Formal Analysis of an Information Flow Type System for MicroJava

Martin Strecker
Technische Universität München,
Fakultät für Informatik,
D-85748 Garching

July 25, 2003

Abstract

This report 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 formalised 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] provide an analysis for essentially the Featherweight [IPW01] subset of Java. A major advance of this approach is that it can deal with heap-allocated objects. In this language, expressions are in general side effect free, but there is a subtle interaction between values and effects in method lookup, and it is not quite clear how this is handled in [BN02] (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 lead 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). In order to accommodate side-effecting expressions and dynamic method lookup, we introduce a type-and-effect system: An expression is assigned two IF types, 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, BN02], 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 – almost a necessity in view of the intricate interplay of standard and IF type system. The paper concludes with a discussion of future extensions (Section 7).

This report is an extended version of the paper [Str03], covering at greater depth the definition of the type system, well-formedness conditions and presenting part of the noninterference proof. Even though, it still does not cover the full formalisation. For all details, please refer to the Isabelle sources at `http://www4.in.tum.de/~streckem/InfoFlow/`

2 Formalisation of the Standard Language

In this section, we give an overview of Isabelle and describe the existing formalisations of Java in Isabelle, concentrating on the source language, MicroJava (the Java virtual machine language is not relevant here). This reduced version of Java [NOP00] accommodates essential aspects of Java, like classes, subtyping, object creation, inheritance, dynamic binding and exceptions, but 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.

2.1 A Short Introduction to Isabelle

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 syntax is reminiscent of ML, so we will only mention a few peculiarities: Consing an element `x` to a list `xs` is written as `x#xs`. Infix `@` is the append operator, `xs ! n` selects the `n`-th element from list `xs`.

We have the usual type constructors `T1 × T2` for product and `T1 ⇒ T2` for function space. The long arrow ⟹ is Isabelle's meta-implication, in the following mostly used in conjunction with rules of the form ⟦`P1; ...; Pn`⟧ ⟹ `C` to express that `C` follows from the premises `P1 ... Pn`. Apart from that, there is the implication ⟶ of the HOL object logic, along with the standard connectives and quantifiers.

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 ⇀ T2`, which just abbreviates `T1 ⇒ (T2 option)`.

The constructor `Some` has a left inverse, the function `the :: 'a option ⇒ 'a`, defined by the sole equation `the (Some x) = x`. This function is total in the sense that also `the None` is a legal, but indefinite value. In a similar way, function `the_Addr` is the inverse of the constructor `Addr` which injects addresses into values, and it is indefinite if applied to a non-address value. Another frequently used term describing an indefinite value is the polymorphic `arbitrary`.

Ultimately, indefinite values are defined with Hilbert's ϵ operator. They denote a fixed, but otherwise unknown value of their respective type.[1] In particular, they cannot be shown to be equal to any specific value of the type. Thus, we cannot prove an equation like `f arbitrary = arbitrary`. Because they are difficult to handle, we will always strive to obtain definite values, for example by imposing side conditions as those mentioned in Section 6.2.

2.2 Term and Class Structure

We will first present the MicroJava term structure (expressions and statements), then describe values and types and then gradually build up more complex structures, such as classes and entire programs.

The definition of expressions and statements is based on Isabelle types `cname` (class name), `vname` (variable name) and `mname` (method name):

```
datatype expr
  = NewC cname               -- class instance creation
  | Cast cname expr          -- type cast
  | Lit val                  -- literal value, also references
  | BinOp binop expr expr    -- binary operation
  | LAcc vname               -- local (incl. parameter) access
  | LAss vname expr          (_::=_ )      -- local assign
  | FAcc cname expr vname    ({_}_..._ ) -- field access
```

[1] Since types in HOL are guaranteed to be non-empty, such an element always exists.

```
  | FAss cname expr vname expr    ({_}_._:=_ ) -- field ass.
  | Call cname expr mname
    (ty list) (expr list)         ({_}_._( {_}_) ) -- method call

datatype stmt
  = Skip                     -- empty statement
  | Expr expr                -- expression statement
  | Comp stmt stmt           (_;; _           )
  | Cond expr stmt stmt      (If (_) _ Else _ )
  | Loop expr stmt           (While (_) _     )
```

The annotations in parentheses define concrete syntax translations; thus, local variable assignment can be written as `v::=e`. 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 and reference types:

```
datatype prim_ty
  = Void          -- 'result type' of void methods
  | Boolean
  | Integer

datatype ref_ty
  = NullT         -- null type
  | ClassT cname  -- class type

datatype ty = PrimT prim_ty   |   RefT  ref_ty
```

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
```

From a class declaration, we can derive an "immediate subclass" relation. The "subclass" relation, its reflexive-transitive closure, is written as `G⊢C ≺C D`.

2.3 Standard Type System

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

- `E ⊢ e :: T` means that expression `e` has type `T` in environment `E`.
- `E ⊢ c √` 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 typical rule is the one for the conditionals. A conditional is well-typed if the condition `e` is a Boolean and the two branches are well-typed:

```
Cond: ⟦ E⊢e::PrimT Boolean;   E⊢s1√;   E⊢s2√ ⟧
   ⟹  E⊢If(e) s1 Else s2√
```

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.

2.4 Operational Semantics

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

- `G ⊢ s -e≻ 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 ⊢ s -c-> s'` says that for program `G`, execution of statement `c` in state `s` yields a new state `s'`.

Here, values are:

```
datatype 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.

```
types aheap  = loc ⇀ obj
      locals = vname ⇀ val
      state  = aheap × locals
      xstate = val option × state
```

For example, the rule for field assignment is as follows (also see the proof in Section 6.2):

```
FAss: ⟦ G⊢  Norm s0  -e1≻a'-> (x1,s1); a = the_Addr a';
        G⊢ (np a' x1,s1) -e2≻v -> (x2,s2);
        h  = heap s2; (c,fs) = the (h a);
        h' = h(a↦(c,(fs((fn,T)↦v)))) ⟧ ⟹
     G⊢Norm s0 -{T}e1..fn:=e2≻v-> c_hupd h' (x2,s2)
```

Here, `Norm` means that the exception component is `None`. Function `np` conditionally raises a null-pointer exception if its first argument is null, otherwise returns its second argument. `c_hupd` conditionally updates a state with the given heap, provided there is no exception, and otherwise leaves the state unchanged.

The operational semantics and the standard type system are related by a type soundness result. It expresses that a "conformance relation" is preserved during evaluation (for details, see [Ohe01]). In Section 5.2, we prove a similar theorem for the IF type system, using a different conformance relation.

3 IF Typing

We are now prepared to define the IF type system, which is built on top of the existing MicroJava formalisation of Section 2. It would have been possible to integrate IF types into the standard typing judgements of Section 2.3. 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 A Lattice of Security Types

For simplicity, we introduce 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
```

Since we only use abstract lattice properties in our proofs, we could, with some notational overhead, parameterise the whole development over an axiomatisation of lattices. Instead, we simply define supremum, infimum and less-or-equal by

```
consts
  sup :: [ity, ity] ⇒ ity  (infixr ⊔)
  inf :: [ity, ity] ⇒ ity  (infixr ⊓)
  ile :: [ity, ity] ⇒ bool (infixr ⊑)

primrec
  Lw ⊔ t = t
  Hg ⊔ t = Hg
```

```
primrec
  Lw ⊓ t = Lw
  Hg ⊓ t = t

primrec
  Lw ⊑ t = True
  Hg ⊑ t = (t = Hg)
```

and then derive the expected equations, such as:

```
lemma ile_inf: z ⊑ x ⊓ y = (z ⊑ x ∧ z ⊑ y)

lemma sup_ile: x ⊔ y  ⊑ z = (x ⊑ z ∧ y ⊑ z)

lemma ile_trans: x ⊑ y ⟹ y ⊑ z ⟹ x ⊑ z
```

We also need the strict version of $\sqsubseteq$ and functions for taking supremum and infimum of lists:

```
constdefs
  iless :: [ity, ity] ⇒ bool (infixr ⊏)
  x ⊏ y == (x ⊑ y) ∧ (x ≠ y)

  inf_list :: ity list ⇒ ity
  inf_list l == foldr inf l Hg
  sup_list :: ity list ⇒ ity
  sup_list l == foldr sup l Lw
```

3.2 Typing Rules

3.2.1 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⊢ᵢ e :: (Te,Tc)` says that expression `e` has expression type `Te` and effect type `Tc` (where both `Te` and `Tc` are an `ity`)
- `E⊢ᵢ c :: Tc` says that statement `c` has effect type `Tc` (where `Tc` is an `ity`)
- `E⊢ᵢ es[::]Ts` says that the expressions in list `es` have the expression and effect types in `Ts` (where `Ts` is a `(ity × ity) list`)

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:

```
types
  ilenv   = vname ⇀ ity
  'c ienv = 'c prog × ilenv
```

The functions `prg` resp. `localT` project out the first resp. second component.

The informal meaning of `E⊢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`.

Example 1 *For example, 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.*

3.2.2 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

```
consts
  clevel :: ['c prog, cname] ⇒ ity
  flevel :: ['c prog, cname, vname] ⇒ ity
  mlevel_ass :: ['c prog, cname, sig] ⇒ mlevel_tp
```

which 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`:

```
constdefs
  mlevel :: ['c prog, cname] ⇒ sig ⇀ mlevel_tp
  mlevel G C sig ==
    case method (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

```
types
  mlevel_tp = (ity list) × (ity list) × ity × ity
```

3.2.3 Rules

The typing rules for expressions and statements are shown in Figure 1 resp. Figure 2.

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 ⊔ Te2 ⊑ Te'` and `Tc'`

```
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 []e[::]e[]

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

Figure 1: IF typing rules for expressions and expression lists

```
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'
```

Figure 2: IF typing rules for statements

$\sqsubseteq$ *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. 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.

After the motivation of the typing judgements in Section 3.2.1, 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. It is a bit surprising that there does not seem to be a similar condition in the system [BN02], which also has to cope with dynamic method lookup.

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). Apparently, the rule in [VSI96] can be justified by another argument and therefore does not introduce an unsoundness; however, this generality is not even required, since we are interested in the most specific type, which is the highest possible effect type.

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
  - contravariant parameter types

This is summarised in the definition:

```
constdefs
  wf_i_cdecl   :: 'c prog => cname => bool
  wf_i_cdecl G C == ∀ C' sig piTes liTs Tem Tcm piTes' liTs' Tem' Tcm'.
  G ⊢ C' ≼C C ⟶
    (clevel G C' ⊑ clevel G C) ∧
    (mlevel_ass G C  sig = (piTes, liTs, Tem, Tcm) ⟶
     mlevel_ass G C' sig = (piTes', liTs', Tem', Tcm')
     ⟶ Tem' ⊑ Tem ∧ Tcm ⊑ Tcm' ∧  list_all2 (op ⊑) piTes piTes')
```

The second important well-formedness condition relates the declared types of a method to its actual types:

```
constdefs
 wf_i_java_mdecl :: 'c prog => cname => java_mb mdecl => bool
 wf_i_java_mdecl G C == λ((mn,pTs),rT,(pns,lvars,blk,res)).
  case (mlevel G C (mn, pTs)) of
    None ⇒ False
  | Some (piTs, liTs, Tem, Tcm) ⇒
    (let  E = (G, empty ((map fst lvars)[↦]liTs)(pns[↦]piTs)(This↦clevel G C))
     in (length piTs = length pns ∧
         length liTs = length lvars ∧
         (∃ Tcmb Temr Tcmr.
             E ⊢ᵢ blk :: Tcmb ∧
             E ⊢ᵢ res::(Temr, Tcmr) ∧
             Temr ⊑ Tem ∧ Tcm ⊑ Tcmb ⊓ Tcmr)))
```

It expresses that in a method declaration,

- the lengths of parameter type list `piTs` and parameter list `pns` have to be the same
- similarly for local variable type list `liTs` and local variable list `lvars`
- the declared expression type `Tem` has to overestimate the actual expression type `Temr` of the result
- 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.

```
constdefs
 wf_i_prog :: 'c wf_mb => 'c prog => bool
 wf_i_prog wf_mb G ==
     (∀ (C, cd) ∈ set G. (wf_i_cdecl G C))
  ∧ (∀c∈ set G. (wf_cdecl_mdecl wf_mb G c))

translations
  wf_i_java_prog == wf_i_prog wf_i_java_mdecl
```

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

```
constdefs
  state_eq_below :: 'c ienv ⇒ ity ⇒ 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.

```
constdefs
  lvars_eq_below ::  [(vname ⇝ ity), ity, (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`:

```
constdefs
  feq_below :: ['c prog, ity, (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:

```
constdefs
  heq_below :: 'c prog ⇒ ity ⇒ 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`:

```
constdefs
  hloc_below :: 'c prog ⇒ aheap ⇒ ity ⇒ 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 (see the discussion in Section 6.2). 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 (also see Section 7).

The relations introduced above enjoy the properties one would expect of equivalence relations. For equivalence of states, we have among others:

```
lemma state_eq_below_refl: (G, liT),Te ⊢ s1 ~ s1
lemma state_eq_below_sym: (G, liT),Te ⊢ s1 ~ s2 ⟹ (G, liT),Te ⊢ s2 ~ s1
lemma state_eq_below_trans:
  ⟦ (G, liT),Te ⊢ s1 ~ s2; (G, liT),Te ⊢ s2 ~ s3 ⟧ ⟹(G, liT),Te ⊢ s1 ~ s3
```

These results are based on similar properties of the defining relations like *heq_below*.

Furthermore, it is easy to derive some monotonicity resp. antimonotonicity properties, for example:

```
lemma hloc_below_mono: Tc' ⊑ Tc  ⟹ hloc_below G h Tc' ⊆ hloc_below G h Tc
lemma feq_below_antimono:
  ⟦ feq_below G Tc f f'; Tc' ⊑ Tc ⟧  ⟹ feq_below G Tc' f f'
lemma state_eq_below_antimono:
  ⟦ (G, liT),Tc ⊢ s1 ~ s2; Tc' ⊑ Tc ⟧  ⟹ (G, liT),Tc' ⊢ s1 ~ s2
```

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, as described in Section 2.4. 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 formalise 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`:

```
constdefs
  i_conf :: 'c prog => aheap => val => ity => 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:

```
  i_lconf :: 'c prog => aheap => ('a ⇀ val) => ('a ⇀ ity) => bool
                                    (_,_ ⊢_i _ [::≼] _)
  G,h ⊢_i vs[::≼]Ts == ∀n iT. Ts n = Some iT
                      ⟶ (∃v. vs n = Some v ∧ G,h ⊢_i v::≼ iT)
```

and to objects:

```
  i_oconf :: 'c prog => aheap => obj => bool         (_,_ ⊢_i _ √)
  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 prog => aheap => bool                (_ ⊢_ih _ √)
  G ⊢_ih h √   == ∀a obj. h a = Some obj ⟶ G,h ⊢_i obj √
```

```
i_conforms :: xstate => 'c 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:

```
constdefs
  lvars_eq_not_above ::  [(vname ⇝ ity), ity, (vname ⇝ val), (vname ⇝ val)]
                         ⇒ bool
  lvars_eq_not_above liT Tc l1 l2 ==
           (∀ Ti. ¬ Tc ⊑ Ti ⟶ lvars_eq_below liT Ti l1 l2)

  heq_not_above :: ['c prog, ity, aheap, aheap] ⇒ bool
  heq_not_above G Tc h h' == (∀ Ti. ¬ Tc ⊑ Ti ⟶ heq_below G Ti h h')
```

Now, invariance is defined as:

```
  invariant_not_above :: ['c prog, (vname ⇝ ity), ity,
                          aheap, (vname ⇝ val), aheap, (vname ⇝ val)] ⇒ bool
  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 G; wf_i_java_prog G ⟧ ⟹
  (G⊢(x,(h,l)) -e ≻v -> (x', (h',l')) ⟶
      (∀ lT.   (x,(h ,l ))::≼(G,lT) ⟶ (∀ T . (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:

Theorem 2 *theorem exec_itype_sound:*

```
⟦ wf_java_prog G; wf_i_java_prog G ⟧ ⟹
  (G⊢(x,(h,l)) -c -> (x', (h',l')) ⟶
      (∀ lT.   (x,(h ,l ))::≼(G,lT) ⟶ (G,lT)⊢c √ ⟶
      (∀ liT. (x,(h, l ))::≼ᵢ (G,liT) ⟶ (∀ Tc. (G,liT) ⊢ᵢ c:: Tc
  ⟶ (x',(h',l'))::≼ᵢ (G,liT) ∧
      (invariant_not_above G liT Tc h l h' l')))))
```

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.

6 Noninterference

6.1 Noninterference Theorem

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:

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

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

```
constdefs
  bisim_eval :: [java_mb prog, val option, aheap, State.locals, expr, val,
                 val option, aheap, State.locals] ⇒ bool

  bisim_eval G x1 h1 l1 e v1 x1' h1' l1' ==
  (∀ x2 h2 l2 x2' h2' l2' v2.
    G⊢(x2, h2, l2) -e ≻v2 -> (x2', h2', l2') ⟶
    x1' = None ⟶
    x2' = None ⟶
    (∀ lT liT T Te Tc Ti.
           (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 also predicates `bisim_evals` and `bisim_exec` for expression list evaluation and execution of statements; their definition is shown in Figure 3.

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

Theorem 3 *theorem non_interference:*

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

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

  (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` and IF expression and effect types `(Te,`

```
bisim_evals :: [java_mb prog, val option, aheap, State.locals,
                expr list, val list, val option, aheap, State.locals]
               ⇒ bool

  bisim_evals G x1 h1 l1 es vs1 x1' h1' l1' ==
  (∀ x2 h2 l2 x2' h2' l2' vs2.
    G⊢(x2, h2, l2) -es [≻] vs2 -> (x2', h2', l2') ⟶
    x1' = None ⟶
    x2' = None ⟶
    (∀ lT liT Ts iTs Ti.
              (G,lT)⊢es [::] Ts ⟶ (G,liT) ⊢_i es [::] iTs ⟶
              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') ∧
              (sup_list (map fst iTs) ⊑ Ti ⟶ vs1 = vs2))))

  bisim_exec :: [java_mb prog, val option, aheap, State.locals, stmt,
                 val option, aheap, State.locals] ⇒ bool

  bisim_exec G x1 h1 l1 c x1' h1' l1' ==
  (∀ x2 h2 l2 x2' h2' l2'. G⊢(x2, h2, l2) - c -> (x2', h2', l2') ⟶
    x1' = None ⟶
     x2' = None ⟶
    (∀ lT liT Tc Ti.  (G,lT)⊢c√ ⟶ (G,liT) ⊢_i c:: 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'))))
```

Figure 3: Definitions of `bisim_evals` and `bisim_exec`

`Tc)` and 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` $\sqsubseteq$ `Ti` is intuitively clear, given that `e` only contains information of level at most `Te`.

As mentioned in Section 4, our heap model and the notion of heap equivalence are quite abstract. Therefore, the allocation of a new object by the statement `NewC` has to be constrained so that for equivalent heaps, the function `new_Addr` always returns the same address:

```
equal_allocator :: ['c prog, (aheap => loc × val option)] ⇒ bool
equal_allocator G alloc ==
      ∀ h1 h2 a1 a2 iT.
      (a1, None) = alloc h1 ⟶ (a2, None) = alloc h2 ⟶
      (heq_below G iT h1 h2) ⟶  a1 = a2
```

Having to add this assumption to Theorem 3 is unsatisfactory and could be avoided by a more refined heap model, as suggested in Section 4.

6.2 Proof of Noninterference

The proof is by induction on the evaluation relation, which essentially leaves us with one subgoal for each term constructor. We will now take a closer look at the case of field update *{fT}e1..fn:=e2*, which is depicted graphically in Figure 4.

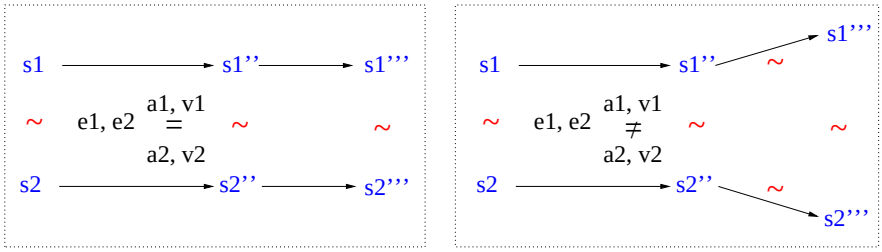

Figure 4: Bisimulation proof for field assignment

Informally, the proof can be summarised as follows: We first compute the states *s1''* resp. *s2''* resulting from evaluating the subexpressions *e1* and *e2*. If *s1 ~ s2* for the start states, then also *s1'' ~ s2''* by induction hypothesis. We now make a case distinction on *(Te1 ⊑ Ti ∧ Te2 ⊑ Ti)*. In the first case (left side of Figure 4), we can conclude by noninterference that the values of *e1* and *e2* (denoted by a_i and v_i) are equivalent under both evaluations, and so are the states *s1'''* resp. *s2'''* obtained when updating the heaps at addresses a_i with the values v_i.

Otherwise (right side of Figure 4), the addresses or values may be different. However, in this case, the typing rule *iFAss* permits to conclude that *¬ flevel G fT fn ⊑ Ti*, so the individual heap updates are equivalent (*s1'' ~ s1'''* and *s2'' ~ s2'''*), and therefore the resulting states are.

The formal Isabelle/Isar proof script is shown in the following (for an introduction to the Isar notation, see [Nip03]). Since the proof is by induction on the "first" evaluation relation, the "second" evaluation relation has to be unravelled, then the types of the subterms are determined by rule inversion, and some information is propagated (no exceptions occur in intermediate states; the intermediate states conform). Only after this bookkeeping can the proof itself begin, by showing equality of values (*eq_val*) and equivalence of result states (*st_eq_res*) – the latter by the above-mentioned case distinction.

```
lemma ni_FAss:
  assumes wfp: "wf_java_prog G"
  assumes wfip: "wf_i_java_prog G"
  assumes alloc: "equal_allocator G new_Addr"
  assumes ev1: "G⊢ Norm (h1, l1) -e1 ≻a1'-> (x1', h1', l1')"
  assumes ev1': "G⊢ (np a1' x1', h1', l1') -e2 ≻v1-> (None, h1'', l1'')"
  assumes bis_ev: "bisim_eval G None h1 l1 e1 a1' x1' h1' l1'"
  assumes bis_ev': "bisim_eval G (np a1' x1') h1' l1' e2 v1 None h1'' l1''"
  assumes a_eq: "a1 = the_Addr a1'"
```

```
  assumes c_fs1: "(c1, fs1) = the (h1'' a1)"
  assumes ev2_all: "G ⊢ (x2, h2, l2) -{fT}e1..fn:=e2≻v2-> Norm (h2''', l2''')"
  assumes stt: "(G, lT) ⊢ {fT}e1..fn:=e2 :: T"
  assumes ift: "(G, liT) ⊢ᵢ {fT}e1..fn:=e2 :: (Te, Tc)"
  assumes conf1: "conf_state G lT liT (Norm (h1, l1))"
  assumes conf2: "conf_state G lT liT (x2, h2, l2)"
  assumes st_eq: "(G, liT),Ti ⊢ Norm (h1, l1) ~ (x2, h2, l2)"
  shows "(G, liT),Ti ⊢ Norm (snd (c_hupd (h1''(the_Addr a1'↦(c1, fs1((fn, fT)↦v1))))
                                  (Norm (h1'', l1'')))) ~ Norm (h2''', l2''') ∧
         (Te ⊑ Ti ⟶ v1 = v2)"

proof -

  from ev2_all
  obtain a2' c2 fs2 h2' l2' h2'' l2'' x2' x2'' where
    ev2: "G ⊢ Norm (h2, l2) -e1≻a2'-> (x2', h2', l2')" and
    ev2': "G ⊢ (np a2' x2', h2', l2') -e2≻v2-> (x2'', h2'', l2'')" and
    c_fs2: "(c2, fs2) = the (h2'' (the_Addr a2'))" and
    x2n:   "x2 = None" and
    ch: "Norm (h2''', l2''') =
          c_hupd (h2'' (the_Addr a2'↦(c2, fs2((fn, fT)↦v2)))) (x2'', h2'', l2'')"
    by cases auto

  from stt
  obtain T' C where
    stt_e2: "(G, lT) ⊢ e2 :: T" and
    stt_e1: "(G, lT) ⊢ e1 :: Class C"
    apply cases
    apply auto
    apply (ind_cases "(G, lT) ⊢ {fT}e1..fn :: T'")
    apply auto
    done

  from ift
  obtain Tc1 Tc2 Te1 Te2 where
    ift_e1: "(G, liT) ⊢ᵢ e1 :: (Te1, Tc1)" and
    ift_e2: "(G, liT) ⊢ᵢ e2 :: (Te2, Tc2)" and
    cnstr_flev_e: "Te1 ⊔ Te2 ⊑ flevel G fT fn" and
    cnstr_te: "Te1 ⊔ Te2 ⊑ Te"
    by cases auto

  from ch
  have x2''n: "x2''=None"
    by (simp add: c_hupd_def split add: split_if_asm)

  from ev1'
```

```
have "np a1' x1' = None"
  by (fast intro: eval_no_xcpt)
hence x1'n: "x1' = None ∧ a1' ≠ Null"
  by (fast dest: np_NoneD)

from ev2' x2''n
have "np a2' x2' = None"
  by (simp, fast intro: eval_no_xcpt)
hence x2'n: "x2' = None ∧ a2' ≠ Null"
  by (fast dest: np_NoneD)

from bis_ev ev2 stt_e1 ift_e1 conf1 conf2 st_eq x1'n x2'n x2n
have st_eq': "(G, liT),Ti ⊢ Norm (h1', l1')
                          ~ Norm (h2', l2') ∧ (Te1 ⊑ Ti ⟶ a1' = a2')"
  by (simp only:) (rule joint_step_eval)

from wfp wfip conf1 stt_e1 ift_e1 ev1 x1'n
have
  conf1': "conf_state G lT liT (None,h1',l1') ∧
           G, h1' ⊢ a1' ::≼ Class C ∧ h1 ≤| h1'"
  by (auto dest: conf_state_eval_value_one)

from wfp wfip conf2 stt_e1 ift_e1 ev2 x2'n x2n
have
  conf2': "conf_state G lT liT (None,h2', l2') ∧
           G, h2' ⊢ a2' ::≼ Class C ∧ h2 ≤| h2'"
  by (auto dest: conf_state_eval_value_one)


from bis_ev' ev2' stt_e2 ift_e2 conf1' conf2' st_eq' x1'n x2'n x2n x2''n
have st_eq'': "(G, liT),Ti ⊢ Norm (h1'', l1'') ~ Norm (h2'', l2'') ∧
               (Te2 ⊑ Ti ⟶ v1 = v2)"
  apply (simp only:)
  apply (rule joint_step_eval)
  apply (simp (no_asm_simp) add: np_def)+
  done

from wfp wfip conf1' stt_e2 ift_e2 ev1' x1'n x2'n x2n x2''n
have
  conf1'': "conf_state G lT liT (None,h1'',l1'') ∧ h1' ≤| h1''"
  by (auto dest: conf_state_eval_value_one)


from wfp wfip conf2' stt_e2 ift_e2 ev2' x1'n x2'n x2n x2''n
have
  conf2'': "conf_state G lT liT (None,h2'', l2'') ∧ h2' ≤| h2''"
```

```
  by (auto dest: conf_state_eval_value_one)

from cnstr_te
have "Te2 ⊑ Te" by (simp add: sup_ile)
with st_eq''
have eq_val: "Te ⊑ Ti ⟶ v1 = v2"
  by (fast intro: ile_trans)

from conf1' conf1'' have conf_a1'_C: "G,h1'' ⊢ a1' ::≼ Class C"
  by (fast intro: conf_hext)
from conf2' conf2'' have conf_a2'_C: "G,h2'' ⊢ a2' ::≼ Class C"
  by (fast intro: conf_hext)

have st_eq_res:
     "(G, liT),Ti ⊢ Norm (h1''(the_Addr a1'↦(c1, fs1((fn, fT)↦v1))), l1'')
                  ~ Norm (h2''(the_Addr a2'↦(c2, fs2((fn, fT)↦v2))), l2'')"
proof (cases "(Te1 ⊑ Ti ∧ Te2 ⊑ Ti)")
  case True
  with st_eq'
  have "a1' = a2'" by simp
  moreover
  from True st_eq''
  have "v1 = v2" by simp
  ultimately
  show ?thesis
  using st_eq'' c_fs1 c_fs2 x2'n conf_a1'_C conf_a2'_C a_eq
    by (simp (no_asm_simp)) (fast intro: state_eq_h_upd_equal)

next
  case False
  with cnstr_flev_e
  have flevel_ti: "¬ flevel G fT fn ⊑ Ti"
    by (simp add: sup_ile) (fast intro: ile_trans)


  from flevel_ti c_fs1 conf_a1'_C x1'n a_eq
  have h1_upd:
       "(G, liT),Ti ⊢ Norm (h1'', l1'')
                    ~ Norm (h1''(the_Addr a1'↦(c1, fs1((fn, fT)↦v1))), l1'')"
    by (fast intro: state_eq_h_upd_diff)

  from flevel_ti c_fs2 conf_a2'_C x2'n
  have h2_upd:
       "(G, liT),Ti ⊢ Norm (h2'', l2'')
                    ~ Norm (h2''(the_Addr a2'↦(c2, fs2((fn, fT)↦v2))), l2'')"
    by (fast intro: state_eq_h_upd_diff)
```

```
    from st_eq'' h1_upd h2_upd show ?thesis by (fast intro: state_eq_below_diamond)
  qed

  from eq_val st_eq_res ch x2''n show ?thesis
    by (simp add: c_hupd_def)
qed
```

It is noteworthy that some of the preconditions of the typing rules are not required in the noninterference proof. For example, the condition involving the effect types (`Tc' ⊑ Tf ⊓ Tc1 ⊓ Tc2`) is not used above, but only in the type soundness proof.

Instrumental in this proof are theorems of a general nature, such as the following which shows that conformance is preserved during evaluation:

```
lemma conf_state_eval_value_one:
  ⟦ G⊢ s- e≻ v → s';
  conf_state G lT liT s;
  (G, lT) ⊢ e :: T; (G,liT) ⊢ᵢ e :: (Te, Tc);
  wf_java_prog G; wf_i_java_prog G ⟧
  ⟹ conf_state G lT liT s' ∧
  (abrupt s' = None ⟶ G,heap (store s') ⊢ v ::≼ T) ∧
  (heap (store s) ≤| heap (store s')) ∧
  (abrupt s' = None ⟶ G,heap (store s') ⊢ᵢ v ::≼ Te)
```

This kind of theorem is applied in almost every subcase of the inductive proof.

Also, we use several specific lemmas, for example the following one showing that state equality is preserved for an observer `Ti` when updating a field whose level is not within the "reach" of `Ti`. Side conditions such as `G,h ⊢ a' ::≼ Class C` ensure that value `a'` is a genuine address.

```
lemma state_eq_h_upd_diff:
 ⟦ ¬ flevel G T fn ⊑ Ti;
  (c, fs) = the (h (the_Addr a'));   G,h ⊢ a' ::≼ Class C';  a' ≠ Null⟧
  ⟹ (G, liT),Ti ⊢ Norm (h, l) ~ Norm (h(the_Addr a'↦(c, fs((fn, T)↦v))), l)
```

Moreover, updating fields of equivalent states at the same address `a'` with the same value `v` leads again to equivalent states:

```
lemma state_eq_h_upd_equal:
  ⟦ (G, liT),Ti ⊢ Norm (h1, l1) ~ Norm (h2, l2);
  (c1, fs1) = the (h1 (the_Addr a')); G,h1 ⊢ a' ::≼ Class C;
  (c2, fs2) = the (h2 (the_Addr a')); G,h2 ⊢ a' ::≼ Class C;
  a' ≠ Null ⟧
  ⟹ (G, liT),Ti ⊢ Norm (h1(the_Addr a'↦(c1, fs1((fn, T)↦v))), l1)
                  ~ Norm (h2(the_Addr a'↦(c2, fs2((fn, T)↦v))), l2)
```

7 Conclusions

In this report, 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.

So far, we have not evaluated the usefulness of the analysis in dealing with "real" programs, for two reasons: Firstly, some language features occurring often in practice, notably arrays or exceptions, are still missing in our model. Secondly, typing expressions and statements requires solving subtype constraints in the lattice of security levels. We are currently investigating whether it is possible to transform the type system into a deterministic one that permits simple type checking, or what a combination of type inference and constraint solving could look like. To preclude coding errors, we aim at deriving an executable type checker directly from the formalisation, using Isabelle's code extraction facility [Ber03].

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. One remedy is to make a distinction between "local" and "heap" effects, which is already present in [BN02]. 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.

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.

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 Sixteenth IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–267, June 2002.

[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.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

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

[Nip03] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer Verlag, 2003.

[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.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. http://www4.in.tum.de/~oheimb/diss/.

[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. In *???*, 2003. submitted for publication.

[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.