Extracting a data flow analyser in constructive logic

David Cachera, Thomas Jensen, David Pichardie, Vlad Rusu

Extracting a data flow analyser in constructive logic

David Cachera [a] Thomas Jensen [b] David Pichardie [a] Vlad Rusu [c]

[a] *IRISA / ENS Cachan (Bretagne), Campus de Beaulieu, 35042 Rennes France*

[b] *IRISA / CNRS, Campus de Beaulieu, 35042 Rennes France*

[c] *IRISA / INRIA, Campus de Beaulieu, 35042 Rennes France*

Abstract

A constraint-based data flow analysis is formalised in the specification language of the Coq proof assistant. This involves defining a dependent type of lattices together with a library of lattice functors for modular construction of complex abstract domains. Constraints are represented in a way that allows for both efficient constraint resolution and correctness proof of the analysis with respect to an operational semantics. The proof of existence of a solution to the constraints is constructive which means that the extraction mechanism of Coq provides a provably correct data flow analyser in Ocaml from the proof. The library of lattices and the representation of constraints are defined in an analysis-independent fashion that provides a basis for a generic framework for proving and extracting static analysers in Coq.

Key words: Program analysis ; constructive logic ; lattices ; theorem proving ; constraint solving.

1 Introduction

Static program analysis is a fully automatic technique for proving properties about the run-time behaviour of a program without actually executing it. The correctness of static analyses can be proved formally by following the theory of abstract interpretation [9] that provides a theory for relating two semantic interpretations of the same language. These strong semantic foundations constitute one of the arguments advanced in favor of static program analysis. The implementation of static analyses is usually based on well-understood constraint-solving techniques and iterative fixpoint algorithms. In spite of the nice mathematical theory of program analysis and the solid algorithmic techniques available one problematic issue persists, *viz.*, the *gap* between the analysis that is proved correct on paper and the analyser that actually runs on the

machine. While this gap might be small for toy languages, it becomes important when it comes to real-life languages for which the implementation and maintenance of program analysis tools become a software engineering task. To eliminate this gap, we here propose a technique based on theorem proving in constructive logic and the program-as-proofs paradigm. This allows to specify static analyses in a way that ensures their *well-formedness* and facilitates their *correctness proof*. Moreover, the constructive nature of the logic means that it is possible to extract, from the proof of existence of a correct program analysis result, a static analyser that maps any given program to their static analysis.

The development of the static analyser is done within the Coq proof assistant. Proofs in Coq are constructive and correspond, via the Curry-Howard isomorphism, to programs in a functional language with a rich type system. The program extraction mechanism in Coq provides a tool for automatic translation of these proofs into a functional language with a simpler type system, namely Ocaml. The extraction mechanism removes those parts of the proof that are only concerned with proving that the result satisfies its specification without actually contributing to its construction. In the case of our static analyser, the constructive part is concerned with calculating a solution to a system of constraints generated from the program. The other part of the proof establishes that a solution to the constraints is indeed a correct approximation of the program's behaviour but does not contribute to the actual construction of the solution.

The methodology that we present here is generic but we have chosen to develop it in the concrete setting of a flow analysis for Java Card byte code, presented in Section 2. The motivation for choosing this particular analysis is that it deals with a minimalistic, yet representative language with imperative, object-oriented and higher-order features, guaranteeing that the approach is transferable to a variety of other analyses. The methodology comprises two phases:

- the modular definition of a library of abstract domains of properties used in the analysis (Section 3). The abstract domains are lattices satisfying a finite-ascending-chains condition which makes it possible to extract a provably correct, generic constraint solver based on fixpoint iteration ;
- a representation of a constraint-based analysis that allows to extract an analyser from the proof of the existence of a best solution to the constraints, using the program extraction mechanism available in Coq (Section 4) and at the same time allows to prove correctness of the analysis (Section 5).

Section 6 compares with other work on formalizing the correctness of data flow analyses, and Section 7 concludes. Appendices A–E contain the formalisation of the analyser and is included as a service to those readers that want to see the details. However, the paper is written so as to be understandable without

having to read these appendices. The Coq sources of the development are available online [20].

Notation: Functions whose type depends on the program being analysed will have dependent type $F : (P : \text{Program}) \to T(P)$ with type T depending on P. We will write F_P for the application of F to a particular program P. The paper uses a mixture of logic and Coq notation. Java Card byte code and Coq functions are written using the teletype font (*e.g.*, `push` in Section 2 and `join` in Section 3). Lattices and abstract operations on these as well as their corresponding Coq types are written using the Roman font (*e.g.*, $\widehat{\text{push}}$ in Section 2 and the domain of abstract states $\widehat{\text{State}}$ illustrated in Figure 2).

A preliminary version of this article was presented at the European Symposium on Programming (ESOP) 2004 [5]. The present article is a thoroughly revised version that contains a more detailed description of the lattice library, the lattice constructors and the proof of well-foundedness of the lattices, a simplified representation of the constraints generated by the analysis and a more detailed presentation of the correctness proofs.

2 A Static Analysis for Carmel

The analysis which serves as a basis for our work is a data flow analysis for the Carmel intermediate representation of Java Card byte code [15] specified using the Flow Logic formalism [12] and proved correct on paper with respect to an operational semantics [22]. The language is a byte code for a stack-oriented machine, much like the Java Card byte code. Instructions include stack operations, numeric operations, conditionals, object creation and modification, and method invocation and return. It is given a small-step operational semantics with a state of the form $\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle$, where h is the heap of objects, and $\langle m, pc, l, s\rangle :: sf$ is a call stack consisting of *frames* of the form $\langle m, pc, l, s\rangle$ where each frame contains a method name m and a program point pc within m, a set of local variables l, and a local operand stack s (see [22] for details). Here and everywhere in the paper, "::" denotes the "cons" operation on lists.

The transition relation $\to_I$ describes how an instruction I affects the state. We give as example the rules defining the instructions `push` for pushing a value onto the operand stack, `invokevirtual` for calling a virtual method, and `return` for returning from a virtual method call. The expression $\text{instructionAt}_P(m, pc)$ denotes the instruction found at address (m, pc) in the program P.

The rule (1) reads as follows: the instruction `push` c at address (m, pc) of state $\sigma = \langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle$ has the effect of pushing c on the operand

stack s of σ and advancing to the instruction at $pc + 1$.

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{push}\ c}{\langle\langle h, \langle m, pc, l, s\rangle :: sf\rangle\rangle \rightarrow_{\texttt{push}\ c} \langle\langle h, \langle m, pc + 1, l, c :: s\rangle :: sf\rangle\rangle} \tag{1}$$

$$\frac{\begin{array}{c}\text{instructionAt}_P(m, pc) = \texttt{invokevirtual}\ M \\ h(loc) = o \qquad m' = lookUp(M, class(o)) \\ f' = \langle m', 1, V, \varepsilon\rangle \qquad f'' = \langle m, pc, l, s\rangle\end{array}}{\langle\langle h, \langle m, pc, l, loc :: V :: s\rangle :: sf\rangle\rangle \rightarrow_{\texttt{invokevirtual}\ M} \langle\langle h, f' :: f'' :: sf\rangle\rangle} \tag{2}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{return} \qquad f' = \langle m', pc', l', s'\rangle}{\langle\langle h, \langle m, pc, l, v :: s\rangle :: f' :: sf\rangle\rangle \rightarrow_{\texttt{return}} \langle\langle h, \langle m', pc' + 1, l', v :: s'\rangle :: sf\rangle\rangle} \tag{3}$$

The rule (2) is slightly more complicated. It reads: for M a method name, the instruction `invokevirtual` M at address (m, pc) of state $\sigma = \langle\langle h, f :: sf\rangle\rangle$ requires that the first frame f on the call stack of σ has an operand stack of the form $loc :: V :: s$, i.e., it starts with a *heap location* denoted by loc, followed by a vector of argument values V. The actual method that will be called is determined by the *lookUp* function that searches upwards in the class hierarchy for the methond name M, starting from the class of the object o that resides in the heap h at the location loc. The new method, together with its starting point $pc = 1$, its vector V of actual parameters, and an empty operand stack ε, constitute a new frame f' pushed on top of the call stack of the resulting state $\sigma' = \langle\langle h, f' :: f'' :: sf\rangle\rangle$. Note, however, that the *second* frame f'' in the call stack is also modified: the sequence $loc :: V$ has been removed from the operand stack of f. This semantics of the `invokevirtual` instruction (which corresponds to the operational definition of the semantics of Java Card), together with the corresponding rules describing its static analysis, made for the most challenging part of the correctness proofs (we return to this point in Section 5).

Finally, the `return` rule (3) removes the last frame from the call stack, and transfers the return value v (from the top of the last frame's operand stack) to the operand stack of the calling frame f'.

2.1 Carmel Flow Logic

The Carmel Flow Logic defined by Hansen [12] specifies a constraint-based data flow analysis for Carmel. This analysis computes a safe approximation of the states that can occur at any program point during execution of a program. This information can then be used to optimize virtual method calls or verify specific properties on the control flow graph (see *e.g.* [4]). Programs

may contain virtual method calls, which are dynamically resolved at execution time ; the analysis reflects this behaviour, and attempts to compute a precise approximation of the called methods and their return values.

Concrete semantic values are either integers or object references. Object references are abstracted by the classes of the objects they refer to, thus, an abstract value is either a subset of the set of classes of the program P or a numerical abstraction. This means that the type $\widehat{\mathrm{Val}}_P$ of abstract values depends on the program P being analysed. This is an example of a dependency that it is important to make explicit because it ensures the finiteness of the abstract domain which would otherwise have an infinite number of subsets of classes. The abstract domain of local variables is another example of an abstract domain that depends on the actual program being analysed (namely, on the number of local variables of the program).

For each program point in P, of the form $(\mathrm{methodName}_P \times \mathrm{progCount}_P)$, the local variables (resp. the operand stack) are abstracted by an array of type $\widehat{\mathrm{LocalVar}}_P$ (resp. a stack of type $\widehat{\mathrm{Stack}}_P$) of abstract values. Then, the *abstract state* domain:

$$\begin{aligned}\widehat{\mathrm{State}}_P = \widehat{\mathrm{Heap}}_P &\times \left(\mathrm{methodName}_P \times \mathrm{progCount}_P \to \widehat{\mathrm{LocalVar}}_P\right) \\ &\times \left(\mathrm{methodName}_P \times \mathrm{progCount}_P \to \widehat{\mathrm{Stack}}_P\right)\end{aligned}$$

contains an over-approximation of all possible concrete heaps [1] and, for each program point, an over-approximation of the local variables and of the operand stack. These approximations are formalised by a relation $\sim$ that connects the concrete domains of the operational semantics and the abstract domains. In logical terms, $s \sim a$ if a is a property of s. In set-theoretic terms, $s \sim a$ if s is a member of the set of states described by a. The formal definition of the $\sim$ relation can be found in the Appendix. Here, we only give an intuitive description:

- a reference to object o is approximated by an abstract value $\hat{V}$ (written $ref(o) \sim \hat{V}$) whenever $\hat{V}$ is a set of classes that contains the class of o,
- the vector of local variables l and operand stack s at a given program address (m, pc), are approximated pointwise,
- a concrete state $\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle$ is approximated by an abstract state $\hat{\Sigma} = (\hat{H}, (\hat{L}, \hat{S}))$ whenever $h \sim \hat{H}$, $l \sim \hat{L}(m, pc)$, and $s \sim \hat{S}(m, pc)$.

The abstract domains are further described in Section 3. An important property of the approximation relation $\sim$ is the *monotonicity with respect to the*

[1] The precise descriptions of the concrete and abstract heap domains are not essential for understanding the rest of the paper; they can be found in the Appendix.

abstract order $\sqsubseteq$. It says that, for each concrete value a (be it a heap, a stack, or a vector of local variables) and abstract values $\widehat{A}, \widehat{A'}$ in the corresponding abstract domain, if $a \sim \widehat{A}$ and $\widehat{A} \sqsubseteq \widehat{A'}$ then $a \sim \widehat{A'}$ holds as well. This property of $\sim$ is proved in Coq once and for all for each concrete and corresponding abstract domain. The relation $\sim$ is used extensively in Section 5 where we show how to prove correctness of the analysis in Coq.

The specification of the flow logic consists of a set of inference rules that for each Carmel instruction define a constraint over an abstract state $\widehat{\Sigma} \in \widehat{\text{State}}$. For $\widehat{\Sigma} = \left(\hat{H}, \hat{L}, \hat{S}\right)$ to be a correct abstraction of program P, $\widehat{\Sigma}$ must satisfy the constraints of the instructions of P. For example, if a `push` instruction is present at address (m, pc), the following constraints should be satisfied:

$$\widehat{\text{push}}\left(c, \hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \tag{4}$$

$$\hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1) \tag{5}$$

where $\widehat{\text{push}}$ is the abstract *push* operation from the abstract domain of stacks.

The constraints (6) and (7) below are attached to the `invokevirtual` instruction. Other constraints attached to this instruction can be found in the Appendix. Together, the constraints (6) and (7) describe the relation between the value of the abstract stack $\hat{S}$ at an address (m, pc) where a method named M is called by an `invokevirtual` instruction, and the value of $\hat{S}$ at the address $(m, pc+1)$ that follows the method's `return`.

$$\widehat{\text{pop}}_n\left(\hat{S}(m, pc), 1 + nbArgs(M)\right) \sqsubseteq \widehat{\text{pop}}\left(\hat{S}(m, pc+1)\right) \tag{6}$$

$$\forall cl \in \widehat{\text{top}}\left(\hat{S}(m, pc)\right),\ \forall m' \in lookUp(M, cl). \qquad (a) \tag{7}$$

$$\widehat{\text{top}}\left(\hat{S}(m', Ret(m'))\right) \sqsubseteq \widehat{\text{top}}\left(\hat{S}(m, pc+1)\right) \qquad (b)$$

In particular, the constraint (7), explained below, plays a particular role in the correctness proof described in Section 5. Assume that the program performs an `invokevirtual` M instruction at an address (m, pc). The constraint (7) computes both (a) a set of methods that contains the method actually called by `invokevirtual`, and (b) an over-approximation of the return values of all the methods computed at Step (a).

- Step (a) simulates the semantics of `invokevirtual` (rule (2)). The rule says that the method actually called is found by looking up for methods named M, in the class of the object referenced by the top of the concrete operand stack when the execution is at address (m, pc). The constraint (7) simulates this behaviour at the abstract level: it searches the class hierarchy for methods called M starting from all the classes contained in the corresponding abstract value $\widehat{\text{top}}\left(\hat{S}(m, pc)\right)$.
- Step (b) is performed by simulating the semantics of the `return` instruction.

By the semantics rule (3), the return value of the method m' actually called is placed on the top of the operand stack at address $(m, pc+1)$. Hence, the constraint (7) imposes that $\widehat{\mathrm{top}}\left(\hat{S}(m, pc+1)\right)$ is greater than the abstract return values $\widehat{\mathrm{top}}\left(\hat{S}(m', Ret(m'))\right)$ of all methods m' computed at Step (a), where $Ret(m')$ is a virtual program point used for collecting abstract results of each method m'.

Note that Step (a) implicitly assumes that the abstract value $\widehat{\mathrm{top}}\left(\hat{S}(m, pc)\right)$ correctly approximates the top of the concrete operand stack when execution is at address (m, pc), i.e., before the `invokevirtual` instruction. That is, the abstract value $\widehat{\mathrm{top}}\left(\hat{S}(m, pc)\right)$ is a set of classes which contains the class of the object o that is referenced by the top of the concrete operand stack when execution is at address (m, pc). This assumption becomes a proof obligation, to be discharged when proving the correctness of the analysis (cf. Section 5).

3 Constructing abstract domains

In this section we define a data type for lattices `(lattice A)`, parameterised by the type of elements of the lattice. We also define higher-order functions which build a lattice object from other lattice objects. This allows to construct the abstract domains (of local variables, stacks, *etc.*) in a compositional fashion from a collection of base abstract domains. The advantage of this modular technique of combining and building lattices is that we do not have to prove properties (such as the finite ascending chain condition, see below) for one big, final lattice, but can do so in a modular fashion for every type of lattice used. Furthermore, local changes to the lattice structure do not invalidate the overall proof.

A lattice object is a record structure with two families of fields: the functional fields which are the operations that will remain in the extracted Ocaml code, and the logical fields that contain properties about the lattice. *E.g.*, the field `join` is a functional field that contains the least upper bound operator of the lattice, whereas the field `acc_property` is a logical field stating that the lattice satisfies the ascending chain condition. The lattice type is conveniently defined as a record type in Coq, as shown in the following Coq declaration, where only details for the `order` relation, the `join` operation and the well-foundedness of the lattice are given. The well-foundedness field will be explained in detail in Section 3.1.

```
Record Lattice [A: Set]: Type := {
  eq  : A → A →  Prop;
  eq_prop : ...    ;;  eq is an equivalence relation

  order : A → A →  Prop;
  order_refl : ∀x,y:A (eq x y)  ⇒ (order x y);
  order_antisym : ∀x,y:A (order x y)  ⇒ (order y x)  ⇒  (eq x y);
  order_trans : ∀x,y,z:A  (order x y) ⇒  (order y z)
                                      ⇒  (order x z);

  join : A → A → A;
  join_bound1 : ∀x,y:A (order x (join x y));
  join_bound2 : ∀x,y:A (order y (join x y));
  join_least : ∀x,y,z:A (order x z) ⇒ (order y z)
                                    ⇒ (order (join x y) z);

  eq_dec : A → A → bool
  eq_dec_prop : ...   ;;  eq_dec is a correct test of equality

  bottom : A;
  bottom_prop : ...   ;;  bottom is the least element

  top : A;
  top_prop : ...   ;;  top is the greatest element

  acc_property :  (well_founded A (λx,y:A, ¬(eq y x)∧(order y x)))
}
```

In this large object, the properties in the logical fields are only necessary during the Coq development to ensure the logical coherence of the structure. Hence only the four functional fields appear in the extracted Ocaml lattice type:

```
type 'a lattice = { join : ('a → 'a → 'a);
                    eq_dec : ('a → 'a → bool);
                    bottom : 'a;
                    top : 'a }
```

Declaring a structure of `Lattice` type will result in a series of proof obligations, one for each of the logical fields. Of these, the last property `acc_property` is the most difficult to establish. It expresses that the strict dual order is well-founded, or, in other words, that there are no infinite, ascending chains. It is the key property used to prove the termination of the final analyser. Thus, strictly speaking we are dealing with lattices satisfying the finite ascending chain-condition but we will for convenience use the general term lattice in the rest of this document.

The lattices are built from two base lattices using four lattice constructors. These constructors are not tied to this particular analysis and can be reused in other contexts.

```
prodLattice : (Lattice A) → (Lattice B) → (Lattice A*B)
sumLattice : (Lattice A) → (Lattice B) → (Lattice (lift A+B))
stackLattice : (Lattice A) → (Lattice (stack A))
arrayLattice : (max:nat) → (Lattice A) → (Lattice (array max A))
```

The most difficult part of each lattice construction is the proof of preservation of `acc_property` (the ascending chain condition), which is essential for defining an analyser that terminates. This is essentially a termination proof which is typically hard to conduct in a proof assistant because the standard techniques of structural orders or well-chosen measures do not apply in the case of lattice orders. The proof has to operate directly with the definition of a well-founded order. We use the standard inductive definition of a well-founded relation, as used in the built-in Coq predicate `well_founded`:

Definition 1 *Let A be a type and $\prec$ a binary relation on A.*

- *The* accessibility *predicat is inductively defined by: an element x of type A is* accessible *if and only if all the predecessors of x are* accessible*,*
- *a binary predicate $\prec$ on a type A is* well-founded *if all the elements of A are accessible for $\prec$.*

The `prodLattice` function is the standard cartesian product with the pointwise order

$$(x_1, y_1) \sqsubseteq_{A\times B} (x_2, y_2) \text{ iff } x_1 \sqsubseteq_A x_2 \wedge y_1 \sqsubseteq_B y_2$$

The ascending chain condition of this structure is proved using the fact that the strict reverse order is a sub-relation of the lexicographic product

$$(x_1, y_1) \sqsupset_{A\times B} (x_2, y_2) \implies x_1 \sqsupset_A x_2 \vee (x_1 =_A x_2 \wedge y_1 \sqsupset_B y_2)$$

The `sumLattice` function builds the separate sum of two lattices A and B according to the Hasse diagram of Figure 1(a). The `acc_property` proof of this lattice is done following the different layers of the diagram: first, we prove that the top element is accessible (no predecessor); then, that all the elements of A, B are accessible (using the fact that $\top$ is accessible and $\sqsupset_A$ well-founded). Finally, we prove that $\bot$ is accessible because all its predecessors are accessible (they are elements of $A \cup B \cup \{\top\}$).

The `stackLattice` constructor builds the lattice of stacks of elements of type `A`. In this lattice, stacks with different sizes are incomparable, according to

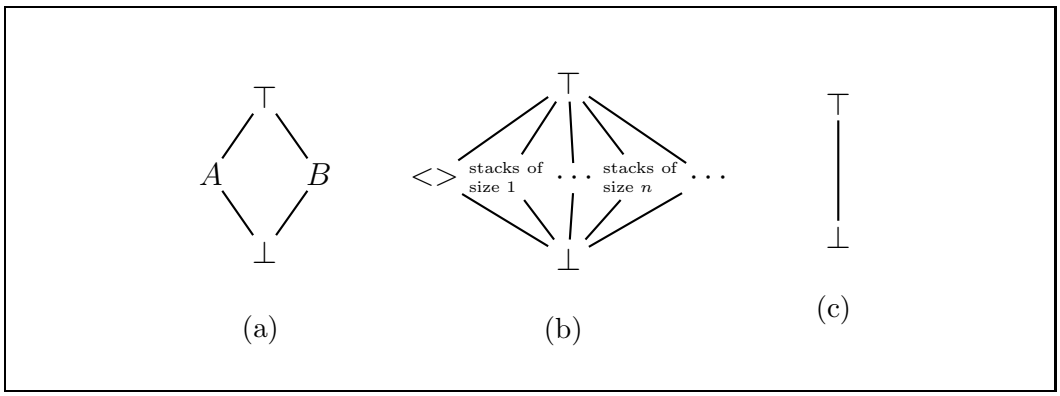

Fig. 1. Hasse diagrams of lattices

the Hasse diagram of Figure 1(b). The ascending chain condition proof again follows the layers of the Hasse diagram but is more technical because of the infinite width of the lattice: for the middle layer, that is, the level of stacks, we use an induction on the stack size. The case of the empty stack is trivial (no predecessor) and for the induction step, we observe that strict inverse order for stacks of size $n+1$ is a sub-relation of the lexicographic product between $(A, \sqsupset)$ (which is supposed well-founded) and the set of stacks of size n (which is well-founded for the strict inverse order by induction hypothesis).

$$(x_1 :: l_1) \sqsupset_{n+1} (x_2 :: l_2) \implies x_1 \sqsupset_A x_2 \vee (x_1 =_A x_2 \wedge l_1 \sqsupset_n l_2)$$

The fourth constructor `arrayLattice` builds the type of arrays whose elements live in a lattice and whose size is bounded by a parameter `max`, using a pointwise order. Notice that arrays of different sizes may be comparable whereas this is not the case for the order we have defined on stacks.

$$t_1 \sqsubseteq t_2 \text{ iff } \forall i \in \{1, \ldots, max\},\ t_1[i] \sqsubseteq_A t_2[i]$$

The array lattice frequently occurs in flow-sensitive analyses where the number of abstract variables depends on the program to analyse—these are then conveniently collected in an array. An efficient implementation of arrays is therefore crucial for obtaining an efficient extracted code, and we have optimized it by using an efficient tree representation of integer maps in the spirit of [19]. The crucial ideas of this implementation are

- to represent arrays using binary trees whose nodes are elements of the array
- to represent indexes using a binary notation; an element at position i in an array is found by interpreting the binary notation of i as the "path" to follow from the root
- to have a lazy structure: if a searched node is missing, its value is by convention $\bot_A$, which allows to represent an array whose elements are all $\bot_A$ by an empty leaf.

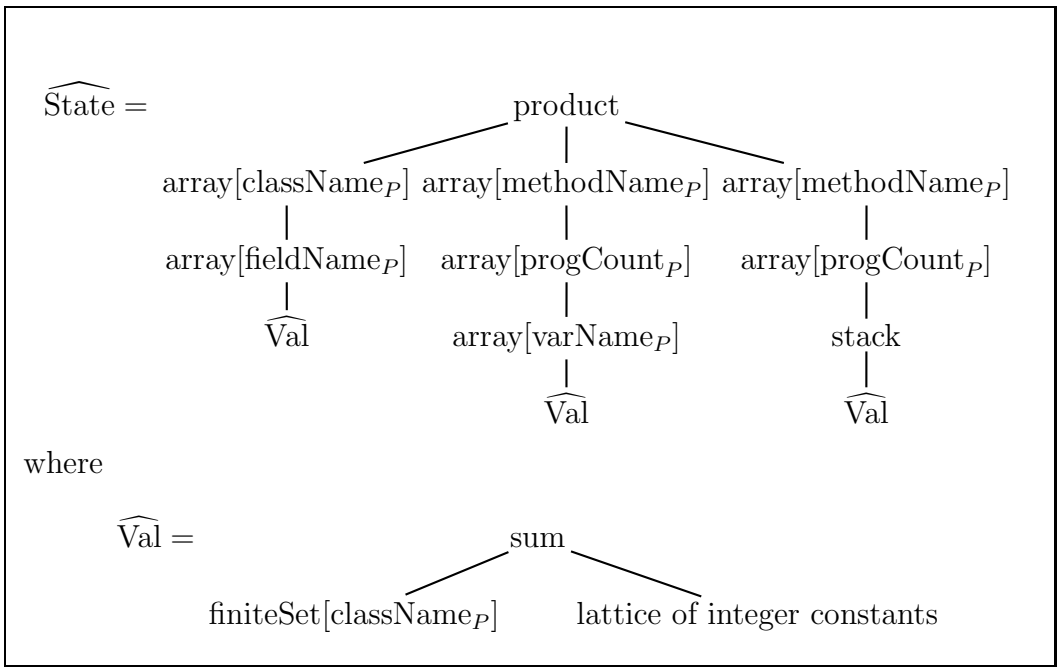

Fig. 2. The lattice of abstract states (each xX$_P$ represents the number of distinct xX elements in program P).

The `acc_property` proof of this lattice is performed by defining an order on trees (a leaf is smaller that any tree; two nodes are smaller if their heads are in the $\sqsubseteq_A$ relation and if their descendents are smaller as well), proving its well-foundedness, and, finally, connecting this order to the array order. It is certainly the most technical of this library. More details on this proof can be found in the corresponding `Coq` development.

In addition to these four functors, two base lattices are defined:

- the flat lattice of integer constants (as used *e.g.*, in constant propagation analysis),
- the lattice of finite sets over a subset $\{0, \ldots, \texttt{max}\}$ of integers: again, an efficient implementation is proposed, by encoding sets using boolean arrays, hence based on the `arrayLattice` functor and two-valued lattice of Figure 1(c).

The lattice employed in our particular analysis is given a graphical representation in Figure 2 (see Appendix for a mathematical description of the lattice). In this diagram, each node represents a lattice functor whose parameters are the sons of the node. For the array functor and the finite-set lattice we write the index domain inside brackets. The modular construction saves a considerable amount of time and effort, *e.g.*, compared to proving `acc_property` for the lattice in Figure 2 as a whole.

Implementing a static program analyser involves building a constraint solver that can compute a solution to constraints like the flow logic constraints shown in Section 2 [17]. The problem of solving a set of constraints over a lattice L of abstract values can be transformed into calculating a *fixpoint* of a monotone function over L—a fixpoint that for reasons of precision should be as small as possible. More precisely, let $f : L \to L$ be a monotone function over L. The basic fixpoint operator `lfp` takes such a monotone function f and computes the least element x of L verifying $f(x) = x$. Furthermore, by a corollary of the Tarski's Fixed Point Theorem, this element can be iteratively calculated as the limit of the (stabilizing) sequence $(f^n(\bot))_{n\in\mathbb{N}}$. Formally, we define the operator `lfp` of type

```
(A:Set) → (L:(Lattice A)) → (f:(A→A)) → (monotone L f) →
  ∃x:A, (eq L x (f x)) ∧ (∀y:A (eq L y (f y)) ⇒ (order L x y))
```

That is, `lfp` takes four arguments : a data type A, a lattice L on A, a function f on A and a proof that f is monotone. It returns the least fixed point of f. We prove in Coq that this type is non-empty, which here consists in instantiating the existentially quantified `x` in the type definition by $\lim_{n\to\infty} f^n(\bot)$. The extraction mechanism of programs from proofs generates for `lfp` the following Ocaml code, in which the purely logical part of the proof (i.e., the part concerned with proving that the chosen witness verifies the fixpoint equation) has been removed:

```
let lfp L f =
    let rec aux x =
       if (L.eq_dec x (f x))  then x  else aux (f x)
    in aux L.bottom
```

We use $x = f(x)$ as halting condition here, but, as is well known from the fixed point theory, we could equally well have used the post-fixed point condition $f(x) \sqsubseteq x$. The equality test appears to be more efficient for the majority of lattices used in our case study.

In order to use the `lfp` operator to solve the constraints arising from the Java Card flow analysis it must be extended to a function `lfp_list` that can deal with systems of the form

$$\{f_i(x) \sqsubseteq x\}_{i=1,\ldots,n}$$

Given a list $f_1, \ldots, f_n$ of monotone functions of type $L \to L$, the operator `lfp_list` computes the least solution x of the system by a round-robin iteration strategy in which the constraints are iterated in increasing order. This computation is implemented by applying the `lfp` operator on the monotone

function $\tilde{f}_n \circ \cdots \circ \tilde{f}_1$, where $\tilde{f}_i(x) = x \sqcup f_i(x)$ for every index i. The fact that this computes a solution to the constraint system is formalised in the type of `lfp_list`, which is expressed in Coq as follows:

```
(l: (L → L) list) → (∀f ∈ l, (monotone L f)) →
 ∃x:A, (∀f ∈ l, (order L (f x) x)) ∧
       (∀ y:A (∀f ∈ l, (order L (f y) y)) ⇒ (order L x y))
```
(8)

This type means that any application of `lfp_list` to a list of functions f_i must be accompanied by a proof of the monotonicity of each f_i. Read at a proof-theoretic level, it states that from the proofs of monotonicity of the f_i we can prove the existence of a least common post-fixpoint for all of the f_i. This function will be used as a generic constraint solver in Section 4.

4 Constructive Constraints

We now turn to the problem of building an analyser that implements the flow analysis from Section 2. The development will be structured into three phases:

(1) The generation of a set of constraints for each instruction.
(2) The building of an analyser `analyse` that computes an abstract state verifying all the constraints generated for a given program.
(3) The proof of correctness of these constraints wrt. the Carmel semantics.

In the rest of this section, we focus on the constraint generation and resolution (Phases 1 and 2). In Section 5 we describe the proof of correctness (Phase 3).

Let $P \vdash \widehat{\Sigma}$ be the predicate meaning that the abstract state $\widehat{\Sigma}$ verifies all constraints of program P (this predicate is defined formally in Definition 5 below). Phases 2 and 3 correspond to proving the following two theorems:

Theorem 2 *For each program P, there exists an abstract state $\widehat{\Sigma}$ satisfying all constraints of P:*

$$\forall P : \text{Program}, \exists \widehat{\Sigma} : \widehat{\text{State}}_P.\ P \vdash \widehat{\Sigma}$$

The constructive proof of this theorem provides the analyser itself: the abstract state $\widehat{\Sigma}$ we construct corresponds to `analyse`(P).

Theorem 3 *An abstract state verifying all the constraints of a program P is*

a correct approximation of the operational semantics of P:

$$\forall P : \text{Program}, \widehat{\Sigma} : \widehat{\text{State}}_P.\ P \vdash \widehat{\Sigma} \Rightarrow [\![P]\!] \sim \widehat{\Sigma}$$

where $[\![P]\!]$ denotes the set of reachable states of program P and $\sim$ is the correctness relation between concrete domains of the operational semantics and the abstract domains.

Putting these two theorems together, we get the correctness of the analyser:

Theorem 4 (Global Correctness)

$$[\![P]\!] \sim \texttt{analyse}(P)$$

4.1 Generating the constraints

When formalising the analysis, several representations of the constraints are possible.

- For the correctness proof (Phase 3), it is sufficient to know which order relation is induced by the constraints on a given set of components of the abstract state. Using an inductive definition for constraints would naturally provide the necessary predicates for this phase. Relational constraints written as (4)–(7) could be translated in Coq in a straightforward manner using inductive definitions.
- On the other hand, the construction of an effective analyser (Phase 2) requires to represent constraints in a functional form like $f(X) \sqsubseteq X$. This representation is typically difficult to extract directly from inductive definitions. [2]

This is why an internal representation of constraints is defined in Phase 1, which allows for both interpretations and leaves room for reuse in other analyses.

Looking at Formulas (4) and (5) for the `push` instruction (Section 2.1), we note that the representation of constraints must contain the following informations: (i) the components of the abstract state that are involved in the constraint, (ii) a start address ad_1 and an end address ad_2 of the data flow, and (iii) the transformation F that is applied to the data that flows. For example, constraint (4) only affects the abstract state $\hat{S}$, and we have $ad_1 = pc, ad_2 = pc+1$, and $F = \lambda\hat{S}.\widehat{\text{push}}(c, \hat{S})$. This naturally leads to an inductive data type of the form

[2] Cognoscenti will know that the Coq extraction mechanism is not able to extract computational content of inductive definition made in the sort `Prop`.

$$
\begin{array}{l}
\texttt{type}\ \mathrm{Constraint}_P\ \texttt{=} \\
\quad \texttt{| S2S of}\quad \mathrm{Address} * \mathrm{Address} * (\widehat{\mathrm{Stack}}_P \to \widehat{\mathrm{Stack}}_P) \\
\quad \texttt{| L2L of}\quad \mathrm{Address} * \mathrm{Address} * (\widehat{\mathrm{LocalVar}}_P \to \widehat{\mathrm{LocalVar}}_P) \\
\quad \ldots
\end{array}
\tag{9}
$$

where each constructor represents a type of dependency between components of the abstract state. For example, `S2S` is a constructor to express a constraint on an abstract stack which depends on another abstract stack. For the particular analysis discussed here eleven constructors were employed.

Each constraint, initially acting on a part of the abstract state, is extended to a function on the whole abstract state, using a mapping

$$\mathcal{F}[\![\cdot]\!] : \mathrm{Constraint}_P \to \left(\widehat{\mathrm{State}}_P \to \widehat{\mathrm{State}}_P\right)$$

for which we prove that it preserves the monotonicity of constraints. *E.g.,* for the `push` instruction

$$
\begin{array}{l}
\mathcal{F}[\![(\texttt{S2S}\ \ ad_1\ \ ad_2\ \ F)]\!] := \lambda(H, L, S).\ (H, L, S[ad_2 \mapsto F(S(ad_1))]) \\
\mathcal{F}[\![(\texttt{L2L}\ \ ad_1\ \ ad_2\ \ F)]\!] := \lambda(H, L, S).\ (H, L[ad_2 \mapsto F(L(ad_1))], S)
\end{array}
$$

Based on this definition of the constraints we define a function `cst_gen`, which for each address, returns the list of constraints for the corresponding instruction in a syntax of the form (9). Continuing with the `push` instruction, the corresponding code part is:

$$
\begin{array}{l}
\texttt{cst\_gen}_P\ \ :=\ \ \lambda\texttt{(m,pc)} \\
\quad \texttt{Case instructionAt}_P\ \texttt{(m,pc) of} \\
\qquad \texttt{| (push}\ c\texttt{)} \to\ \ \texttt{(S2S (m,pc) (m,pc+1)}\ \lambda\hat{S}.\ \widehat{\texttt{push}}\left(\hat{c}, \hat{S}\right)\texttt{) ::} \\
\qquad\qquad\qquad\qquad \texttt{(L2L (m,pc) (m,pc+1)}\ \lambda\hat{L}.\ \hat{L}\texttt{)} \\
\qquad \texttt{...}
\end{array}
$$

The well-formedness of this function depends on the actual program P being analysed because every instance of (m, pc) must be shown to refer to a valid program point of P. In a paper-and-pencil proof, this is often left as an implicit hypothesis. In a formal proof however, this fact must be stated explicitly. In a dependently-typed framework, the constraint generation will thus be parameterised by the program being analysed, yielding a function $\texttt{cst\_gen}_P$ which takes as argument an address (m, pc) in the program P and generates the constraints corresponding to the type of instruction at (m, pc).

We now can formally define what it means for an abstract state $\widehat{\Sigma}$ to verify all the constraints of a program P.

Definition 5 *Let* Addr_P *denote the set of addresses appearing in* P*:*

$$P \vdash \widehat{\Sigma} \quad \equiv \quad \forall (m, pc) : Addr_P, \forall c \in \texttt{cst\_gen}_P(m, pc).\ \mathcal{F}[\![c]\!](\hat{\Sigma}) \sqsubseteq \hat{\Sigma}$$

4.2 Construction of the analyser

Recall that the goal is to build an analyser that, given an input program, computes an abstract state that verifies all the constraints of the program. We construct a function `analyse` of dependent type $(P : \mathrm{Program}) \to \widehat{\mathrm{State}}_P$ which must verify

$$\forall P : \mathrm{Program},\ P \vdash \texttt{analyse}(P) \tag{10}$$

In addition, we want to obtain a non-trivial solution of the constraint system: *e.g.*, an analyser returning the top element of the lattice for any input is a correct solution, but of poor interest. We thus add the requirement that our solution is the least solution of the constraint system:

$$\forall P : \mathrm{Program}, \widehat{\Sigma} : \widehat{\mathrm{State}}_P\ P \vdash \widehat{\Sigma} \Rightarrow \texttt{analyse}(P) \sqsubseteq \widehat{\Sigma} \tag{11}$$

The constraint resolution tool is based on the generic solver `lfp_list` (8) described in Section 3.2. The most difficult part of the work has already be done during the definition of the solver, *i.e.*, proof of termination and correctness. It is instantiated here with the particular abstract state lattice of the analysis (depicted in Figure 2); then,

- For each instruction of program P, the constraints are collected from the lists defined by $\texttt{cst\_gen}_P$ (cf. Section 4.1).
- Each constraint is translated into a function on abstract states using the mapping $\mathcal{F}$. The resulting list of functional constraints is called $\texttt{collect\_func}_P$. As $\mathcal{F}$ preserves the monotonicity of constraints, we conclude that

$$\forall f \in \texttt{collect\_func}_P\ f \text{ is monotone} \tag{12}$$

We now have all the ingredients to define the constraint solver:

$$\texttt{analyse}(P) = \texttt{lfp\_list}(\widehat{\mathrm{State}}_P, \texttt{collect\_func}_P, \texttt{collect\_func\_monotone})$$

where `collect_func_monotone` is the name given to the proof of (12).

By the properties of `lfp_list` (defined by Formula (8)) we know that $\texttt{analyse}(P)$ is the least abstract state $\widehat{\Sigma}$ in $\widehat{\mathrm{State}}_P$ verifying

$$\forall f \in \texttt{collect\_func}_P \quad f(\widehat{\Sigma}) \sqsubseteq \widehat{\Sigma} \tag{13}$$

Thus, $\texttt{analyse}(P)$ is the least $\widehat{\Sigma}$ satisfying $P \vdash \widehat{\Sigma}$.

We stress that this approach defines a methodology that remains valid for other analyses. Indeed, all proofs in this section are independent of the system of constraints defined by the user. They only depend on the different types of constraints introduced as in (9) (`S2S`, `L2L`,...). As a consequence, modifications to the system of constraints only affect proofs made about the monotonicity of constraints and during Section 5, rather than the construction and the correctness of the solver itself.

5 Correctness

Section 4 has shown that an effective solver for the constraints of a program exists. We now show that the solver is indeed a correct analyser for the program. The analysis is correct if every abstract state $\widehat{\Sigma}$ satisfying all the constraints of the analysis is an approximation of the reachable states $[\![P]\!]$ of the program:

$$\forall P : \text{Program}, \widehat{\Sigma} : \widehat{\text{State}}_P.\ P \vdash \widehat{\Sigma} \Rightarrow [\![P]\!] \sim \widehat{\Sigma} \tag{14}$$

The implication (14) has been proved in Coq by well-founded induction on the length of the program executions. The base step is trivial. The induction step depends on whether the last instruction is `return` or some other instruction.

5.1 Induction Step: the non-`return` *Instructions*

For I an instruction, let $\rightarrow_I$ denote the transition relation of I (examples of which have been given in Section 2). The general form of the induction step for any Carmel instruction $I \neq$ `return` is of the following form.

$$P \vdash \widehat{\Sigma} \Longrightarrow \forall \sigma, \sigma' \in [\![P]\!], \sigma \sim \widehat{\Sigma} \wedge \sigma \rightarrow_I \sigma' \Rightarrow \sigma' \sim \widehat{\Sigma} \tag{15}$$

That is, if a state σ is approximated by an abstract state $\widehat{\Sigma}$ that satisfies the constraints of program P, and if, by performing instruction I, the state σ becomes σ', then σ' is approximated by $\widehat{\Sigma}$ as well. We now sketch the proof of (15).

(1) A Coq script unfolds the definition of the transition rule for instruction I.
(2) Then, another script unfolds the definitions of $\sigma \sim \widehat{\Sigma}$ and $P \vdash \Sigma$ and automatically turns them into hypotheses of the current Coq goal[3]. For example, if $\sigma = \langle\!\langle h, \langle m, pc, l, s\rangle :: sf \rangle\!\rangle$, $\widehat{\Sigma} = (\widehat{H}, \widehat{L}, \widehat{S})$, and $I =$ `push` c then the following hypotheses are generated:

$$s \sim \widehat{S}(m, pc), \quad \widehat{\text{push}}\left(c, \widehat{S}\,(m, pc)\right) \sqsubseteq \widehat{S}\,(m, pc+1) \tag{16}$$

[3] This script simulates the standard Coq *inversion* tactic for inductive datatypes.

(3) Next, the conclusion of the Coq goal: $\sigma' \sim \widehat{\Sigma}$ — *i.e.*, the new state σ' is approximated by the abstract state $\widehat{\Sigma} = (\widehat{H}, \widehat{L}, \widehat{S})$ — is split into three subgoals, one for each of the components $(\widehat{H}, \widehat{L}, \widehat{S})$ of $\widehat{\Sigma}$.

For $I = \texttt{push}\ c$, the subgoal corresponding to the abstract stack $\widehat{S}$ is

$$c :: s \sim \widehat{S}(m, pc + 1) \tag{17}$$

(4) Finally, the subgoals generated at Step 3 are proved using the hypotheses generated at Step 2 and monotonicity of $\sim$ with respect to $\sqsubseteq$ (cf. Section 2). For $I = \texttt{push}\ c$, the only non-trivial subgoal is represented by Formula (17). It is proved using the fact that $\widehat{\text{push}}$ is a correct abstraction of the concrete *push* operator "::", *i.e.*, $c :: s \sim \widehat{\text{push}}\left(c, \widehat{S}\,(m, pc)\right)$. This, together with the hypothesis (16) and the monotonicity of the $\sim$ relation for stacks, implies the subgoal (17), and the proof is done.

5.2 Induction Step: the Case of the `return` *Instruction*

Formula (15) above has the general aspect of the induction step in a proof by *simple induction*. That is, **if** the abstract state $\widehat{\Sigma}$ approximates the concrete state σ, **then** $\widehat{\Sigma}$ also correctly approximates all *immediate* successors σ' of σ. However, this simple implication could not be proved for the `return` instruction. This is because the effect of the `return` is simulated by a constraint (cf. Formula (7)) attached to a different instruction: the `invokevirtual` instruction that called the method now performing the `return`. As seen in Section 2.1, in order to evaluate which methods may have been called, the constraint (7) must be used together with the assumption that *the top of the concrete stack at the address* (m, pc) *where the* `invokevirtual` *instruction has been performed*, is *correctly approximated by the top of the abstract stack* $\widehat{top}\left(\widehat{S}(m, pc)\right)$.

More generally, we need to assume that the concrete state σ'' where the `invokevirtual` instruction has been performed, was correctly approximated by the abstract state $\widehat{\Sigma}$ as well. But σ'' may have been encountered arbitrarily far in the past. Hence, our proof of the induction step for the `return` instruction uses a *well-founded induction* hypothesis, which imposes that the whole proof of correctness be done by well-founded induction.

Let $[\![P]\!]_{<n}$ denote the set of states of program P that are reachable using less than n instructions. The induction step for $I = \texttt{return}$ is:

$$\forall n \in \mathbb{N}.[\forall \sigma'' \in [\![P]\!]_{<n}.\ \sigma'' \sim \widehat{\Sigma}] \Rightarrow \forall \sigma \in [\![P]\!]_{<n}, \forall \sigma'.[\ \sigma \rightarrow_I \sigma' \Rightarrow \sigma' \sim \widehat{\Sigma}] \tag{18}$$

Formula (18) reads: **if** the abstract state $\widehat{\Sigma}$ approximate all earlier states σ'' (well-founded induction hypothesis); **and** σ evolves, by performing a `return` instruction, into σ'; **then**, $\widehat{\Sigma}$ approximates σ' as well.

The proof of Theorem (18) is substantially more involved than the proofs of Theorems (15). It should be pointed out that this difficulty had been avoided by a previous pencil-and-paper proof [12], where some details - the actual JavaCard semantics of the `invokevirtual` instruction - have been overlooked.

6 Related work

Proving correctness of program analyses is one of the main applications of the theory of abstract interpretation [9]. However, most of the existing proofs are pencil-and-paper proofs of analyses (formal specifications) and not mechanised proofs of analysers (implementations of analyses). The only attempt of formalising the theory of abstract interpretation with a proof assistant is that of Monniaux [16] who has built a Coq theory of Galois connections. Prost in his thesis [21] conduces a theoretical study of the relation between type theory and program analysis, but this work did not lead to an implementation of a concrete analysis.

Mechanical verification of program processing tools has so far mainly focussed on the correctness of optimising compilers. Genet *et al.* [11] use the generic proof assistant PVS for proving the correctness of algorithms for transforming Java Card byte code into the more compact CAP format. Similar work was done by Denney [10], using the program extraction mechanism of Coq. These optimisations do not involve any sophisticated static analysis. Lerner *et al.* [14] have developed Cobalt, a dedicated programming language for writing C program optimisers and automatically proving their soundness. This language allow to define statement transformation guarded by predicates on execution traces. To prove the correctness of the optimisation, these *transformation patterns* produce proof-obligations to be discharged by an automatic theorem prover. The authors propose an execution engine to compile Cobalt descriptions in an executable form. The framework allows to write several optimisations whose correctness is automatically proved by the Simplify theorem prover. The scope of this work seem currently restricted to intra-procedural analysis with simple lattices of approximations. Finally, ongoing work in the French research action "Concert" [8] currently explores the feasibility of developing a realistic certified compiler in Coq. First results concern the certification of three classical low-level optimizations based on dataflow analysis and some first experiments in program transformation.

Previous formalisations of static analyses for Java (Card) byte code have all dealt with **intra**-procedural type verification. In contrast, we have also shown how to handle **inter**-procedural data flow analysis in a natural manner; this is due to the fact that we use the general setting of Flow Logic [18] and constraint-based analysis. Research on mechanical verification of the Java byte

code verifier includes that of Barthe *et al.* [2] who have shown how to formalise the Java Card byte code verification in the proof assistant Coq by isolating the byte code verification in an executable semantics of the language. In [1], they propose to automate the derivation of a certified verifier from a formalisation of the JCVM. Their approach does not rely on a general theory of static analysis, and is oriented towards type verification. Bertot [3] used the Coq system to extract a certified bytecode analyser specialized for object initialization, but no attention has been paid to the efficiency of the analyser. In [7], Coglio *et al.* described their ongoing efforts to implement a bytecode verifier by refinement from the specification of a constraint-solving problem on lattices. Klein and Nipkow [13] have proved the correctness of a Java byte code verifier using the proof assistant Isabelle/HOL. In particular their work includes a correctness proof of Kildall's iterative workset algorithm for solving data flow equations. They also provide a modular construction of lattices. The major difference with our approach is the use of abstract data types that are not implementable as such.

An alternative to the Coq proof extraction mechanism is the B method that has had considerable industrial success. Casset *et al.* [6] have extracted a proof-carrying code-based on-card bytecode verifier for Java Card from a high-level specification by a succession of refinement steps using the B technique. The development required the proof of thousands of proof obligations, of which several hundreds could not be dealt with automatically by the B prover. The B tool could most probably be used for building an analyzer like ours but we doubt that using B would lead to a simpler proof effort. In addition, the program extraction mechanism in B does not enjoy the same solid foundations as that of Coq. Hence our decision to base our development on Coq.

7 Conclusion

The results presented in this article demonstrates that it is feasible to construct a non-trivial, provably correct data flow analyzer using the program extraction mechanism of constructive logic implemented in Coq. This bridges the gap that often exists between a paper-specification of an analysis and the analyser that is actually implemented. Our approach applies to analyses expressed in the constraint-based Flow Logic specification framework and is hence applicable to a large variety of program analyses for different language paradigms. We have instantiated it to a data flow analysis for Java Card. To the best of our knowledge, it is the first formal construction (with proof of correctness) of a data flow analysis other than the Java byte code verifier.

Formalising a program analyser in a proof assistant imposes a strict discipline that catches a certain number of bugs, including typing errors in the spec-

ification. The present development revealed several (innocuous) inaccuracies in the pencil-and-paper specifications and proof of correctness. Moreover, it pinpointed the adjustment that had been made of the actual semantics of Java Card in the correctness proof on paper—an adjustment that (as argued in Section 5.2) made the proof far simpler than a proof done against a more accurate semantics.

Our methodology makes use of the proof-as-programs paradigm. This paradigm is sometimes presented as developing programs as a by-product of building a constructive proof interactively and incrementally for an "existential" theorem with a proof assistant. While this presentation is conceptually simple and appealing, the development of any non-trivial piece of software (including the present analyser) rather tends to be done by defining (most of) the function and then showing that it is indeed a witness to the theorem. This technique has the further advantage that it is simpler to control the efficiency of the resulting program. In our case, the provably correct analyser was developed by splitting the correctness proof into

- an existential proof of a solution to a constraint system from which a constraint solver can be extracted and,
- a proof that the solutions to the constraint system are correct approximations of the semantics of the analysed program; this proof does not contribute to the actual construction of the solution.

The methodology includes several steps of varying complexity. The development of the lattice library required a Coq expert to structure the proofs of the properties associated with the lattice constructors. Once this library in place, it turned out to be a relatively straightforward task to prove correctness of the constraint generation and to extend the constraint generation to instructions others than those originally studied. It took a Coq neophyte less than two months to complete the correctness proof, including the time and effort needed to understand the general framework of the project. Only basic features of the tool, those available in any other general-purpose theorem prover, have been employed in the correctness proof.

The program extraction mechanism has a reputation for producing inefficient programs. This is not the case with our methodology: the extracted analyser is about 2000 lines of Ocaml code and takes only a few seconds to analyse 1000 lines of bytecode. The extracted version of `analyse` has now a type $\text{Program} \to \widehat{\text{State}}$ because Ocaml does not have dependent types. As mentioned above, the methodology leaves some possibilities for programming the resolution mechanism. This, and the inclusion of widening operators, is one important step forward to be accomplished. Another is further automation of the proof obligations arising during the development of the analyser in order to make the methodology the standard way of implementing static analysers.

A Syntax

Instruction ::=	`nop`	
	`push` c	stack manipulation
	`pop`	
	`dup`	
	`dup2`	
	`swap`	
	`numop` op	
	`load` x	local variables manipulation
	`store` x	
	`if` pc	jump
	`goto` pc	
	`new` cl	heap manipulation
	`putfield` f	
	`getfield` f	
	`invokevirtual` m_{id}	method call and return
	`return`	

B Semantics

$$
\begin{aligned}
\text{Value} &= \text{num } n \quad n \in \mathbb{N} \\
&\quad\ \text{ref } r \quad\ r \in \text{Reference} \\
&\quad\ \text{null} \\
\text{Stack} &= \text{Value}^* \\
\text{LocalVar} &= \text{Var} \to \text{Value} \\
\text{Frame} &= \text{ProgCount} \times \text{nameMethod} \times \text{LocalVar} \times \text{Stack} \\
\text{CallStack} &= \text{Frame}^* \\
\text{Object} &= \text{nameClass} \times (\text{FieldName} \to \text{Value}) \\
\text{Heap} &= \text{Reference} \to \text{Object}_\perp \\
\text{State} &= \text{Heap} \times \text{CallStack}
\end{aligned}
$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{nop}}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{nop}} \langle\!\langle h, \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{push } c}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{push } c} \langle\!\langle h, \langle m, pc+1, l, c :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{pop}}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{pop}} \langle\!\langle h, \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{dup}}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{dup}} \langle\!\langle h, \langle m, pc+1, l, v :: v :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{dup2}}{\langle\!\langle h, \langle m, pc, l, v_1 :: v_2 :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{dup2}} \langle\!\langle h, \langle m, pc+1, l, v_1 :: v_2 :: v_1 :: v_2 :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{swap}}{\langle\!\langle h, \langle m, pc, l, v_1 :: v_2 :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{swap}} \langle\!\langle h, \langle m, pc+1, l, v_2 :: v_1 :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{numop } op}{\langle\!\langle h, \langle m, pc, l, n_1 :: n_2 :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{numop } op} \langle\!\langle h, \langle m, pc+1, l, [\![op]\!](n_1, n_2) :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{load } x}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{load } x} \langle\!\langle h, \langle m, pc+1, l, l[x] :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{store } x}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{store } x} \langle\!\langle h, \langle m, pc+1, l[x \mapsto v], s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{if } pc' \qquad n = 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{if } pc'} \langle\!\langle h, \langle m, pc', l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{if } pc' \qquad n \neq 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{if } pc'} \langle\!\langle h, \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{goto } pc'}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{goto } pc'} \langle\!\langle h, \langle m, pc', l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\begin{array}{c}\text{instructionAt}_P(m, pc) = \texttt{new } cl \\ \exists c \in \text{classes}(P) \text{ with } \text{nameClass}(c) = cl \qquad (h', loc) = \text{newObject}(cl, h)\end{array}}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{new } cl} \langle\!\langle h', \langle m, pc+1, l, loc :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{putfield } f \qquad h(loc) = o \qquad o' = o[f \mapsto v]}{\langle\!\langle h, \langle m, pc, l, v :: loc :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{putfield } f} \langle\!\langle h[loc \mapsto o'], \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{getfield } f \qquad h(loc) = o}{\langle\!\langle h, \langle m, pc, l, loc :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{getfield } f} \langle\!\langle h, \langle m, pc+1, l, \text{fieldValue}(o, f) :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\begin{array}{c}\text{instructionAt}_P(m, pc) = \texttt{invokevirtual } M \\ h(loc) = o \qquad m' = lookUp(M, class(o)) \\ f' = \langle m', 1, V, \varepsilon\rangle \qquad f'' = \langle m, pc, l, s\rangle\end{array}}{\langle\!\langle h, \langle m, pc, l, loc :: V :: s\rangle :: sf\rangle\!\rangle \rightarrow_{\texttt{invokevirtual } M} \langle\!\langle h, f' :: f'' :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{return} \qquad f' = \langle m', pc', l', s'\rangle}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: f' :: sf\rangle\!\rangle \rightarrow_{\texttt{return}} \langle\!\langle h, \langle m', pc'+1, l', v :: s'\rangle :: sf\rangle\!\rangle}$$

C Abstract lattices

$$
\begin{aligned}
\widehat{\mathrm{Num}} &:= \mathbb{N}_{\perp}^{\top} & \widehat{\mathrm{Ref}}_P &:= \mathcal{P}(\mathrm{ClassName}_P) \\
\widehat{\mathrm{Val}}_P &:= \left(\widehat{\mathrm{Ref}}_P + \widehat{\mathrm{Num}}\right)_{\perp}^{\top} & \widehat{\mathrm{Stack}}_P &:= \left(\widehat{\mathrm{Val}}_P^{*}\right)_{\perp}^{\top} \\
\widehat{\mathrm{LocalVar}}_P &:= \mathrm{Var}_P \to \widehat{\mathrm{Val}}_P & \widehat{\mathrm{Object}}_P &:= \mathrm{FieldName}_P \to \widehat{\mathrm{Val}}_P \\
\widehat{\mathrm{Heap}}_P &:= \mathrm{ClassName}_P \to \widehat{\mathrm{Object}}_P \\
\widehat{\mathrm{State}}_P &:= \widehat{\mathrm{Heap}}_P \times \left(\mathrm{nameMethod}_P \times \mathrm{ProgCount}_P \ \to\ \widehat{\mathrm{LocalVar}}_P\right) \\
&\quad \times \left(\mathrm{nameMethod}_P \times \mathrm{ProgCount}_P \ \to\ \widehat{\mathrm{Stack}}_P\right)
\end{aligned}
$$

with

$$
\begin{aligned}
\mathrm{Var}_P &:= \{x \in \mathrm{Var} \mid x \text{ appears in } P\} \\
\mathrm{FieldName}_P &:= \{f \in \mathrm{FieldName} \mid f \text{ appears in } P\} \\
\mathrm{ClassName}_P &:= \{cl \in \mathrm{ClassName} \mid cl \text{ appears in } P\}
\end{aligned}
$$

D Correctness relations

$$
\begin{aligned}
n \ &\sim_{\mathrm{Num}} \ \hat{N} \ \text{ iff } \ \hat{N} = \{n\} \vee \hat{N} = \top \\
r \ &\sim^{h}_{\mathrm{Ref}} \ \hat{R} \ \text{ iff } \ \left(h(r) = o \Rightarrow \{\mathrm{class}(o)\} \sqsubseteq_{\mathrm{Ref}} \hat{R}\right) \\
v \ &\sim^{h}_{\mathrm{Val}} \ \hat{V} \ \text{ iff } \ v = \mathrm{null} \ \vee \ \hat{V} = \top_{\mathrm{Val}} \vee \\
&\qquad \left(v \in \mathrm{Reference} \wedge \hat{V} \in \widehat{\mathrm{Ref}} \wedge v \sim^{h}_{\mathrm{Ref}} \hat{V}\right) \ \vee \\
&\qquad \left(v \in \mathrm{Num} \wedge \hat{V} \in \widehat{\mathrm{Num}} \wedge v \sim_{\mathrm{Num}} \hat{V}\right) \\
v_1 :: \cdots :: v_n \ &\sim^{h}_{\mathrm{Stack}} \ \hat{S} \ \text{ iff } \ \hat{S} = \top_{\mathrm{Stack}} \vee \\
&\qquad \begin{pmatrix} \hat{S} = \hat{V}_1 :: \cdots :: \hat{V}_n \wedge \\ v_1 \sim^{h}_{\mathrm{Val}} \hat{V}_1 \wedge \cdots \wedge v_n \sim^{h}_{\mathrm{Val}} \hat{V}_n \end{pmatrix} \\
l \sim^{h}_{\mathrm{LocalVar}} \hat{L} \ &\text{ iff } \ \forall x \in \mathrm{Var},\ l(x) \sim^{h}_{\mathrm{Val}} \hat{L}\,(x) \\
o \ &\sim^{h}_{\mathrm{Object}} \ \hat{O} \ \text{ iff } \ \forall f \in \mathrm{FieldName},\ \mathrm{fieldValue}(o, f) \sim^{h}_{\mathrm{Val}} \hat{O}\,(f) \\
h \ &\sim_{\mathrm{Heap}} \ \hat{H} \ \text{ iff } \ \forall r \in \mathrm{Reference}, \\
&\qquad h(r) = o \ \Rightarrow \ o \sim^{h}_{\mathrm{Object}} \hat{H}\,(\mathrm{class}(o))
\end{aligned}
$$

$$\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \sim_{\text{State}} \left(\hat{H}, \hat{S}, \hat{L}\right) \text{ iff } h \sim_{\text{Heap}} \hat{S} \wedge l \sim^{h}_{\text{LocalVar}} \hat{L}\,(m, pc) \wedge s \sim^{h}_{\text{Stack}} \hat{S}\,(m, pc)$$

E Constraints

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{nop} \quad \text{iff } \hat{S}\,(m, pc) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{push}\ c \quad \text{iff } \widehat{\text{push}}\left(\widehat{c}, \hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{pop} \quad \text{iff } \widehat{\text{pop}}\left(\hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{numop}\ op \quad \text{iff } \widehat{\text{binop}}\left(op, \hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{dup} \quad \text{iff } \widehat{\text{push}}\left(\widehat{\text{top}}\left(\hat{S}\,(m, pc)\right), \hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{dup2} \quad \text{iff } \widehat{\text{push}}\left(\widehat{\text{top}}\left(\hat{S}\,(m, pc)\right), \widehat{\text{push}}\left(\widehat{\text{top}}\left(\widehat{\text{pop}}\left(\hat{S}\,(m, pc)\right)\right), \hat{S}\,(m, pc)\right)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{swap} \quad \text{iff } \widehat{\text{push}}\left(\widehat{\text{top}}\left(\widehat{\text{pop}}\left(\hat{S}\,(m, pc)\right)\right), \widehat{\text{push}}\left(\widehat{\text{top}}\left(\hat{S}\,(m, pc)\right), \widehat{\text{pop}}\left(\widehat{\text{pop}}\left(\hat{S}\,(m, pc)\right)\right)\right)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{load}\ x \quad \text{iff } \widehat{\text{push}}\left(\widehat{\text{apply}}\left(\hat{L}\,(m, pc), x\right), \hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1)$$

$$\left(\hat{H}, \hat{L}, \hat{S}\right) \models (m, pc) : \texttt{store}\ x \quad \text{iff } \widehat{\text{pop}}\left(\hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \;\; \widehat{\text{subst}}\left(\hat{L}\,(m, pc), x, \widehat{\text{top}}\left(\hat{S}\,(m, pc)\right)\right) \sqsubseteq \hat{L}\,(m, pc+1)$$

$\left(\hat{H},\hat{L},\hat{S}\right) \models (m,pc) : \texttt{if } pc'$

iff $\widehat{\text{test}}_{=0}\left(\widehat{\text{top}}\left(\hat{S}\,(m,pc)\right), \widehat{\text{pop}}\left(\hat{S}\,(m,pc)\right)\right) \sqsubseteq \hat{S}\,(m,pc')$

$\widehat{\text{test}}_{=0}\left(\widehat{\text{top}}\left(\hat{S}\,(m,pc)\right), \hat{L}\,(m,pc)\right) \sqsubseteq \hat{L}\,(m,pc')$

$\widehat{\text{test}}_{\neq 0}\left(\widehat{\text{top}}\left(\hat{S}\,(m,pc)\right), \widehat{\text{pop}}\left(\hat{S}\,(m,pc)\right)\right) \sqsubseteq \hat{S}\,(m,pc+1)$

$\widehat{\text{test}}_{\neq 0}\left(\widehat{\text{top}}\left(\hat{S}\,(m,pc)\right), \hat{L}\,(m,pc)\right) \sqsubseteq \hat{L}\,(m,pc+1)$

$\left(\hat{H},\hat{L},\hat{S}\right) \models (m,pc) : \texttt{new } cl$

iff $\widehat{\text{push}}\left(\{cl\}, \hat{S}\,(m,pc)\right) \sqsubseteq \hat{S}\,(m,pc+1)$

$\hat{L}\,(m,pc) \sqsubseteq \hat{L}\,(m,pc+1)$

$\widehat{\text{default}}(cl) \sqsubseteq \widehat{\text{apply}}\left(\hat{H},cl\right)$

$\left(\hat{H},\hat{L},\hat{S}\right) \models (m,pc) : \texttt{getfield } f$

iff $\forall cl \in \widehat{\text{top}}\left(\hat{S}\,(m,pc)\right) :$

$\widehat{\text{push}}\left(\widehat{\text{apply}}\left(\widehat{\text{apply}}\left(\hat{H},cl\right), f\right), \widehat{\text{pop}}\left(\hat{S}\,(m,pc)\right)\right) \sqsubseteq \hat{S}\,(m,pc+1)$

$\hat{L}\,(m,pc) \sqsubseteq \hat{L}\,(m,pc+1)$

$\left(\hat{H},\hat{L},\hat{S}\right) \models (m,pc) : \texttt{putfield } f$

iff $\widehat{\text{pop}}\left(\widehat{\text{pop}}\left(\hat{S}\,(m,pc)\right)\right) \sqsubseteq \hat{S}\,(m,pc+1)$

$\hat{L}\,(m,pc) \sqsubseteq \hat{L}\,(m,pc+1)$

$\forall cl \in \widehat{\text{top}}\left(\widehat{\text{pop}}\left(\hat{S}\,(m,pc)\right)\right) :$

$\widehat{\text{top}}\left(\hat{S}\,(m,pc)\right) \sqsubseteq \widehat{\text{apply}}\left(\widehat{\text{apply}}\left(\hat{H},cl\right), f\right)$

$\left(\hat{H},\hat{L},\hat{S}\right) \models (m,pc) : \texttt{invokevirtual } M$

iff $\hat{L}\,(m,pc) \sqsubseteq \hat{L}\,(m,pc+1)$

$\widehat{\text{pop}}_n\left(\hat{S}(m,pc), 1+nbArgs(M)\right) \sqsubseteq \widehat{\text{pop}}\left(\hat{S}(m,pc+1)\right)$

$\forall cl \in \widehat{\text{top}}\left(\hat{S}(m,pc)\right), \forall m' \in lookUp(M,cl),$

$\widehat{\text{top}}\left(\hat{S}(m', Ret(m'))\right) \sqsubseteq \widehat{\text{top}}\left(\hat{S}(m,pc+1)\right)$

$\{cl\} :: \widehat{\text{pop}}_n\left(\hat{S}\,(m,pc), nbArgs(M)\right) \sqsubseteq \hat{L}\,(m',1)\,[0..n]$

$\widehat{\text{nil}} \sqsubseteq \hat{S}\,(m',1)$

$\left(\hat{H},\hat{L},\hat{S}\right) \models (m,pc) : \texttt{return}$

iff $\hat{S}\,(m,pc) \sqsubseteq \hat{S}\,(m, Ret(m))$

References

[1] Gilles Barthe, Guillaume Dufay, Marieke Huisman, and Simão Melo de Sousa. Jakarta: A Toolset for Reasoning about JavaCard. In *Proc. e-SMART'01*, number 2140 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[2] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In *Proc. ESOP'01*, number 2028 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[3] Yves Bertot. Formalizing a JVML Verifier for Initialization in a Theorem Prover. In *Proc. CAV'01*, number 2102 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[4] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model ckecking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[5] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Proc. ESOP'04*, number 2986 in Lecture Notes in Computer Science, pages 385 – 400. Springer-Verlag, 2004.

[6] Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *Proc. of IEEE Int. Conference on Dependable Systems & Networks (DSN)*, 2002.

[7] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a Provably-Correct Implementation of the JVM Bytecode Verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.

[8] Xavier Leroy (coordinator). French ARC Concert research project : compilateurs certifiés. `http://www-sop.inria.fr/lemme/concert/`.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, 1977.

[10] Ewen Denney. The synthesis of a Java Card tokenisation algorithm. In *Proc. of 16th Int. Conf. on Automated Software Engineering (ASE 2001)*, pages 43–50. IEEE Press, 2001.

[11] Thomas Genet, Thomas Jensen, Vikash Kodati, and David Pichardie. A Java Card CAP converter in PVS. In *Proc. of 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2003)*, 2003.

[12] René Rydhof Hansen. Flow Logic for Carmel. Technical Report SECSAFE-IMM-001, Danish Technical University, 2002.

[13] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[14] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231. ACM Press, 2003.

[15] Renaud Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005, Trusted Logic SA, May 2001.

[16] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. In French.

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

[18] Hanne Riis Nielson and Flemming Nielson. Flow Logics for Constraint Based Analysis. In *Proc. CC'98*, number 1383 in Lecture Notes in Computer Science, pages 109–127. Springer-Verlag, 1998.

[19] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Proc. of the ACM SIGPLAN Workshop on ML*, pages 77–86, 1998.

[20] David Pichardie. Coq sources of the development. `http://www.irisa.fr/lande/pichardie/CarmelCoq/`.

[21] Frédéric Prost. *Interprétation de l'analyse statique en théorie des types*. PhD thesis, École normale supérieure de Lyon, 1999. In French.

[22] Igor Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Logic and Automated Reasoning*, 58(1–2):3–25, 2004.