Verifying a Secure Information Flow Analyzer

May 12, 2005

David A. Naumann*

Stevens Institute of Technology, Hoboken NJ 07030 USA

Abstract. Denotational semantics for a substantial fragment of Java is formalized by deep embedding in PVS, making extensive use of dependent types. A static analyzer for secure information flow for this language is proved correct, that is, it enforces noninterference.

1 Introduction

This paper reports on the use of the PVS theorem prover [13] to formalize a semantic model for a core fragment of Java and machine check the correctness of a secure information flow analyzer.

The primary objective of the project is to check the results of Banerjee and Naumann [2]. Their work specifies a static analysis, in the form of a type system, for secure information flow in a substantial fragment of Java including mutable state, recursive types, dynamic allocation, inheritance and dynamic binding, type casts, and the code-based access control mechanism (stack inspection [6]). Security policy, i.e., confidentiality/integrity, is expressed by annotating object fields and method signatures with levels from a security lattice. The main result of [2] is that the typing rules are sound in the sense that a program deemed safe by the analysis is noninterferent, w.r.t. the policy (as in [18], see the survey by Myers and Sabelfeld [15]). A detailed proof of soundness has undergone peer review but it is sufficiently complicated to merit machine checking.

In recent years, operational semantics has been popular but [2] is based on a denotational model in the style of Scott and Strachey. In particular, the interesting semantic domains are higher order dependent function spaces and the main results are proved by fixpoint induction. A second objective of our project is to explore how well PVS supports formalizing and reasoning about such a model. Owing to the use of a denotational model we reap the benefits of a deep embedding while avoiding the need to manipulate or even define syntactic substitution. Moreover there is no explicit method call stack, which simplifies reasoning about noninterference.

A third objective was for the author to gain his first experience with a proof assistant. To give this a more scholarly spin, let us say the objective was to assess the effectiveness of PVS as a proof assistant for a user without expertise in mechanical theorem proving.

The project is largely completed. With the omission of code-based access control, the results of [2] have been completely formalized and proved. The omission was made only to scale down this first phase of the project. The static analysis is presented as

* Supported in part by NSF CCR-0208984, CCF-0429894, and CCR-ITR-0326540; ONR N00014-01-1-0837; and the New Jersey Commission on Science and Technology.

a recursive function that implements the typing rules of [2], and an arbitrary security lattice is considered, whereas the paper considers only the two-point lattice.

The design decision with the most far-reaching consequences was to use higher order dependent types to encode the semantics in a way that closely matches the paper [2]. The PVS type system is undecidable; the type checker generates type correctness conditions (TCCs) that must be proved. Once proved they are useful as lemmas. The use of dependent predicate types to express, e.g., the absence of dangling pointers and ill-typed program states, means that type-correctness of our definitions requires nontrivial proof in some cases. A tangential consequence is type soundness for the modeled language.

PVS is notable for its high level of automation and integrated decision procedures but to our knowledge there are no previous large applications involving higher order dependent types like ours. A considerable amount of user interaction appears to be needed for the proofs, for reasons explained in the sequel. The proofs mostly follow the structure of those in the paper, as reflected in the formulation of lemmas. But rather than transcribing the details from the paper, I tried to carry out proofs by the seat of my pants and I found PVS to be a fairly pleasant assistant in this working style. As a beginning user I refrained from trying to do sophisticated control of rewriting, or defining new proof strategies, or using automatic conversions between total and partial functions represented by lifting. I decided not to factor large proofs by introducing lots of little lemmas with no independent interest, given that PVS offers a graphical display that facilitates inspection of sequents at intermediate points. Some proof scripts run to several dozen lines and a few are a hundred lines.

Axioms are used only to express assumptions as follows: (a) the program to be analyzed is syntactically well formed (e.g., ordinary type correctness, subclassing is acyclic); (b) flow policy is well formed, i.e., invariant with respect to subclassing; and (c) the memory allocator, otherwise arbitrary, returns fresh locations. The axioms are simple and soundness is obvious. PVS theory interpretations have been used to confirm this claim but that is omitted for lack of space.

Related work. Strecker [16] uses Isabelle/HOL to show soundness for a language and security typing rules very similar to those in [1], i.e., a sequential fragment of Java, without access control. Strecker's result, like that in [1], makes an assumption of "parametricity" for the memory allocator in order to use equality rather than an arbitrary bijection on locations in the definition of noninterference. This simplifies the proofs considerably but is at odds with memory management in practice. Strecker also confines attention to the two-element lattice. The present work confirms the wisdom of those simplifications. Although an arbitrary lattice is needed for practical applications, it results in an additional quantifier in each of the key properties (indistinguishability, write-confinement, and noninterference) and this pervades the proofs. Treating memory allocation realistically necessitates the use of a bijection on locations in the definition of indistinguishability. As Strecker remarks, this means that there are fewer opportunities to exploit provers' built-in equality reasoning.

Jacobs, Pieters and Warnier [8] report on using PVS to verify soundness of a static information flow analysis for a simple imperative language. The analysis is based on abstract interpretation instead of types, which allows a variable to be used for information of different levels at different times. Hähnle and Sands [5] use the KeY tool, an

interactive theorem prover, to prove confidentiality of programs in a subset of JavaCard. Confidentiality is formalized in dynamic logic, following the semantic approach of Joshi and Leino [9]. Rushby [14] uses EHDM to check an "unwinding theorem" that says the "no read up" and "no write down" rules [15] for individual steps suffice to ensure noninterference. This elegant theory is set in an abstract model rather than a programming language; the focus is on intransitive noninterference.

Overview. This paper reviews the work of [2] and describes the formalization in PVS of the semantics (Sect. 2), policy and static analysis (Sect. 3), and proof of noninterference (Sect. 4). Sect. 5 concludes. For readers not familiar with PVS, key notations are briefly explained. More details can be found in a technical report on the author's home page, together with the PVS files.

2 Formalizing of the language: syntax and semantics

This section describes the object language, as defined "in the paper" (meaning [2]), and its formalization in PVS. For clarity, some minor details of the description differ from the paper and the PVS code, but only in ways that seem unlikely to lead to confusion.

Signature of a class table. A well formed program is given in the form of a class table, i.e., a collection of class declarations, each giving a superclass, field types, method signatures, and method implementations. The paper follows [7] in using several auxiliary functions: $superC$ gives the direct superclass declared by a class named C; $dfieldsC$ gives the field declarations $f0 : T0, f1 : T1, \ldots$ of C; $fieldsC$ combines the declared and inherited field declarations; and $mtype(m,C)$ gives the parameter and return type for method m declared or inherited in C. If there is no such method, $mtype(m,C)$ is undefined. Fields and methods of a class may make mutually recursive reference to any other classes. We assume there are given, disjoint sets of field names, class names, and method names, ranged over by f, C, and m respectively. Finally, $mbody(m,C)$ gives the method body, if method m has a declaration in class C, and is undefined otherwise.

The data types T in Java consist of primitive types (such as `boolean` and `int`), names of declared classes, and the unit type (`void`). Data types are the types of fields, local variables, method parameters and method return; the unit type is used as return type for a method called only for its effect on state. Data types are given by the grammar

$$T ::= \texttt{bool} \mid \texttt{unit} \mid C$$

where C ranges over the set of declared class names. This is formalized as a PVS inductive datatype, `dty`, in theory `dty` which is parameterized on a nonempty set `Classname` used as argument for the constructor `classT(name: Classname)`. Datatype recognizers are named according to the usual convention, e.g., `classT?`.

Theory `classtableSig` formalizes the signature of a well formed class table, i.e., fields, superclass, and method signatures for every class. It declares uninterpreted sets `Classname`, `Methname`, and `Varname` (the latter for fields and also local variables and parameters). Unlike the paper, we do not formalize the syntax of class declarations but rather work directly with the auxiliary functions `super`, `fields`, etc.; e.g., the following uninterpreted function declaration gives method signatures:

```
mtype: [Classname, Methname
        -> lift[ [# parN: Varname, parT: dty, resT: dty #] ] ]
```

Undefinedness is represented using the lift constructor, as PVS is a logic of total functions (`[T->U]` is notation for functions, `[# lab:T #]` for records, and `r`lab` for field selection). If `mtype(m,C)` is defined (not `bottom`) then its value is a record with field `parT` giving the parameter type, `resT` giving the result type, and `parN` giving the parameter name. As in the paper, the parameter name is treated as part of the type; this loses no generality and simplifies definitions related to method call.

Method types in Java are invariant, i.e., if `C` is a subclass of `D` and inherits or overrides a method `m` of `C` then the method signatures are the same. This is expressed by

```
inherit_meths: AXIOM C <= D & up?(mtype(D,m)) => mtype(C,m) = mtype(D,m)
```

Axiom `inherit_meths` also embodies inheritance in the sense that if `m` is defined in a class then it is also defined in subclasses thereof. The set of methods defined for a given class is lifted to the level of types as follows:

```
definedMeth(C)(m): bool = up?(mtype(C,m))
DefinedMeth(C): TYPE = (definedMeth(C))
```

The first line defines a predicate and the second line uses the PVS notation of enclosing parentheses to lift a predicate to a type, here the type of methods either declared or inherited in class `C`. The declaration and inheritance of fields is treated similarly.

The declaration-based subclassing relation turned out to be slightly intricate to formalize. Theory `classtableSig` declares an uninterpreted function `super` to designate the immediate superclass (except for mapping the distinguished class `Object` to itself).

```
super: [Classname -> Classname]
super_fin_top: AXIOM EXISTS j: j > 0 & iterate(super,j)(C) = Object
```

Here `C` ranges over class names owing to the declarations

```
C, D, E:   VAR Classname          T, T1, T2: VAR dty
```

In the sequel we omit such declarations. Subclassing on class names is defined by

```
<=(C, D): bool = EXISTS j:  iterate(super,j)(C) = D
```

and this extends easily to a relation `<=` on data types. Lemmas state that `Object` is the top element and that `<=` is a preorder. The proofs require minimal user guidance, e.g., here is the script for transitivity of `<=` on class names:

```
(skosimp) (expand "<=") (lemma "iterate_add_applied[Classname]") (grind)
```

It uses a lemma about the `iterate` function from the PVS prelude as well as the most powerful strategy, `grind`, which repeatedly applies simplification, instantiation, skolemization and if-lifting.

Many of the results are proved using a secondary induction on inheritance chains, formalized using the following (where `/=` is disequality).

```
stepsToObj(C)(j): bool  =  iterate(super,j)(C) = Object
cdepth(C): nat = min(stepsToObj(C))
cdepth_super: LEMMA C /= Object => cdepth(super(C)) < cdepth(C)
```

Theory classtableSig includes a number of additional definitions and results concerning subclassing and inheritance. It also defines contexts for use in typing rules. In the paper, a (variable) context Γ is a partial function from variable names to data types. It is formalized in PVS as a dependent record type.

```
Vxt: TYPE = [# dom: set[Varname], map: [(dom) -> dty] #]
```

Here (dom) is notation for the lift of predicate dom to the level of types. The type set[Varname] is syntactic sugar for the function type [Varname -> bool].

In the semantics, the state on which a command operates includes a type correct assignment of values to variables (i.e., parameters and locals including self) and the state of an object in the heap is a type correct assignment of values to field names. The context for fields is defined as follows, using set comprehension notation { f |...}.

```
fieldVxt(C: Classname): Vxt =
  (# `dom := { f | fields(C)(f) }, `map := lambda f: ftype(C,f) #)
```

Similarly, methParVxt(C)(m) declares self:C and also the method's parameter.

Typing of method bodies. The context free syntax of expressions and commands is given by PVS datatypes exp and com in a theory lang which is parameterized on the sets Classname, Varname, and Methname. An assignment $x := e$ is represented by the term assign(x,e,T) with an explicit trace of the type of e to simplify type checking; similarly for the other constructs. The expressions are: variables, boolean literals, null, equality test, field access, type test, and type cast. The commands are: assignment, field update, new, dynamically dispatched method call, sequence and conditional. (Loops are omitted since general recursion is included.)

Theory typing gives the typing rules for expressions and commands. The typing judgement $\Gamma \vdash e : T$ is expressed by a predicate expOK(V,T)(e) where V ranges over contexts, T over data types, and e over expressions.

```
expOK(V,T)(e): RECURSIVE bool =
  CASES e OF
  vblV(n):  V`dom(n) & T = V`map(n) ,
  fieldAccess(e1,T1,f): expOK(V,T1)(e1) & classT?(T1) & ftype(name(T1),f)=T
 ... MEASURE e by <<
```

This uses pattern matching on the datatype exp of expressions. A variable named n has type T in context V if n is in the domain of V and is assigned type T. Recursive definitions in PVS must be proved total using an explicitly designated measure, in this case the subterm relation << generated automatically by the datatype definition for exp.

For commands, the paper's judgement $\Gamma \vdash S$ is formalized as follows.

```
comOK(V)(S): RECURSIVE bool =
  CASES S OF
    assign(n,e,T): n /= Self & V`dom(n) & expOK(V,T)(e) & T <= V`map(n)...
```

The target n of assignment cannot be self and must be declared. The expression must have the type recorded T in the syntax and T must be a subtype of the type of n. The typing rules here and in the paper are syntax directed, so the semantics can be defined

by recursion on syntax. Subsumption is not present as a separate rule but is instead embodied by subtyping conditions.

All of the TCCs in theories `typing` and `classtableSig` are proved by the default strategy without user intervention.

Theory `wellformedCT` declares an uninterpreted function for method bodies.

```
 mbody: [C: Classname, m: Methname ->
             lift[[# localvar: Varname, localvarType: dty, body: com #]] ]
```

To simplify the formalization slightly, each method is assumed to have exactly one local variable. The formalization also enforces that the local variable name is distinct from the distinguished names `self` and `result` as well as from the parameter name (not shown here).

Every method should have an implementation and every method declaration should be typable; this assumption is expressed as follows.

```
declaredMeth(C)(m): bool = definedMeth(C)(m) & up?(mbody(C,m))
DeclaredMeth(C): TYPE = (declaredMeth(C))
every_meth_has_body: AXIOM  definedMeth(C)(m) & NOT declaredMeth(C)(m)
                             => C /= Object & definedMeth(super(C))(m)
all_bodies_typable: AXIOM
  declaredMeth(C)(m) => comOK( bodyVxtFor(C,m) )( down(mbody(C,m))`body )
```

Semantic domains. According to the paper, the state of a method in execution is comprised of a *heap h*, which is a finite partial function from locations to object states, and a *store r* which assigns locations and primitive values to local variables and parameters. States are self-contained in the sense that all locations in fields and in variables are in the domain of the heap. For locations, we assume that a set *Loc* is given, along with a distinguished entity *nil* not in *Loc*. To track an object's class we assume given a function $loctype : Loc \to Classname$ such that there are infinitely many locations ℓ with $loctype\ \ell = C$, for each C. In the formalization, it suffices to assume that the allocator is a total function. Theory `semanticDomains` begins with the uninterpreted declarations `Loc: TYPE+` and `loctype: [Loc -> Classname]`. It imports theory `value[Loc]` which defines a datatype of values with constructors `semNil`, `semLoc(valL: Loc)`, etc. These are classified as follows.

```
locsBelow(C)(l: Loc): bool = loctype(l) <= C
LocsBelow(C): TYPE = (locsBelow(C))
valOfType(T)(v: Value): bool =
  CASES T OF
  unitT:     semIt?(v) ,
  boolT:     semBool?(v) ,
  classT(C): semNil?(v) OR ( semLoc?(v) & locsBelow(C)(valL(v)) ) ENDCASES
ValOfType(T): TYPE = (valOfType(T))
val_subsumptionT: LEMMA  T <= T1 & valOfType(T)(v) => valOfType(T1)(v)
```

A number of subsumption properties are needed in the semantics. Formulation is a little delicate, e.g., in a case like `val_subsumptionT` this very property is needed to discharge a TCC for `valOfType(T1)(v)`. Once the TCC has been proved, one can

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

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

$$\begin{aligned}
[\![\Gamma]\!] &= \{r \mid dom\, r = dom\, \Gamma \wedge r\,\mathsf{self} \neq nil \wedge \forall x \in dom\, r \bullet r\, x \in [\![\Gamma\, x]\!]\} \\
[\![objstate C]\!] &= \{s \mid dom\, s = dom(fields C) \wedge \forall (f : T) \in fields C \bullet s\, f \in [\![T]\!]\} \\
[\![Heap]\!] &= \{h \mid dom\, h \subseteq_{fin} Loc \wedge closed\, h \wedge \forall \ell \in dom\, h \bullet h\ell \in [\![objstate\,(loctype\,\ell)]\!]\} \\
&\quad \text{where } closed\, h \text{ iff } rng\, s \cap Loc \subseteq dom\, h \text{ for all } s \in rng\, h \\
[\![Heap \otimes \Gamma]\!] &= \{(h, r) \mid h \in [\![Heap]\!] \wedge r \in [\![\Gamma]\!] \wedge rng\, r \cap Loc \subseteq dom\, h\} \\
[\![Heap \otimes T]\!] &= \{(h, v) \mid h \in [\![Heap]\!] \wedge v \in [\![T]\!] \wedge (v \in Loc \Rightarrow v \in dom\, h)\} \\
[\![C, \overline{x}, \overline{T} \to T]\!] &= [\![Heap \otimes (\overline{x} : \overline{T}, \mathsf{self} : C)]\!] \to [\![(Heap \otimes T)_\perp]\!] \\
[\![MEnv]\!] &= \{\mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } mtype(m, C) \text{ is defined,} \\
&\qquad \text{and } \mu C m \in [\![C, pars(m, C), mtype(m, C)]\!] \text{ if } \mu C m \text{ defined } \}
\end{aligned}$$

Table 1. Semantic domains, named by categories θ including *MEnv* for method environments; $(C, \overline{x}, \overline{T} \to T)$ for methods of C with parameters $\overline{x} : \overline{T}$, return T; and $Heap \otimes \Gamma$ for closed states.

appeal to it to get an immediate proof of the lemma. But the lemma needs to be explicitly stated in order to generate the TCC.

In the paper, notation for the hierarchy of semantic domains is based on syntactic categories θ, see Table 1. These categories are not separately formalized in PVS. Instead there are just named types, e.g., for $[\![\Gamma]\!]$ we define `Store(V)`. (Recall that identifier `V` is used instead of Γ.) A store for context `V` maps each name `n` in the domain of `V` to a value of the type assigned to `n` in `V`.

```
Store(V): TYPE = [n: (V`dom) -> ValOfType(V`map(n))]
```

Heaps are defined in two stages, the second imposing the containment condition.

```
ObjState(C): TYPE = Store(fieldVxt(C))
preHeap: TYPE = [# dom: finite_set[Loc],
                   map: [ l:(dom) -> ObjState(loctype(l)) ] #]
closedStore(V)(h: preHeap)(r: Store(V)): bool =
  FORALL (n:(V`dom)):
   classT?(V`map(n)) & NOT semNil?(r(n)) => h`dom(valL(r(n)))
heap(h: preHeap): bool =
  FORALL ( l:(h`dom) ):
   LET V = fieldVxt(loctype(l)), r = h`map(l) IN closedStore(V)(h)(r)
Heap: TYPE = (heap)
state(V)(h: Heap, r: Store(V)): bool = closedStore(V,h)(r)
State(V): TYPE = (state(V))
```

If `s: State(V)` then `s`1` is the heap part, using the PVS projection notation.

In the paper, no domains are explicitly defined for expressions but it is stated that the meaning of an expression $\Gamma \vdash e : T$ is a function $[\![Heap \otimes \Gamma]\!] \to [\![T_\perp]\!]$ that takes a state $(h, r) \in [\![Heap \otimes \Gamma]\!]$ and returns either a value $v \in [\![T]\!]$, such that $(h, v) \in [\![Heap \otimes T]\!]$ (i.e., v is in $dom\, h$ if v is a location), or the improper value $\perp$ which represents errors. In the formalization the domain for expressions is made explicit:

```
preExpr(V,T): TYPE = [ State(V) -> lift[ValOfType(T)] ]
contValOfType(T, h)( v ): bool =
  valOfType(T)(v) & ( classT?(T) & NOT semNil?(v) => h‘dom(valL(v)) )
CValOfType(T: dty, h: Heap): TYPE = (contValOfType(T,h))
semExpr(V,T)(g: preExpr(V,T)): bool =
  FORALL (s:State(V)): liftContValOfType(T,s‘1)(g(s))
SemExpr(V,T): TYPE = (semExpr(V,T))
```

The paper says that a typable command $\Gamma \vdash S$ denotes a function $[\![MEnv]\!] \to [\![Heap \otimes \Gamma]\!] \to [\![(Heap \otimes \Gamma)_\perp]\!]$. In proving noninterference, which involves extending bijections on the heap domain, it became apparent that commands must have an additional property not made clear in the paper: Any location allocated in the initial heap is still allocated in the final heap. (By not modeling garbage collection, we simplify the formulation of noninterference.) This condition could be proved as a lemma but is instead imposed on the semantic domains for commands (and also method meanings). This is formalized as follows.

```
preSemCommand(V): TYPE = [ State(V) -> lift[State(V)] ]
semCommand(V)(g: preSemCommand(V)): bool =
  FORALL (s: State(V), l:(s‘1‘dom)):  up?(g(s)) => down(g(s))‘1‘dom(l)
SemCommand(V): TYPE = (semCommand(V))
```

Finally, the denotation of a method `m` takes a pair `(h,r)`, where `r` is a store with the arguments, i.e., `self` and for the parameter. (An alternate formulation would pass a tuple of values, with the benefit that the parameter name would not be significant in method types. The cost would be additional definitions of indistinguishability etc. for tuples.) It returns $\perp$ or a pair `(h1,v)` where `v` is the result value and `h1` the updated heap. The semantic domain `SemMeth(C,m)` imposes the conditions that `v` is in the domain of `h1` and that the domain of `h1` extends that of `h` (the latter condition is missing from the equation for $[\![(C, \overline{x}, \overline{T} \to T)]\!]$ in Table 1).

Method environment. The semantics for commands, discussed later, is defined in terms of a method environment which provides for each defined method an appropriate semantics.

```
MethEnv: TYPE = [C: Classname -> [m: DefinedMeth(C) -> SemMeth(C,m)]]
```

Theory `semCT` defines the semantics of a class table as a method environment obtained as the limit of a chain of approximations.

```
approxMethEnv(j)(C): RECURSIVE [m:DefinedMeth(C) -> SemMeth(C,m)] =
  lambda (m:DefinedMeth(C)):
     IF j=0 THEN abortMeth(C,m)
     ELSE IF declaredMeth(C)(m) THEN mbodySem(C,approxMethEnv(j-1))(m)
     ELSE restrict[...]( approxMethEnv(j)(super(C))(m) ) ENDIF ENDIF
   MEASURE lex2(j, cdepth(C)) BY <
```

For the 0th approximation, every method aborts. For the jth approximation, if m is declared in C then its semantics is given in terms of the semantics of its body, interpreted with respect to the $(j-1)$st approximation for methods it calls. (We defer the definition of `mbodySem`.) This exactly mirrors an operational semantics in which the call stack

size is bounded by $j-1$ (note that there is no stack in the denotational model). The semantics of a complete program has a one-line definition:

```
semCT: MethEnv = lub( approxMethEnv )
```

This uses the least upper bound operator defined in theory `orderDomains`, where it is shown that the semantic constructs are monotonic and continuous. Operationally, taking the `lub` removes the bound on stack size. We omit further details.[1]

Returning to `approxMethEnv`, if *m* is inherited then its semantics is inherited, i.e., given by `approxMethEnv(j)(super(C))(m)`. But this function is applied to a smaller domain (the argument stores where `self` has type `C` rather than `super(C)`), whence the need for `restrict[...]` to make the conversion explicit for the PVS type checker.

The definition of `approxMethEnv` generates eight TCCs. One is that the given order is well founded. Three of the TCCs require more than one step of user interaction, mainly to show that `approxMethEnv(j)(super(C))(m)`, which has type `SemMeth(super(C),m)`, also has type `SemMeth(C,m)` (when its domain is restricted `State(methParVxt(C)(m))`). The proofs use axiom `every_meth_has_body`, lemma `cdepth_super`, and the following.

```
inherit_defined: LEMMA
   FORALL C, (m: DefinedMeth(super(C))), (g: SemMeth(super(C),m)):
     semMeth(C, m)(restrict[...](g))
```

For lemma `inherit_defined` the proof is easy:

```
(skosimp*)(typepred "g!1")(use "super_above")(use "semMeth_subsumpt")(prop)
```

On the other hand, to discharge the subtyping TCCs generated by `inherit_defined` requires several steps, in two of which instantiations needed to be supplied. The TCCs for this lemma are then used subsequent proofs.

Semantics of commands and method bodies. Theory `comSemantics` gives the semantics for commands, given an uninterpreted function for memory allocation.

```
fresh: [Classname, Heap -> Loc]
freshness: AXIOM  loctype(fresh(C,h)) = C & NOT h`dom(fresh(C,h))
```

A simple and realistic model for this assumption is to let *Loc* be the set of pairs (i,C) with *C* a classname and *i* a natural; *fresh* chooses the least unused *i*.

For each syntactic command construct we explicitly define a semantic operation, e.g., here is the semantic operation that allocates a fresh object and assigns it to `n`:

```
newS(V)(n: {n | V`dom(n) & n/=Self}, C: {C | below(V`map(n))(classT(C))})
       (s: State(V)): lift[State(V)] =
  LET l = fresh(C, s`1),
      h1 = (# `dom := add(l, s`1`dom),
              `map := s`1`map WITH  [ (l) |-> initObState(C)] #)
  IN up( updateVar(V)(n, (h1, s`2), semLoc(l)) )
```

[1] The `lub` of method environments admits an elementary characterization, owing to the simple concrete representation for states. If method meanings were stored in the heap then we would need to work with the solution of a nontrivial domain equation. Indeed, Levy [12] gives a denotational model for a higher order language with pointers, but the model does not capture relational parametricity or recursive types.

The semantic function comSem has a succinct definition, mapping each construct to its corresponding semantic operation. In the paper, the semantics $[\![\Gamma \vdash S]\!]$ is only defined for derivable judgement $\Gamma \vdash S$. A direct encoding of this would lift the predicate for typing, comOK(V), to a type, but then we would have to state and prove an induction rule for the set of typable commands. In order to use the induction rule generated by PVS for the datatype com, i.e. for proofs to begin simply (induct "S"), we define semantics for all commands, typable or not.

```
comSem(V: Vxt, me: MethEnv)(S: com): RECURSIVE SemCommand(V) =
  IF NOT comOK(V)(S) THEN abortCom(V) ELSE
  CASES S OF
    new(n, C):        newS(V)(n, C) ,
    seq(S1,S2):       seqS(V)( comSem(V,me)(S1), comSem(V,me)(S2) ) ,
    mcall(n,e1,m,e2,T1,T2):   % that is, n := e1.m(e2)
       mcallS(V,me)( n, T1, expSem(V,T1)(e1), m, expSem(V,T2)(e2) ) ,
 ...MEASURE S by <<
```

Type soundness. The theories discussed so far give a deep embedding of the syntax and semantics of a fragment of Java with features including mutable objects, recursive class definitions, type casts, pointer equality, and inheritance. The semantic domains express invariants like these: self is not null; objects are never deallocated; the value of any field of any object is an element of the (denotation of) the type declared for the field. The net effect is that type soundness is a consequence of definedness of the semantics. To see how this works, consider an assignment assign(n,e,T) typable in context V. The semantics is assignS(V)(n, T, expSem(V,T)(e)), using the following.

```
assignS(V)(n: {n|V`dom(n) & n/=Self}, T: Below(V`map(n)), g: SemExpr(V,T))
          ( s: State(V) ): lift[State(V)] =
   IF up?(g(s)) THEN up( updateVar(V)(n,s,down(g(s))) ) ELSE bottom ENDIF
```

Note that assignS(V) is applicable only to a variable n in the domain of V and not equal to self; moreover, the type T of the assigned expression must be a subtype of the type of n in V. The declaration indicates that assignS(V)(n,T,g) applies to a state for V and returns a lifted state. An additional result says that in fact it has the type SemCommand(V).

```
assignS_type: JUDGEMENT
   assignS(V)(n:{n|V`dom(n) & n/=Self}, T:Below(V`map(n)), g:SemExpr(V,T))
   HAS_TYPE SemCommand(V)
```

This can be read as a lemma, saying that if assign(n,e,T) typechecks then it denotes a function satisfying the invariants imposed by SemCommand(V). The definition of comSem generates three TCCs for assign, which can be discharged easily using assignS_type and inversion of the typing rule.[2]

The domain definitions, subsumption, inheritance, etc., come together in the semantics method call, mcallS. The TCCs for this definition are among the longer proofs in the development, as user interaction is needed to invoke various subsumption lemmas.

[2] The JUDGEMENT form provides refined type information to the type checker, to reduce redundant TCCs. It is of little use in our work because it does not cater for dependent types, e.g., JUDGEMENT loc(l: locsBelow(C)) HAS_TYPE ValOfType(classT(C)) is not allowed because C is free. Instead we use LEMMA FORALL (C:Classname, l: LocsBelow(C)): valOfType(classT(C))(loc(l)).

3 Formalizing the static analysis and noninterference

Security policy. Noninterference for a given security labeling means that if a pair of initial states are indistinguishable by an observer who only sees fields/variables labeled below level L then the corresponding final states are indistinguishable below L.

Theory `lattice` defines the relevant operations on an uninterpreted type, `Level`. Variables `L`, `L1` etc range over `Level`. Security policy assigns a level to each field as well as to method parameters and result. Theory `indistinguishable` defines the indistinguishability of states, with respect to a given policy.

```
methPolicy: TYPE = [Classname,Methname -> [# self, par, res, hp: Level #]]
fieldPolicy: TYPE = [Classname -> [Varname -> Level]]
```

The method policy for a given class and method gives not only levels for the parameters and result, serving as upper bounds on their information, but also a lower bound on the levels of fields written (`hp`). We refrain from tying policy to a particular class table signature, but rather let the policy assign arbitrary values for undefined methods and fields.

Indistinguishability of values involves a partial bijective renaming of locations to mask allocation effects. Theory `indistinguishable` develops suitable theory for type-respecting partial bijections on locations, e.g., how composition of such relations behaves. Indistinguishability of values of type `T`, relative to a typed bijection `rho`, is defined as follows:

```
indis( rho, T )( v, v?: ValOfType(T) ): bool =
   CASES T OF
     unitT: TRUE ,
     boolT: v = v? ,
     classT(C): ( semNil?(v) & semNil?(v?) )
             OR ( semLoc?(v) & semLoc?(v?) & rho( valL(v), valL(v?) ) )
```

Where the paper uses v, v' for related pairs, the PVS formalization uses `v?` because symbol $'$ is not allowed in identifiers. For this reason we mostly avoid the convention of using `?` to signal predicate names. The definition above induces the notions of indistinguishability (overloading the name `indis`) for each of the other semantic domains. We need the notion of a policy for stores:

```
Levels(V):   TYPE = [(V`dom) -> Level]
```

Indistinguishability of stores with respect to an observer of level `L` and local variable labeling `lev` is defined as follows:

```
indis( L, rho, V, (lev: Levels(V)) ) ( r, r?: Store(V) ): bool =
   FORALL (n:{n|V`dom(n) & lev(n)<=L}): indis(rho,V`map(n))( r(n), r?(n) )
```

The definition says that `r` is indistinguishable from `r?` for an observer at level `L` provided that the values for `n` are indistinguishable for every variable `n` with level at most `L`.

Confinement. Confinement expresses, semantically, the key rules "no read down" and "no write up" [15]. For an expression meaning g to be *read confined* for observers at level L means that it cannot distinguish states that are indistinguishable for L.

```
readConf(L, V, T, fpField)(levs: Levels(V))(g: SemExpr(V,T)): bool =
  FORALL rho, (s, s?: State(V)):
   indis(L,rho,V,fpField,levs)(s,s?) => indis(rho,T)(g(s),g(s?))
```

For a command to be *write confined* above L means that it writes no variable or field below level L; that is, its final state is indistinguishable, for observers not at a level above L, from the initial state, using as renaming bijection the identity on the domain of the initial heap. Write confinement is also defined for methods; details omitted.

Safety. Theory safe gives the static analysis in terms of uninterpreted policy functions and also an assignment of levels to the local variable(s) of method bodies.[3]

```
mPolicy: [Classname, Methname -> [# self, par, res, hp: Level #]]
fPolicy: [Classname -> [Varname -> Level]]
localLevel: [Classname, Methname, Varname -> Level]
```

It is assumed that the policies are invariant with respect to subclassing.

```
inherit_meth_lev: AXIOM C<=D & up?(mtype(D,m)) => mPolicy(C,m)=mPolicy(D,m)
```

In the paper, the static analysis is specified using labeled types (T, κ) for expressions, where T is a data type and κ a security level. For commands, the judgement $\Delta \vdash S : \kappa, \kappa'$ expresses that S is secure, with respect to a given policy assigning levels to locals (Δ), fields, and method parameters/returns, and moreover S writes fields (resp. locals) of level at least κ (resp. κ'). There is a similar judgement for expressions. The paper gives rules to inductively define these judgements but in PVS we define a recursive function that, for expression e, gives the least L at which e would be typable in the paper. This serves as a deterministic and reasonably efficient implementation.

In the paper, the analysis, like the semantics, is specified only for typable programs. In the PVS formalization, safety is defined for all programs but only typable programs are deemed safe. (Compare comSem.)

```
expSafe( V: Vxt, lev: Levels(V), T: dty )(e: exp): RECURSIVE lift[Level] =
  If NOT expOK(V,T)(e) THEN bottom ELSE
  CASES e OF vblV(n): up( lev(n) ) ,
             fieldAccess(e1, T1, f):
               s_lub( expSafe(V,lev,T1)(e1), up(fPolicy(name(T1))(f)) )...
```

For a method declaration to be safe, mSafe, is defined to mean that its body is safe with respect to the method policy. Function comSafe is similar to expSafe. Finally, safety of a whole program is defined as follows.

```
 safe_CT: bool = FORALL C, (m: DeclaredMeth(C)): mSafe(C)(m)
```

[3] The formalization could be improved by factoring out the policy from both theories safe and indistinguishable.

4 Main results

Confinement. Theory `confinement` shows that the analyzer ensures read and write confinement. For the semantic operation used to interpret each program construct, there is a lemma expressing conditions under which the semantics is confined. For example, here is the case that the expression is a variable:

```
vblS_read_conf: LEMMA FORALL ( V: Vxt, levs: Levels(V) ):
   V‘dom(n) & V‘map(n) = T & levs(n)<= L
   => readConf(L, V, T, fpField)(levs)( vblS(V)(n) )
```

These lemmas are used to prove that if the analysis returns level `L` for an expression then its semantics is read confined at that level. Similarly, if the analyzer says a command is safe at a certain level pair `LLp` then it is write confined for `LLp`.

```
safe_com_write_conf: LEMMA
  FORALL ( V: Vxt, lev: Levels(V), S: com, me: WriteConfMenv ):
   LET LLp = comSafe(V,lev,S)
   IN up?(LLp) => writeConf(down(LLp), V, fPolicy, lev)( comSem(V,me)(S) )
```

Note the antecedent that method meanings in `me` are write confined. The lemma is proved by induction on the structure of `S`; in each case one inverts the safety rule and uses lattice properties to establish the hypothesis for the corresponding semantic confinement lemma.

To discharge the assumption about method environment, we have to show that if all method bodies are safe then all their meanings are write confined. This is proved by showing that all approximations are write confined and that this implies the same for the least upper bound.

```
write_conf_approx: LEMMA
   safe_CT => (FORALL j: writeConfMenv( approxMethEnv(j) ))
write_conf_admissible: LEMMA
   FORALL ( app: AscendingMenvs ):
     (FORALL j: writeConfMenv(app(j))) => writeConfMenv( lub(app) )
safe_CT_write_conf: LEMMA  safe_CT => writeConfMenv( semCT )
```

Admissibility is a straightforward series of steps unfolding the definitions, owing to our explicit characterization of `lub` in theory `orderDomains`. For `write_conf_approx` the proof goes by induction on the lexical ordering of `j` and depth of inheritance (using lemma `cdepth_super`).

The least satisfying proof in this theory is the write confinement lemma for semantics of method call. It is long, mainly because a number of TCCs are generated at several points where lemmas are invoked. Some of the TCCs are discharged by a simple strategy which tries all the TCCs from the definition of the semantic function `mcallS`. For a number of others the strategy fails and I have to explicitly appeal to a specific one of the TCCs associated with the definition of `mcallS` in theory `comSemantics`.

Noninterference. It is only in the final theory, `safeProps`, that the TCCs are really onerous. First come definitions, e.g., what it means for a command to be noninterferent with respect to a policy `lev` for variables (the policy for fields is imported from theory `safe`) and an observer that can see variables and fields of level at most `L`:

```
nonint( L, V, (lev: Levels(V)) )( g: SemCommand(V) ): bool =
   FORALL rho, (s, s?: State(V)):
    indis(L,rho,V,fPolicy,lev)(s,s?) & up?(g(s)) & up?(g(s?))
    => EXISTS tau: subset?(rho,tau)
       & indis(L,tau,V,fPolicy,lev)(down(g(s)), down(g(s?)))
```

There is a similiar notion, `nonint(L,C,m)`, of noninterference for the meaning of method `m` at class `C`. A method environment is noninterferent provided every method meaning is noninterferent for observers at every level `L`. (The definition is factored into two to fit a standard induction rule.)

```
nonintMenv2( me, C ): bool =
  FORALL L, (m: DefinedMeth(C)):  nonint(L, C, m)( me(C)(m) )
nonintMenv( me ): bool = FORALL C: nonintMenv2(me,C)
```

Next come noninterference lemmas for the semantic operations, for example:

```
nonint_assignS: LEMMA
 FORALL L, V, (lev: Levels(V)),  (n: {n | (V`dom) & n/=Self)}),
        (T: Below(V`map(n))), (g: SemExpr(V,T)):
   readConf(lev(n),V,T,fPolicy)(lev)(g)
   => nonint(L,V,lev)( assignS(V)(n,T,g) )
```

The proof involves manipulating a pair `s,s?` of states related by `indis` and unfolding the definition of `assignS` on both `s` and `s?`. This generates two sets of TCCs. The TCCs can be discharged by using, as lemmas, the TCCs associated with the definition of `assignS`. But there is a problem: one wants two instantiations of the latter, one for each of the two states at hand. The PVS heuristics for instantiation fare poorly here and must be guided by manually deleting the distracting formulas in the sequent. Reference to these formulas is by line number and this makes the proof script brittle in the face of changes to the prover or the proof.

The noninterference lemma for `mcallS` is the most complex in the entire development. The proof itself is easy, but only because it is factored into three very complicated technical lemmas with long proofs. The semantics of method call is inherently complicated, so these lemmas involves two heaps, two target objects, two argument stores, etc. Each application of the semantics, `mcallS`, generates about eight TCCs as does each use of a lemma like monotonicity of `indis`. The strategy of trying each `mcallS` TCC until one works fails hopelessly here. I have to designate a specific one to use and also the instantiation of its top level variables. But this is not enough. There are nested quantifiers and in the worst case the TCC appears in a sequent with about 35 formulas. An explicit `hide` command is needed to remove half of these so the strategy `smash` can finish the job with simplification and heuristic instantiation. A specialized strategy to automate the `hide` step might work by removing formulas that mention identifiers in which `?` occurs, or those for which there is a matching formula with `?`. But this is neither principled nor straightforward to implement; is there another way?

The noninterference property for commands is as follows.

```
nonint_com: LEMMA
 FORALL L,V,S,(lev:Levels(V)),(me | nonintMenv(me) & writeConfMenv(me)):
     up?( comSafe(V, lev, S) ) => nonint(L, V, lev)( comSem(V,me)(S) )
```

The proof is by structural induction on S. For each command construct, the safety condition is unfolded and the noninterference lemma for that construct's semantics is invoked. For if/else, an instantiation is explicitly provided for a lemma on monotonicity of write confinement. But in the rest of the proof the heuristic instantiations work fine and there are no difficulties with TCCs. Finally, the main result is succinctly stated:

```
nonint_CT: THEOREM  safe_CT => nonintMenv( semCT )
```

Like the write confinement lemma for semantics of the class table, the proof is by measure induction (on the approximation chain and on inheritance), using nonint_com for method bodies in the induction step. User interaction is needed to introduce lemmas and in a couple of cases provide instantiations.

5 Discussion

The first objective of the project was met: the results of [2] (omitting code-based access control) were checked successfully. The complete import chain for theory safeProps comprises about 3K lines of PVS specifications. There are a total of 587 proofs which take approximately 30 minutes to check on a 1.2Ghz Intel CPU with .5G RAM (and took about 8 person-weeks to create). A rough guess for the user written proof scripts is 8K lines.[4]

As for the second objective, the expressiveness of the PVS language certainly lends itself to a semantic model of this sort. The facilities of PVS served well but the JUDGEMENT mechanism was of little use and heuristic instantiation of quantifiers leaves something to be desired. But the biggest problem with instantiation is in the contexts where there are two copies of everything in sight, which is likely to pose a problem for any prover and any style of modeling for noninterference.

The effort uncovered a bug in [2]: We had neglected to impose on the semantic domains that the domain of the heap never shrinks, but this was assumed implicitly in some proofs. To my suprise this was the only bug. This is not to say no mistakes were made during the PVS development: In the generalization to an arbitrary lattice of levels I formulated revised definitions off the top of my head, without proving correctness on paper. The third project objective was to find out whether PVS could serve as a proof assistant and help find the correct formulations. I was not disappointed.

The plan is to complete the project by adding access control to the language; this should offer some experience in "proof maintenance". It should be straightforward to extract executable code for the analyzer; this may be used to verify policies created by an inference tool being developed by Qi Sun [17], but it will require extension of the policy language to include level polymorphism. Another interesting project would be to add generics and prove type soundness; on paper the semantics has already been extended [3] to generics as in C# [10]. In joint work with Gary Leavens on foundations of JML [11], we plan to check rules for behavioral subclassing.

[4] A couple of the theories with complicated dependent types run afoul of PVS bugs which hinder use of some tools for exploring proofs. But I have no reason to doubt soundness of the proofs.

Acknowledgements. Natarajan Shankar and Patrick Lincoln arranged a Visiting Fellowship at SRI International for the month of September, 2003, during which time I learned PVS and more than half of this work was carried out. Harold Rueß's elegant domain theoretic work [4] was the leading example for my initial study. Shankar, Sam Owre, and Bruno Dutertre were particularly generous with explanations and advice; Bruno taught me that the way to smash a sequent is to `(smash)`. Walkthroughs by the whole group, with John Rushby guiding me in fluent emacs-keystroke, were very helpful and encouraging. César Muñoz helped with strategies for TCCs. Thanks to Anindya Banerjee for comments on this paper, not to mention correctness of [2].

References

1. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *15th IEEE Computer Security Foundations Workshop*, 2002.
2. A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2), 2003. Special issue on Language Based Security.
3. A. Banerjee and D. A. Naumann. State based encapsulation and generics. Technical Report CS-2004-11, Stevens Institute of Technology, 2004.
4. F. Bartels, H. Pfeifer, F. von Henke, and H. Rueß. Mechanizing domain theory. Technical Report UIB-96-10, 1996.
5. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Workshop on Issues in the Theory of Security (WITS)*. ACM, 2003.
6. L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
7. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–459, May 2001.
8. B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *Workshop on Issues in the Theory of Security (WITS)*. ACM, 2005.
9. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
10. A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*, 2001.
11. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, 2003.
12. P. Levy. Possible world semantics for general storage in call-by-value. In *Computer Science Logic*, LNCS 2471, 2002.
13. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, Saratoga, NY, June 1992.
14. J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, Dec. 1992.
15. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
16. M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.
17. Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Static Analysis Symposium*, LNCS 3148, 2004.
18. D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, LNCS 1214, 1997.