal parameter z. Assume that we have a non-deterministic function[12] $Choose(m, C, \phi_0)$ which returns a triple (ψ_0, ψ, X) such that $\{\psi_0\}\ \_\ \{\psi\}\ [X] \in \Pi(C, m)$ where $\phi_0 \preceq \psi_0[y/\mathit{self}, w/z]$. Then:

$$
\begin{aligned}
&\mathit{sp}(S, \phi_0) = \\
&\quad \mathsf{let}\ (\psi_0, \psi, X) = Choose(m, C, \phi_0)\ \mathsf{in} \\
&\quad \mathsf{let}\ \phi_X = disj(\phi_0, X \cup \{x\})\ \mathsf{in} \\
&\quad \mathsf{let}\ \phi = \psi[x/result] \cup \phi_X\ \mathsf{in}\ (\phi, X \cup \{x\})
\end{aligned}
$$

Construction of method summaries. In an actual implementation, the summary environment Π may be built incrementally, by using *sp* to analyze a new method in the context of the current Π (see, e.g., [25]). For recursive methods, however, the user might be required to provide the summaries, as in ESC/Java [16].

7.3 Strongest Postcondition

We shall now look at conditions for when *sp*, as defined in the previous section, is indeed the *strongest* postcondition. We want to prove the following completeness theorem

THEOREM 7.11 (**Completeness**). *If* $\mathit{sp}(S, \phi_0) = (\phi, X)$ *and* $\{\phi_0\}\ S\ \{\phi'\}\ [X']$, *then* $\phi \blacktriangleright \phi'$ *and* $X \blacktriangleright X'$.

For that purpose, we need to control the nondeterminism in the selection of abstract locations in rule [New].

ASSUMPTION 7.12. *Each occurrence of "new" is associated with a specific irreducible abstract location* L_0 *such that the only rule applicable for that occurrence is*

$$\{true\}\ x := \mathbf{new}\ C\ \{x \rightsquigarrow L_0;\ x\ltimes\}\ [\{x\}].$$

Then we can concretize, as done in Table 3, the function $Choose$ for assignments. Thanks to Assumption 7.12, we can show that $Choose$ computes the "strongest applicable version".

DEFINITION 7.13 (**Strongest Applicable Version**). *Given rule schema (*$j \in J$*),* $\{\psi_j\}\ S\ \{\psi'_j\}\ [X_j]$. *For given* ϕ_0*, we say that* j_0 *is the strongest applicable version if*

- $\phi_0 \preceq \psi_{j_0}$
- *For all* j *such that* $\phi_0 \preceq \psi_j$*, it holds that* $\psi'_{j_0} \preceq \psi'_j$ *and* $X_{j_0} \blacktriangleright X_j$.

Under the further assumption that the method summaries have been constructed such that there exists a strongest applicable version for method calls, we can prove the completeness (Theorem 7.11) of *sp*, provided that $\preceq$ is equivalent to $\blacktriangleright$ (as is the case without programmer assertions, c.f. Theorem 7.6).

[12] Required because we have a set of summaries for different calling contexts, so we need to select the appropriate one.

8. Discussion

We have specified, via a Hoare-style logic, an interprocedural and flow-sensitive information flow analysis for object-oriented programs. (The analysis is *in*sensitive to termination, but we expect that adding assertions of the form $\bot\ltimes$, c.f. [3], would make it sensitive to termination). Because aliasing can compromise confidentiality, the logic uses region assertions to describe aliasing that may arise between variables and between heap values. Independence assertions describe the absence of leaks due to data and control flow in a program. Together with the knowledge that particular abstract addresses are disjoint, i.e., they must not alias, the logic can be employed to specify a more precise information flow analysis than extant type-based approaches. We also permit JML style programmer assertions in code. Such assertions allow more programs to be deemed secure than would be permitted by region and independence assertions alone, albeit at the cost of a fully automatic checker. The technical report considers dynamic dispatch (which we avoid in this paper); the proof rule for method call needs to be augmented with side conditions as in [If].

Local reasoning about state is supported in our logic and we show a number of examples. While ordinary Hoare logic without aliasing is compositional by nature, aliasing makes it challenging to reason locally about the heap. By drawing upon fundamental ideas from separation logic, we achieve local reasoning: we use small specifications for each command and combine specifications via a frame rule. The small specifications only mention abstract addresses relevant to a command and semantically correspond to the footprint of the command in the global state [21]. The frame rule permits a move from local to non-local specifications.

As we mentioned in Sec. 5, Table 2 specifies two sets of rules. The reader might have noted that the rules that mention *region assertions only* specify a points-to analysis similar to well-known ones, e.g., [13, 9]. Data flow facts used in typical points-to analyses can be viewed as assertions. Nevertheless, we have not found in the literature an explicit Hoare-style specification of interprocedural points-to analysis that is based on local reasoning via small specifications and the frame rule. On top of such a points-to analysis, a host of other analyses (rather than just information flow analysis) could be specified.

There is much work that remains. We wish to experimentally validate whether local reasoning with the frame rule indeed provides scalability. Towards this goal, we plan to extend ESC/Java2[13] and its assertion language, JML [12], to handle region and independence assertions. This would provide a verification framework for information flow properties. For checking benchmarks (e.g.,[5]) that use declassification, we conjecture that independence assertions might help in statically predicting program points where declassification may be used.

A significantly harder problem is obtaining a modular interprocedural *analysis*. This requires devising a modular algorithm for computing strongest postconditions, one that discovers and updates procedure summaries on the fly. We plan to explore how local reasoning might be employed in this process.

Although our logic does not have separation logic's spatial conjunction ($\star$) operator, we conjecture that the semantics of assertions could be alternatively given as follows: the meaning of e.g., $x \rightsquigarrow L$ in state (s, h) under η, could consider a partition of h into disjoint subheaps h_1, h_2 such that $dom(h_1) = \{s(x)\}$ with $(s(x))\, \eta\, L$.

Our hope is that local reasoning will be used in the specification of program analyses and — in the security context — used as a foundation for checking security policies for practical systems composed of components.

[13] `http://secure.ucd.ie/products/opensource/ESCJava2`

$Choose(x := \mathbf{new}\ C, \phi_0) =$
  let L_0 be the designated abstract location
    for this occurrence of "new" in
  $(\{\}, \{x \rightsquigarrow L_0, x\ltimes\}, \{x\})$

$Choose(x := z, \phi_0) =$
  let $L = \phi_0(z)$ in
  $\mathbf{if}\ \phi_0 \preceq z\ltimes$
  $\mathbf{then}\ (\{z \rightsquigarrow L, z\ltimes\}, \{x \rightsquigarrow L, x\ltimes\}, \{x\})$
  $\mathbf{else}\ (\{z \rightsquigarrow L\}, \{x \rightsquigarrow L\}, \{x\})$

$Choose(x := y.f, \phi_0) =$
  let $L = \phi_0(y) = L_1 \sqcup ... \sqcup L_k$ in
  let $LI = \sqcup_{j \in 1...k} \phi_0(L_j.f)$ in
  $\mathbf{if}\ \phi_0 \preceq y\ltimes, L.f\ltimes$
  $\mathbf{then}\ (\{y \rightsquigarrow L, L.f \rightsquigarrow LI, y\ltimes, L.f\ltimes\}, \{x \rightsquigarrow LI, x\ltimes\}, \{x\})$
  $\mathbf{else}\ (\{y \rightsquigarrow L, L.f \rightsquigarrow LI\}, \{x \rightsquigarrow LI\}, \{x\})$

$Choose(x := E, \phi_0) =$
  let $z_1, \ldots, z_n = \mathrm{free}(E)$ in
  $\mathbf{if}\ \phi_0 \preceq z_1\ltimes, \ldots, z_n\ltimes$
  $\mathbf{then}\ (\{z_1\ltimes, \ldots, z_n\ltimes\}, \{x\ltimes\}, \{x\})$
  $\mathbf{else}\ (\{\}, \{\}, \{x\})$

$Choose(x := \mathbf{null}, \phi_0) = (\{\}, \{x \rightsquigarrow \bot, x\ltimes\}, \{x\})$

$Choose(x.f := y, \phi_0) =$
  let $L = \phi_0(x) = L_1 \sqcup ... \sqcup L_k$ in
  let $LI' = \phi_0(y)$ in
  let $LI = \sqcup_{j \in \{1...k\}} \phi_0(L_j.f) \sqcup LI'$ in
  $\mathbf{if}\ \phi_0 \preceq x\ltimes, y\ltimes, L.f\ltimes$
  $\mathbf{then}\ (\{x \rightsquigarrow L, y \rightsquigarrow LI, L.f \rightsquigarrow LI, x\ltimes, y\ltimes, L.f\ltimes\},$
    $\{L.f \rightsquigarrow LI, L.f\ltimes\}, \{L.f\})$
  $\mathbf{else}\ (\{x \rightsquigarrow L, y \rightsquigarrow LI, L.f \rightsquigarrow LI\}, \{L.f \rightsquigarrow LI\}, \{L.f\})$

Table 3. The function $Choose$, given normalized ϕ_0.

Acknowledgements

To Alex Aiken, Dave Naumann, Peter O'Hearn, John Reynolds, Tamara Rezk, Andrei Sabelfeld, Dave Sands, Dave Schmidt, Lyn Turbak and the POPL reviewers for discussions, comments and encouragement. We were supported in part by NSF grants CCR-0209205, ITR-0326577, and CCR-0296182.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow analysis of pointer programs. Technical Report CIS TR 2005-1, Kansas State University, July 2005.

[2] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In SAS, LNCS 3148, pages 100–115. Springer-Verlag, 2004.

[3] T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*, special issue of SAS 2004. To appear.

[4] A. Askarov. Secure Implementation of cryptographic protocols: A case study of mutual distrust. Master's dissertation, Chalmers University of Technology, April 2005.

[5] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In ESORICS, LNCS 3679, pages 197–221. Springer-Verlag, 2005.

[6] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *JFP* 15(2):131–177, Mar. 2005.

[7] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004.

[8] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.

[9] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In PLDI, pages 103–114, 2003.

[10] M. Bishop. Computer Security: Art and Science. Addison-Wesley, 2003.

[11] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering* 21(10):785–798, 1995.

[12] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.

[13] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures (with retrospective). In Best of PLDI, pages 343–359, 1990.

[14] E. S. Cohen. Information transmission in sequential programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

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

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In PLDI, pages 234–245, 2002.

[17] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.

[18] S. Hunt and D. Sands. On flow-sensitive security types. In POPL 2006. To appear.

[19] A. C. Myers. JFlow: Practical mostly-static information flow control. In POPL, pages 228–241, 1999.

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

[21] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In CSL, LNCS 2142, pages 1–19. Springer-Verlag, 2001.

[22] F. Pottier and V. Simonet. Information flow inference for ML. *TOPLAS* 25(1):117–158, Jan. 2003.

[23] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In LICS, pages 55–74. 2002.

[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[25] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In SAS, LNCS 3148, pages 84–99. Springer-Verlag, 2004.

[26] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.