Formal Analysis of an Information Flow Type System for MicroJava

Martin Strecker

Fakultät für Informatik, Technische Universität München
`Martin.Strecker@in.tum.de`
`http://www.in.tum.de/~streckem`

Abstract. This paper presents a type-based information flow analysis for MicroJava, a substantial subset of sequential Java. The analysis is given in the form of a type-and-effect system, which is required for handling side-effecting expressions and dynamic method lookup. The type system and a proof of noninterference are formalized in the Isabelle proof assistant, extending a comprehensive existing model of MicroJava.

1 Introduction

There is growing concern that data which are meant to be secret or private, such as a PIN or a password, are divulged to persons not entitled to see them. The risk of unintentional information leakage is becoming even larger in an environment where many programs from different sources cooperate, such as applets on a multi-application smartcard. Therefore, recent studies of the smartcard industry [Eur] advocate checking code statically before deployment.

Type-based information flow analyses (see [SM03] for a survey) just begin to be applied to realistic programming languages: First investigations started out with a simple `while`-language [VSI96] and have gradually been extended to include procedures [VI97]. In the context of the ς calculus, [BS99] examine noninterference for an object oriented language.

Closer to an existing language, [BN02,BN03] provide an analysis for a fragment of Java similar to ours; both approaches can deal with heap-allocated objects. In contrast, our system includes side-effecting expressions. In conjunction with dynamic method lookup, this leads us to the definition of a type-and-effect system (see the discussion in Section 3.2). Jif [Mye99] is a full-fledged programming language, augmenting Java by security labels. To date, it is one of the most comprehensive implementations of a language with security features, but its metatheory remains to a large extent unexplored.

On quite a different branch, extensions of functional languages [HR98] have led to realistic ML implementations [PS03] with information flow types.

Our own work is based on MicroJava [NOP00], a formalisation of Java in the Isabelle proof assistant [NPW02]. Even though MicroJava is by far not a complete model of Java, it covers the most important features (see Section 2 for a summary). It is thus a further step in the direction of a realistic programming language; the only major omission for the purposes of our analysis is an appropriate treatment of exceptions.

Our information flow (IF) type system is added modularly on top of the existing formalisations (Section 3). Our type-and-effect system assigns two IF types to an expression, one of them an upper bound of the values read, the other a lower bound of the assignments performed in the expression.

We prove a classical noninterference result in the tradition of [VSI96], showing that observable equivalence is preserved for well-typed expressions during execution. We therefore first spell out the notion of equivalence (Section 4), then establish an invariant preserved by a single execution (Section 5) and finally arrive at the desired result by means of a bisimulation proof (Section 6). The proofs have been entirely carried out in Isabelle, and we have derived a type inference algorithm directly from the formalisation. Thus, a type-based analysis is itself made the object of a formal analysis. The paper concludes with a discussion of future extensions (Section 7).

Due to space limitations, we can only sketch our formalisation, and the description is not entirely self-contained. The full Isabelle sources are available from `http://www4.in.tum.de/~streckem/InfoFlow/`, for a more detailed exposition see [Str03].

2 Formalization of the Standard Language

In this section, we will summarize the existing MicroJava formalization, as far as relevant for our purposes. A more complete treatment is given in [NOP00].

Before, a few words on Isabelle are in order: Isabelle is a generic framework for encoding different object logics. In this paper, we will only be concerned with Isabelle/HOL [NPW02], which comprises a higher-order logic and facilities for defining data types as well as primitive and terminating general recursive functions.

Isabelle's functional fragment is reminiscent of ML (note that prefixing an element *x* to list *xs* is written as *x#xs*). For product resp. function space, we have the usual type constructors $T1 \times T2$ resp. $T1 \Rightarrow T2$.

Isabelle's logical fragment uses standard quantifiers and connectives, among them $\longrightarrow$ as implication of the HOL object logic. The long double arrow $\Longrightarrow$ is Isabelle's meta-implication, in the following mostly used in conjunction with rules of the form $[\![P_1; \ \ldots; \ P_n]\!] \Longrightarrow C$ to express that C follows from the premises $P_1 \ \ldots \ P_n$.

Isabelle has an elaborate syntax definition facility. We will introduce new notation (usually directly behind the corresponding type declaration) for some relations as we go along.

The polymorphic option type *datatype 'a option = None | Some 'a* is frequently used to simulate partiality in a logic of total functions: Here, *None* stands for an undefined value, *Some x* for a defined value *x*. Lifted to function types, we obtain the type of "partial" functions *T1* $\rightharpoonup$ *T2*, which just abbreviates *T1* $\Rightarrow$ *(T2 option)*.

2.1 Term and Class Structure

MicroJava is a reduced version of Java which accommodates essential aspects of Java, like classes, subtyping, object creation, inheritance and dynamic binding. A few system exceptions are supported; user-defined exceptions are not, neither is exception handling. The formalization abstracts away from most arithmetic data types, interfaces, arrays and multi-threading. It is a good approximation of the JavaCard dialect of Java, targeted at smart cards.

We do not explicitly spell out the language constructs available in MicroJava – they become apparent from the typing rules in Figure 1. Note that, in order to ease the definition of the type system, field access and assignment and method call contain annotations (in braces) of the defining class.

MicroJava standard types *ty* are made up of primitive types (essentially *Boolean* and *Integer*) and reference types.

From these, we define field declarations *fdecl* and a method signatures *sig* (method name and list of parameter types). A method declaration *mdecl* consists of a method signature, the method return type and the method body, whose type is left abstract. The method body type *'c* remains a type parameter of all the structures built on top of *mdecl*, in particular *class* (superclass name, list of fields and list of methods), class declaration *cdecl* (holding in addition the class name) and program *prog* (list of class declarations).

```
types fdecl    = vname × ty
      sig      = mname × ty list
      'c mdecl = sig × ty × 'c
      'c class = cname × fdecl list × 'c mdecl list
      'c cdecl = cname × 'c class
      'c prog  = 'c cdecl list
```

The definition of *mdecl* is parametric in order to be usable on the source- and bytecode level. For the source level, we take the instantiation *java_mb prog*, where *java_mb* consists of a list of parameter names, list of local variables (i.e. names and types), and a statement block, terminated with a single result expression (this again is a deviation from original Java).

```
types java_mb   = vname list × (vname × ty) list × stmt × expr
      java_prog = java_mb  prog
```

2.2 Standard Type System

The standard type system uses two kinds of typing judgements, defined inductively:

– $E \vdash$ `e :: T` means that expression `e` has type `T` in environment `E`.
– $E \vdash$ `c` $\surd$ means that statement `c` is well-typed in environment `E`.

The *environment* `E` used here is `java_mb env`, a pair consisting of a Java program `java_mb prog` and a mapping `lenv` from variable names to types.

A program `G` is well-formed (`wf_java_prog G`) if the bodies of all its methods are well-typed and in addition some structural properties are satisfied – mainly that all class names are distinct and the superclass relation is well-founded.

2.3 Operational Semantics

The operational semantics is defined in the style of a big-step (natural) semantics with the aid of two inductive relations:

– `G` $\vdash$ `s -e`$\succ$ `v-> s'` says that for program `G`, expression `e` yields value `v` when evaluated in start state `s`. Since expressions are side-effecting, evaluation produces a new state `s'`.
– `G` $\vdash$ `s -c`$\rightarrow$ `s'` says that for program `G`, execution of statement `c` in state `s` yields a new state `s'`.

Here, values are: `val = Unit | Null | Bool bool | Intg int | Addr loc`

`Unit` is a (dummy) result value of void methods, `Null` a null reference, and `Addr` a reference to locations `loc`.

The *states* (of type `xstate`) manipulated by the evaluation rules are triples, consisting of an optional exception component that indicates whether an exception is active, a heap `aheap` which maps locations `loc` to objects, and a local variable environment `locals` mapping variable names to values.

The operational semantics and the standard type system are related by a type soundness result; in Section 5.2, we prove a similar theorem for the IF type system.

3 IF Typing

We are now prepared to define the IF type system, which is built on top of the existing MicroJava formalization of Section 2. It would have been possible to integrate IF types into the standard typing judgements of Section 2.2. We have opted against it in order to keep the two type systems as modular as possible and to avoid cluttering up the rules.

3.1 Information Flow Types

IF types are assumed to form a complete lattice. We enforce this requirement with the aid of Isabelle's axiomatic type classes: Type annotations of the form `'l :: complete_lattice` indicate that type `'l` is constrained accordingly and comes equipped with the usual operations such as supremum $\sqcup$, infimum $\sqcap$

and less-or-equal $\sqsubseteq$. Below, we also use the strict version $\sqsubset$ of $\sqsubseteq$ and functions *inf_list*, *sup_list* for taking supremum and infimum of lists.

These axiomatic type classes can later be instantiated, for example to a lattice of IF types *ity* which just consists of two values *Lw* and *Hg*, corresponding to two security levels "low" and "high" (or "public" and "secret"): *datatype ity = Lw | Hg*. When talking about "types" in the following, we will refer to IF types unless indicated otherwise.

3.2 Typing Rules

Judgements The IF typing judgements resemble very much the ones of the standard type system, with the difference that an additional type keeps track of the effect of a statement resp. a side-effecting expression. More precisely,

- $E\vdash_i$ *e :: (Te,Tc)* says that expression *e* has expression type *Te* and effect type *Tc* (where both *Te* and *Tc* are IF types)
- $E\vdash_i$ *c :: Tc* says that statement *c* has effect type *Tc* (where *Tc* is an IF type)
- $E\vdash_i$ *es[::]Ts* says that the expressions in list *es* have the expression and effect types in *Ts* (where *Ts* is a list of pairs of IF types)

The informal meaning of $E\vdash_i$ *e :: (Te,Tc)* is that during evaluation of *e*, only values of security level *Te* or below are read and only locations of level *Tc* or above are modified (this is expressed more formally by type soundness (Theorem 1) in Section 5.2). Thus, *Te* is an upper bound of the reads and *Tc* a lower bound of the writes performed in *e*.

We keep track of IF type information by adjoining an (unspecified) IF typing component to programs, obtaining a type *iprog*. In analogy to the standard type system, the IF environment *E* is a pair consisting of a program and a mapping of local variables to IF types:

```
typedecl 'l ity_info

types
  ('c, 'l) iprog = 'c prog × 'l ity_info
  'l ilenv = vname ⇀ 'l
  ('c, 'l) ienv = ('c, 'l) iprog × 'l ilenv
```

Example 1. Given variables *x: Hg, y: Lw*, an assignment *x ::= y* could be typed by *(Lw, Hg)*. Since we have no uniqueness of types, we could also give it the type *(Hg, Lw)*, which would be less precise (cf. Lemma 1). If, instead, the variables are typed by *x: Hg, y: Hg*, the second typing remains valid, but not the first one, since it fails to record that the *Hg* variable *y* is read.

Class, field and method types For stating the typing rules, we need an assignment of IF types to classes, fields and methods. We do this by declaring functions

```
clevel :: [('c,'l) iprog, cname] ⇒ 'l
flevel :: [('c,'l) iprog, cname, vname] ⇒ 'l
mlevel_ass :: [('c,'l) iprog, cname, sig] ⇒ 'l mlevel_tp
```

Here, we have used the types *cname* (class name), *vname* (variable name) and *mname* (method name). These functions are implicit parameters of the subsequent development. Thus, *clevel G C* is the IF type of class *C* in program *G*, similarly *flevel G C fn* the type of field *fn* in class *C*.

The treatment of methods is more complex. We are aiming at a function *mlevel* which takes a program *G*, a class *C* and a signature *sig* and which is compatible with method lookup (*method*) in the sense that whenever method lookup returns the same method for different *G, C, sig*, function *mlevel* yields the same method type. This is achieved by defining, with the aid of *mlevel_ass*:

```
mlevel :: [('c,'l) iprog, cname] ⇒ sig ⇀ 'l mlevel_tp
mlevel G C sig ==
  case method (prg G,C) sig of
       None ⇒ None
     | Some (D, rT, mb) ⇒ Some (mlevel_ass G D sig)
```

What should the IF type of a method declaration be? It assigns types to the formal parameters and the local variables and specifies the expected expression type and effect type, just as for expressions. Altogether, this gives

```
'l mlevel_tp =  ('l list) × ('l list) × 'l × 'l
```

Rules The typing rules for expressions, expression lists and statements are shown in Figure 1.

Before commenting individual rules, let us first make some general remarks: A common pattern found in most rules is that the expression type of an expression is bounded by the supremum (similarly the effect type by the infimum) of its subexpressions, as for example in rule *BinOp*, where $Te1 \sqcup Te2 \sqsubseteq Te'$ and $Tc' \sqsubseteq Tc1 \sqcap Tc2$. Given a valid typing of an expression, we can raise the level of its expression type and lower the level of its effect type, thus making the information flow analysis less precise, as witnessed in Example 1. This is expressed more formally in the following

Lemma 1. *lemma ity_expr_mono:*
$[\![E \vdash_i e{::}\ (Te,Tc);\ Te \sqsubseteq Te';\ Tc' \sqsubseteq Tc\ ]\!] \Longrightarrow E \vdash_i e{::}\ (Te',Tc')$

A similar result does not hold for statements: The rule *iLoop* constrains the result type *Tc'* to be higher than the expression type *Te* of the termination condition.

Let's now comment on the rules: Rule *iNewC* is obvious (however, see the discussion at the end of Section 6). A type cast does not modify the dynamic type of an object and therefore does not change its informational content. A literal value *per se* does not contain any information; therefore, an arbitrary type can be selected for it.

```
inductive ity_expr ity_exprs intros
  iNewC: ⟦ clevel (prg E) C  ⊑ Te; Tc ⊑ clevel (prg E) C ⟧ ⟹
         E ⊢i NewC C :: (Te, Tc)
  iCast: ⟦ E ⊢i e::(Te, Tc)    ⟧ ⟹
         E ⊢i Cast D e:: (Te,Tc)
  iLit:  E ⊢i Lit x :: (Te, Tc)
  iBinOp: ⟦ E ⊢i e1::(Te1, Tc1);  E ⊢i e2::(Te2,Tc2);
            Te1 ⊔ Te2 ⊑ Te'; Tc' ⊑ Tc1 ⊓ Tc2 ⟧ ⟹
          E ⊢i BinOp bop e1 e2::(Te', Tc')
  iLAcc: ⟦ localT E v = Some Tv; Tv ⊑ Te ⟧ ⟹
         E ⊢i LAcc v::(Te, Tc)
  iLAss: ⟦ E ⊢i e::(Te, Tc); localT E v = Some Tv;
           Te ⊑ Tv; Te ⊑ Te'; Tc' ⊑ Tv ⊓ Tc ⟧ ⟹
         E ⊢i v::=e:: (Te', Tc')
  iFAcc: ⟦ E ⊢i e:: (Te, Tc);  flevel (prg E) fd fn ⊑ Te ⟧ ⟹
           E ⊢i {fd}e..fn:: (Te, Tc)
  iFAss: ⟦ E ⊢i e1 ::(Te1, Tc1); E ⊢i e2 ::(Te2, Tc2);
           Tf = flevel (prg E) fd fn;
           Te1 ⊔ Te2 ⊑ Tf; Te1 ⊔ Te2 ⊑ Te';
           Tc' ⊑ Tf ⊓ Tc1 ⊓ Tc2 ⟧ ⟹
         E ⊢i {fd}e1..fn:=e2::(Te', Tc')
  iCall: ⟦ E ⊢i e:: (Te, Tc);  E ⊢i ps[::]piTs;
           mlevel (prg E) C (mn, pTs') = Some (piTes', liTs, Tem, Tcm);
           list_all2 (op ⊑) (map fst piTs) piTes';
           Teps = sup_list (map fst piTs);
           Tcps = inf_list (map snd piTs);
           Te ⊔ Teps ⊔ Tem ⊑ Te';  Tc' ⊑  Tc ⊓ Tcm ⊓ Tcps;
           Te ⊔ Teps ⊑ Tcm ⟧ ⟹
         E ⊢i {C}e..mn({pTs'}ps) :: (Te', Tc')

  iNil: E ⊢i [][::][]
  iCons: ⟦ E ⊢i e:: (Te, Tc);  E ⊢i es[::]Ts ⟧ ⟹
        E ⊢i e#es[::] (Te, Tc)#Ts

inductive ity_stmt intros
  iSkip: E ⊢i Skip :: Tc
  iExpr: ⟦ E ⊢i e::(Te, Tc) ⟧ ⟹
         E ⊢i Expr e:: Tc
  iComp: ⟦ E ⊢i  s1 :: T1;   E ⊢i s2 :: T2; Tc' ⊑ T1 ⊓ T2 ⟧ ⟹
         E ⊢i  s1;;s2 :: Tc'
  iCond: ⟦ E ⊢i e:: (Te, Tc);
          E ⊢i s1 :: T1;   E ⊢i s2 :: T2;
          Te ⊑ T1 ⊓ T2;  Tc' ⊑ Tc ⊓ T1 ⊓ T2⟧ ⟹
         E ⊢i If(e) s1 Else s2 :: Tc'
  iLoop: ⟦ E ⊢i e:: (Te, Tc);   E ⊢i s :: Tc';
          Te ⊑ Tc'; Tc'⊑  Tc ⟧ ⟹
         E ⊢i While(e) s :: Tc'
```

Fig. 1. IF typing rules for expressions, expression lists and statements

After the motivation of the typing judgements in Section 3.2, the rules for local variable access and assignment, `iLAcc` and `iLAss`, should be clear. The rules for field access and assignment, `iFAcc` and `iFAss`, are similar; the field level `flevel` corresponds to the local variable type `localT`.

In rule `iCall`, we first determine the expression and effect types `Te, Tc` of the object on which the method `mn` is invoked, and the types of the actual parameters, `piTs`. The first condition requires the actual parameter effect types (`map fst piTs`) to be compatible ($\sqsubseteq$) with the formal parameter effect types (`piTes'`). As usual, the resulting expression and effect types `Te', Tc'` are bounded by the object's type, the parameter types and the corresponding types of the method declaration `Tem, Tcm`. What is most noteworthy is the condition `Te` $\sqcup$ `Teps` $\sqsubseteq$ `Tcm`, which is meant to prevent an implicit information flow: the expression `e` and the parameters `ps` determine, via dynamic lookup, which method is selected and, consequently, which code is executed. Therefore, they have a similar influence as conditions in branching statements and loops.

As for statements: The rule `iExpr` drops the expression type and only keeps the effect type. Rule `iCond` for conditionals is very much the same (if `T1 = T2`) as the one in [VSI96], but leaves more freedom in the absence of antimonotonicity of effect types for statements in the sense of Lemma 1. The only genuine difference is with rule `iLoop`, which requires the result effect `Tc'` to be the same as the effect of the loop body, in contrast to [VSI96], where the result effect may be smaller than the effect of the loop body. A closer analysis reveals an erroneous use of an induction hypothesis in the proof of [VSI96] (the induction hypothesis would result from a structural induction and not from an induction over the evaluation relation).

3.3 Well-Formedness

Well-formedness criteria establish "global" type properties, as compared to the "local" conditions of the typing rules. Among them are the correspondence between standard and IF type system and the correspondence between declared and actual method types.

A class declaration for class `C` is well-formed if

- all subclasses `C'` have a lower `clevel`
- method declarations in subclasses have covariant expression types, contravariant effect types and contravariant parameter types

This is summarized in the definition:

```
wf_i_cdecl   :: [('c, 'l::complete_lattice) iprog, cname] => bool
wf_i_cdecl iG C == ∀ C' sig piTes liTs Tem Tcm piTes' liTs' Tem' Tcm'.
(prg iG) ⊢ C' ≼C C ⟶
  (clevel iG C' ⊑ clevel iG C) ∧
  (mlevel_ass iG C  sig = (piTes, liTs, Tem, Tcm) ⟶
   mlevel_ass iG C' sig = (piTes', liTs', Tem', Tcm')
   ⟶ Tem' ⊑ Tem ∧ Tcm ⊑ Tcm' ∧  list_all2 (op ⊑) piTes piTes')
```

It may seem that it would be more convenient to have contravariance with respect to class levels, i.e. `clevel iG C` $\sqsubseteq$ `clevel iG C'` in the above definition. Unfortunately, this would lead to unsoundness, for example during method call, because a "high" level object could then be dynamically bound to a variable of "low" type, thus violating an essential invariant established in Section 5.

However, the above restriction is less severe than it appears at first glance: Fields can be assigned IF types independently of the classes they are declared for, and nothing prevents fields in subclasses from having a higher security level than fields in superclasses.

The second important well-formedness condition, `wf_i_java_mdecl`, relates the declared types of a method to its actual types. It expresses, among others, that in a method declaration,

- the declared expression type `Tem` has to overestimate the actual expression type `Temr` of the result: `Temr` $\sqsubseteq$ `Tem`
- the declared effect type `Tcm` has to underestimate the total effect of the method

These conditions are tied together in the predicate `wf_i_java_prog`, which defines a program to be well-formed wrt. IF types if all classes in the program are well-formed and all classes have well-formed method declarations.

4 Equivalence of States

The noninterference result to be proved in Section 6 says that observational equivalence is preserved during execution. This section is devoted to a discussion of the notion of equivalence of states used in the following.

For an observer having access to data up to a certain security level `iT`, two states are equivalent if the data he can see agree in both states, while data of higher security levels are possibly different. Components of the state that can be inspected are exceptions, the heap and local variables. Thus, we define equivalence of two states (relative to an environment and an IF type `iT`) by

```
state_eq_below  ::
 [('c, 'l::complete_lattice) ienv, 'l, xstate, xstate]
                                    ⇒ bool (_,_ ⊢ _ ~ _)
E,iT |- s1 ~ s2 ==
(abrupt s1 = abrupt s2) ∧
(heq_below (prg E) iT (heap (store s1)) (heap (store s2))) ∧
(lvars_eq_below (localT E) iT (locals (store s1)) (locals (store s2)))
```

This expresses that the exception components (`abrupt`) of the states have to be equal (exceptions will henceforth mostly be neglected, see Section 6), and heaps and local variables have to be equivalent.

Local variables, represented by two value assignments `l1` and `l2`, are equivalent below an IF type `iT` (relative to an IF type assignment `liT`) if they agree on all variables of type `iT` or below.

```
lvars_eq_below ::
   [(vname ⇀ ('l::complete_lattice)), 'l,
    (vname ⇀ val), (vname ⇀ val)]  ⇒ bool
lvars_eq_below liT iT l1 l2 ==
            (∀ vn Tv. liT vn = Some Tv ⟶ Tv ⊑ iT ⟶ l1 vn = l2 vn)
```

Equivalence of heaps is more complex. In analogy to local variables, we first define what it means for two fields to be equivalent below a type *iT*:

```
feq_below :: [('c, 'l::complete_lattice) iprog, 'l,
      (vname × cname ⇀ val), (vname × cname ⇀ val)] ⇒ bool
feq_below G iT f1 f2 ==
          (∀  vn cn. flevel G cn vn ⊑ iT ⟶ f1 (vn, cn) = f2 (vn, cn))
```

Now, two heaps *h1* and *h2* are equivalent below *iT* if for all locations designating objects *obj1* resp. *obj2* with types below *iT*, the class names (*fst* component) agree and the fields (*snd* component) are equivalent:

```
heq_below :: [('c, 'l::complete_lattice) iprog, 'l, aheap, aheap] ⇒ bool
heq_below G iT h1 h2 == ((hloc_below G h1 iT) = (hloc_below G h2 iT)) ∧
     (∀  loc ∈ (hloc_below G h1 iT). ∀  obj1 obj2.
         (h1 loc = Some obj1) ⟶ (h2 loc = Some obj2) ⟶
         ((fst obj1) = (fst obj2)) ∧ feq_below G iT (snd obj1) (snd obj2))
```

This definition expresses an even stronger requirement: The heaps have to agree on their locations below type *iT*, which means that the set of locations referencing objects below *iT*:

```
hloc_below :: [('c, 'l::complete_lattice) iprog, aheap, 'l] ⇒ loc set
hloc_below G h iT == {l. ∃obj. h l = Some obj ∧ clevel G (fst obj) ⊑ iT}
```

are the same. In particular, it should not be possible for an observer of type *iT* to have access to a location (e.g. store its reference) in one heap and be denied access in the other heap, because that would make the heaps distinguishable. Under this condition, it is clear why *heq_below* does not care about class name and field equivalence of objects of higher type than *iT*: These are not accessible to an observer of type *iT* anyway.

Our heap model is quite abstract: The *new_Addr* operator does not allocate memory of a specific size, depending on the class of the object to be created, and the heap size does not have a specific bound, so our model cannot keep track of information leaks due to memory consumption.

The notion of equivalence of heaps is simplistic in yet another sense: We assume that there is equality of all locations referring to objects below type *iT*. A realistic memory allocator will easily invalidate this assumption. Therefore, it would have been more appropriate to define heap equivalence with respect to a bijection that maps "low" addresses in *h1* to "low" addresses in *h2*. Then, we would have been obliged to interpret equivalence of values (in particular addresses) modulo this bijection, which would have complicated the definitions of *lvars_eq_below* and *feq_below* considerably, and possibly also proofs, which so far exploit Isabelle's strength in dealing with standard equality. We have therefore given preference to the model described above.

5 IF Type Soundness

As a first step to the noninterference result of Section 6, we will show that evaluation resp. execution preserves a certain invariant. Because of its similarity to type soundness in the standard type system, this invariance property can also be understood as type soundness of the IF type system. In Section 5.1, we will spell out the definition of the invariant in detail, then state the type soundness theorem and discuss its proof in Section 5.2.

5.1 IF Conformance and Invariance

We introduce two concepts:

- *IF conformance* describes that runtime values (in local variables and the heap) are in accordance with the statically predicted types. This notion is the IF counterpart of conformance of the standard type system, which is used in the type soundness proof mentioned in Section 2.3. At the same time, it is related to the concept of "simple security" as used by [VSI96], because it expresses that the values read during execution are "below" a given type.
- *Invariance* says that at most the contents of locations above a certain IF type is modified and otherwise remains invariant. This is related to the notion of "confinement" in that it says that only locations above a certain type are written to.

We have chosen to formalize evaluation in the form of a big-step semantics, which in some respects facilitates proofs, but has the disadvantage that some properties can only be expressed indirectly. For example, invariance as defined below literally only says that a location has the same value before and after evaluation, but not that there are no modifications in intermediate states. For this, a small-step semantics would be better suited.

IF conformance is defined gradually, first for values, then for objects and heaps, finally for an entire state.

A value *v* conforms to type *iT* in heap *h* and for program *G* if, whenever *v* has a class type (in the standard type system), then the class level is below *iT*:

```
i_conf :: [('c,'l::complete_lattice) iprog, aheap, val, 'l] => bool
            (_,_ ⊢ᵢ _ ::≼ _)
  G,h ⊢ᵢ v::≼ iT == ∃T'. typeof (option_map obj_ty o h) v = Some T' ∧
                          (∀ C. T' = (Class C) ⟶ clevel G C ⊑ iT)
```

This concept is then extended to value assignments:

```
  G,h ⊢ᵢ vs[::≼]Ts == ∀n iT. Ts n = Some iT
                     ⟶ (∃v. vs n = Some v ∧ G,h ⊢ᵢ v::≼ iT)
```

and to objects:

```
  G,h ⊢ᵢ obj √ ==
     G,h ⊢ᵢ snd obj[::≼]
         map_of (map (λ ((vn,cn), T). ((vn,cn), flevel G cn vn))
                    (fields (G,fst obj)))
```

expressing that for each field, the value conforms to the field's `flevel`.

Finally, a heap is conformant if all its objects are, and a state is if both its heap and its local variable components are:

```
i_hconf :: [('c,'l::complete_lattice) iprog, aheap] => bool (_ ⊢ᵢh _ √)
G ⊢ᵢh h √   == ∀a obj. h a = Some obj ⟶ G,h ⊢ᵢ obj √

i_conforms :: [xstate, ('c,'l::complete_lattice) ienv] => bool (_ ::≼ᵢ _)
s::≼ᵢ E == prg E ⊢ᵢh heap (store s) √ ∧
           prg E,heap (store s)⊢ᵢlocals (store s)[::≼]localT E
```

For these concepts, we can prove properties reminiscent of those that hold for the standard type system, for example that object conformance is preserved when extending the heap:

```
lemma i_oconf_hext: G,h ⊢ᵢ obj√ ==> h≤|h' ==> G,h' ⊢ᵢ obj√
```

and that conformance is monotone in its IF type component:

```
lemma i_conf_widen:  G,h ⊢ᵢ x::≼T ⟶ T ⊑ T' ⟶ G,h ⊢ᵢ x::≼T'
```

Invariance is introduced with the aid of the concepts developed in Section 4. We define invariance for local variables, saying that the contents of local variables is unmodified for all types not above a type `Tc`, and similarly for heaps:

```
lvars_eq_not_above liT Tc l1 l2 ==
        (∀ Ti. ¬ Tc ⊑ Ti ⟶ lvars_eq_below liT Ti l1 l2)

heq_not_above G Tc h h' == (∀ Ti. ¬ Tc ⊑ Ti ⟶ heq_below G Ti h h')
```

Now, invariance is defined as:

```
invariant_not_above G liT iT h l h' l' ==
          heq_not_above G iT h h' ∧ lvars_eq_not_above liT iT l l'
```

5.2 IF Type Preservation

We can now state the extended type soundness theorem. When regarding expression evaluation, we have the following assumptions:

- the program under consideration is well-formed for the standard and the IF type system.
- evaluation of an expression `e` starts in a state `(x,(h,l))`, ends in a state `(x', (h',l'))` and produces result value `v` (here, `x` is the exception, `h` the heap and `l` the local variable component).
- the start state is conformant both wrt. the standard and the IF type system.
- the expression is well-typed wrt. the standard type system and has type `(Te, Tc)` in the IF type system.

Then, we can conclude that

- the result state is conformant wrt. the IF type system.

– the result value conforms to IF type *Te*, provided that evaluation produces no exception.
– locations not above type *Tc* are not modified.

Altogether, the theorem for expression evaluation reads:

Theorem 1. *theorem eval_itype_sound:*

```
⟦ wf_java_prog (prg G); wf_i_java_prog G ⟧ ⟹
  (prg G⊢(x,(h,l)) -e  ≻v  -> (x', (h',l')) ⟶
    (∀lT.   (x,(h ,l ))::≼(prg G,lT) ⟶ (∀T . (prg G,lT)⊢e  :: T ⟶
    (∀ liT. (x,(h, l ))::≼_i (G,liT) ⟶ (∀ Te Tc. (G,liT) ⊢_i e:: (Te, Tc)

  ⟶ (x',(h',l'))::≼_i (G,liT) ∧
      (x'=None ⟶ G,h' ⊢_i v  ::≼ Te ) ∧
      (invariant_not_above G liT Tc h l h' l'))))))
```

The one for statements is similar, with the difference that no mention of expression types and result values is necessary.

We will not go into details of the proof (for these, see the Isabelle proof script), but only state some general principles and motivate some of the preconditions.

The proof is by mutual induction over the evaluation resp. execution relation. The conformance and well-formedness requirements of the standard type system are necessary to ensure that we always access defined components. Without this, method lookup, among others, might yield an arbitrary method whose body, on execution, would not behave as specified by the method's well-formedness conditions. (We had to impose similar requirements when verifying a compiler for MicroJava [Str02], so they did not come as a surprise). On the downside, this means that we have to propagate these assumptions through the proof by appealing to the soundness theorem and typing rules of the standard type system. Apart from that, the proof is standard, but not fully automatic, because it requires the judicious application of lemmas about conformance and invariance such as those mentioned in Section 5.1.

It is worth noting that only part of the preconditions of the typing rules are required for type soundness. Among the conditions that become relevant only for noninterference is the constraint $Te \sqcup Teps \sqsubseteq Tcm$ for method call.

6 Noninterference

Noninterference is established by means of a bisimulation proof which shows that equivalence of states (in the sense of Section 4) is preserved during execution.

Before starting out with definitions and a statement of the theorem, we want to make clear that the noninterference result does not comprise exceptions: We explicitly assume exceptions not to occur during execution. We even assume that in the presence of exceptions, our result does not hold – differently said, the typing rules would have to be strengthened.

In order to keep proofs manageable, we introduce a few abbreviations, among them a common predicate for the two notions of conformance:

```
conf_state :: ('c,'l) iprog ⇒ lenv ⇒ 'l ilenv ⇒ xstate ⇒ bool
conf_state G lT liT s ==   s ::≼(prg G,lT) ∧ s ::≼ᵢ (G,liT)
```

Next, we define predicates expressing part of the bisimulation property. The one for expression evaluation is:

```
bisim_eval G x1 h1 l1 e v1 x1' h1' l1' ==
  (∀ x2 h2 l2 x2' h2' l2' v2.
   prg G⊢(x2, h2, l2) -e  ≻v2  -> (x2', h2', l2') ⟶
    x1' = None ⟶
    x2' = None ⟶
    (∀ lT liT T Te Tc Ti.
           (prg G,lT)⊢e  :: T  ⟶ (G,liT) ⊢ᵢ e :: (Te, Tc) ⟶
           conf_state G lT liT (x1, h1, l1) ⟶
           conf_state G lT liT (x2, h2, l2) ⟶
           (G, liT), Ti ⊢ (x1, h1, l1) ~ (x2, h2, l2)
        ⟶ (G, liT), Ti  ⊢ (x1', h1',l1') ~ (x2', h2', l2') ∧
           (Te ⊑ Ti  ⟶  (v1 = v2))))
```

We have similar predicates `bisim_evals` and `bisim_exec` for expression list evaluation and execution of statements.

Let us discuss these definitions in the context of the noninterference theorem:

Theorem 2. *theorem non_interference:*

```
⟦wf_java_prog (prg G); wf_i_java_prog G; equal_allocator G new_Addr ⟧ ⟹
  (prg G⊢(x1, h1, l1) -e  ≻v1  -> (x1', h1',l1') ⟶
  bisim_eval G x1 h1 l1 e v1 x1' h1' l1') ∧

  (prg G⊢(x1, h1, l1) -es [≻] vs1  -> (x1', h1',l1') ⟶
  bisim_evals G x1 h1 l1 es vs1 x1' h1' l1') ∧

  (prg G⊢(x1, h1, l1) - c -> (x1', h1',l1') ⟶
  bisim_exec G x1 h1 l1 c x1' h1' l1')
```

Apart from the preconditions in the first line and looking at expressions, we assume that we have one evaluation sequence of expression `e`, starting in state `(x1, h1, l1)` and yielding value `v1`, and that there is a second evaluation sequence of `e`, starting in `(x2, h2, l2)` and yielding value `v2` (the latter is in the definition of `bisim_eval`). Furthermore, we assume that the result states are exception-free, that the expression has standard type `T`, IF expression and effect types `(Te, Tc)`, and that the start states conform. If the two start states are equivalent below an arbitrary type `Ti`, then so are the result states. Furthermore, if `e` has a type lower than `Ti`, then the result values of both executions are the same.

Informally, this means that if an observer having access to data at most `Ti` cannot tell two start states apart, then also the result states appear equivalent. The equality of result values for `Te ⊑ Ti` is intuitively clear, given that `e` only contains information of level at most `Te`.

The proof is by induction on the evaluation relation, which essentially leaves us with one subgoal for each term constructor. We cannot go into details and refer to the Isabelle sources and [Str03] for excerpts of the proof script.

7 Conclusions

In this paper, we have presented an information flow analysis in the form of a type-and-effect system which is applicable to a significant subset of Java, and have proved a noninterference result with the proof assistant Isabelle.

Typing expressions and statements requires solving subtype constraints in the lattice of security levels. Due to the specific format of the constraints, this can be achieved with a polynomial-time algorithm, which has been proved sound and complete. Using Isabelle's code extraction facility [Ber03], an ML implementation of the whole type analysis framework has been derived, with which we are currently experimenting on simple programs.

We envisage integrating some more language features occurring often in practice, notably arrays and exceptions, and then investigating the usefulness of the analysis on programs taken from an industrial context.

Our model is still quite coarse, in several respects: We have mentioned the heap model (Section 4) which does not permit to characterise information leakage through memory consumption. In our big-step operational semantics, we cannot directly express timing leaks, not even leaks due to nontermination. Even for the fragment we consider, the analysis is presumably not very precise. A flow sensitive analysis, as suggested for an Algol-like language in [CHH02], is attractive, but some work is still required to make it take heap effects into account.

It is worth investigating whether a language with side-effecting expressions can simply be "desugared" by assigning to auxiliary variables, or whether the resulting effects on the state necessarily entail a less precise analysis.

We are not aware of another tool-supported formalisation of an information-flow type system. Even though it is time-consuming to set up, it hopefully pays off when we start experimenting with alternatives and extensions.

Acknowledgements

Thanks to Gerwin Klein, Marko Luther, Tobias Nipkow, Norbert Schirmer and Martin Wildmoser for discussions about this work. David Naumann and Anindya Banerjee have provided valuable comments on the extended version of this paper.

References

[Ber03] Stefan Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, TU München, 2003.

[BN02] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–267, June 2002.

[BN03] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *Proceedings of the Sixteenth IEEE Computer Security Foundations Workshop (CSFW)*, 2003.

[BS99] G. Barthe and B. Serpette. Partial evaluation and non-interference of object calculi. In A. Middeldorp and T. Sato, editors, *Proceedings of FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67, 1999.

[CHH02] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Journal of Computer Languages*, 28(1), 2002.

[Eur] ERCIM / Eurosmart. RESET project – draft roadmap. Future challenges and roadmaps for RTD on secure devices and platforms, http://www.ercim.org/reset/results.html.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–377, January 1998. http://cm.bell-labs.com/cm/cs/who/nch/slam.ps.

[Mye99] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, January 1999.

[NOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[Str02] Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

[Str03] Martin Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[VI97] Dennis Volpano and Cynthia Irvine. Secure flow typing. *Computers and Security*, 16(2), 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.