Extracting a Data Flow Analyser in Constructive Logic

David Cachera Thomas Jensen David Pichardie Vlad Rusu

Abstract

We show how to formalise a constraint-based data flow analysis 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 expressed in an intermediate representation 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 correct, minimal solution to the constraints is constructive which means that the extraction mechanism of Coq provides a provably correct data flow analyser in OCAML. The library of lattices together with the intermediate 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.

This extended abstract is the short version of a paper presented at ESOP'04.

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 [CC77] 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, which allows to specify static analyses in a way that ensures their *well-formedness* while facilitating their *correctness proof*, and to extract a static analyser from the proof of existence of a correct program analysis result.

In this paper, we show how to specify a static analysis in the logical formalism underlying the proof assistant Coq. This formalism offers a strong type system for expressing correctness of specifications, together with a mechanism for compiling the specifications into the OCAML programming language. More precisely, we offer the following contributions:

- We show how to define a library of abstract domains of properties used in the analysis in a generic fashion (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.

- We show how to represent a constraint-based analysis in a constructive manner (Section 4) that allows at the same time to prove correctness of the analysis (Section 5) and 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 6).

We have chosen to develop this approach 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. Section 7 compares with other work on formalizing the correctness of data flow analyses, and Section 8 concludes. The Coq sources of the development are available at http://www.irisa.fr/lande/pichardie/CarmelCoq.

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 for which we (due to space restrictions) cannot offer a full introduction.

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 [Mar01] specified using the Flow Logic formalism [Han02] and proved correct on paper with respect to an operational semantics [Siv04]. 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 [Siv04] 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, and `invokevirtual` for calling a virtual method.

The rule (1) reads as follows: the instruction `push` c at address (m, pc) of state σ of the form $\langle\langle h, \langle m, pc, l, s\rangle :: sf\rangle\rangle$ has the effect of pushing c on the operand stack s of σ and advancing to the instruction at $pc + 1$.

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

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

The rule (2) is slightly more complicated. It reads: for M a method name, the instruction (`invokevirtual` M) at address (m, pc) of state $\sigma = \langle\langle h, f :: sf\rangle\rangle$ may only occur if 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 values V. The actual method that will be called is to be found in the object that resides in the heap h at the address $h(loc)$, and the actual parameters of that method are contained in the vector V. Then, the methodLookup function searches the class hierarchy for the method name M in that object, and returns the actual method to which the control will be transferred.

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 in the call stack after the virtual method invocation is not f any more, but a new frame f'' with a different operand stack. This is because the `invokevirtual` instruction has a *side effect*: the location loc and the vector V of actual parameters are removed from f (cf. rule (2) above).

This rather intricate behavior is what actually happens in the Java Card language. It made for the most challenging part of the proof of correctness of the analysis with respect to the semantics (we return to this point in Section 5). The analysis is now briefly described.

2.1 Carmel Flow Logic

The Carmel Flow Logic defined by Hansen [Han02] 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 the program. The *abstract state* domain $\widehat{\text{State}}_P$, defined by

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

contains an approximation of all possible heaps and, for each program point, an over-approximation of the value of each local variable and of the operand stack. The abstract domains are further described in Section 3.

The specification of the logic consists of a set of inference rules that for each Carmel instruction defines a constraint over an abstract state $\widehat{\Sigma} \in \widehat{\text{State}}$. In order 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 imposed by all the instructions of P. For example, if a `push` instruction is present at address (m, pc), the following constraints should be satisfied:

$$\widehat{\text{push}}\left(c, \hat{S}\,(m, pc)\right) \sqsubseteq \hat{S}\,(m, pc+1) \tag{3}$$

$$\hat{L}\,(m, pc) \sqsubseteq \hat{L}\,(m, pc+1) \tag{4}$$

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

3 Constructing abstract domains

In this section we define the type `(lattice A)` parameterised by the type of data manipulated in the lattice, and construct higher order functions which produce a lattice object from other lattice objects. This allows to construct the abstract domains (of local variables, stacks, *etc.*) in a compositional fashion starting from a collection of base abstract domains. This modular technique of combining and building lattices has the advantage 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 in the lattice structure do not invalidate the overall proof.

A lattice object is a structure with two families of fields : the functional fields which are objects used 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. Only the functional fields will appear in the OCAML code of the constructed analyser, but the properties in the logical fields are used to resolve proof obligations during the construction of the analyser.

The lattice type is conveniently defined as a record type in Coq, as shown in the following Coq declaration, where details for the `order` relation and the well-foundedness of the lattice are given.

```
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_prop :   ;;  join is a correct binary least upper-bound
  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)))
}
```

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 point to prove the termination of the final analyser.

Note that our definition of the `Lattice` type includes the ascending chain condition, thus, we are not defining a lattice in general. However, for convenience the term lattice is employed for such a structure in the rest of this document.

3.1 The lattice library

We present here the lattices that we have developed for our analysis. The lattices are built from two base lattices using four lattice constructors. These developments are largely analysis-independent and can be reused in other contexts.

Two base lattices are defined: the flat lattice of integer constants and the lattice of finite sets over a subset $\{0,\ldots,\texttt{max}\}$ of integers. Additionally, there are four functions to combine lattices:

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

The first two functions are the standard constructions of the cartesian product and the separate sum (disjoint union extended with a new top and bottom element) of two lattices. The `arrayLattice` function builds the type of arrays whose elements live in a lattice and whose size is bounded by a parameter `max`.

The array lattice frequently occurs in flow-sensitive analyses where the number of abstract variables depends on the program to analyse, and they 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 [OG98]. Details are left out for space reasons.

The fourth constructor `stackLattice` builds the lattice of stacks of elements of type `A`. In this lattice, stacks with different sizes are incomparable. In addition to the standard lattice operations, this lattice also carries abstract versions of the usual stack operations *pop*, *push* and *top*.

The lattice employed in our particular analysis is shown in Figure 1. The modular construction saves a considerable amount of time and effort, *e.g.*, compared to proving `acc_property` for the lattice in Figure 1 as a whole.

3.2 The constraint solver

The basic fixpoint operator `lfp` takes a monotone function $f : A \to A$ on a lattice L (parameterised by a type A) and computes the least element x of L verifying $f(x) = x$, which, by a corollary of the Tarski's Fixed Point Theorem, 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)
```

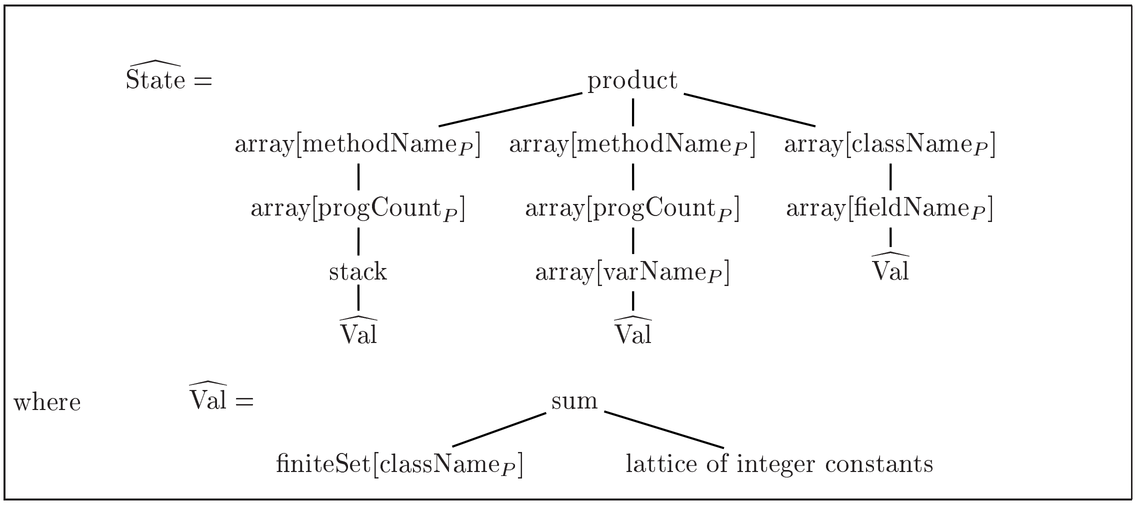

Figure 1: The lattice of abstract states (each xX$_P$ represents the number of distinct xX elements in program P).

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 then prove in Coq that this type is non-empty, which here consists in instantiating the existentially quantified `x` in the type definition by $\lim_{n\to\infty} f^n(\perp)$.

Then, the extraction mechanism of programs from proofs generates for `lfp` the following OCAML code, in which the purely logical objects of the proof (i.e., the chosen witness verifies the fixpoint equation) have 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
```

This operator is then used to solve a list of constraints in the following way. 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 $f_i(x) \sqsubseteq x \ \forall i \in \{1, \ldots, n\}$ 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.

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 definition of a set of constraints associated to each instruction.

2. The proof of correctness of these constraints wrt. the Carmel semantics.

3. The building of an analyser `analyse` that computes an abstract state verifying all the constraints generated for a given program.

Formally, we have to prove the two following theorems:

$$\forall P : \text{Program}, \widehat{\Sigma} : \widehat{\text{State}}_P.\ \texttt{verifyAllConstraints}_P(\widehat{\Sigma}) \Rightarrow [\![P]\!] \sim \widehat{\Sigma} \tag{5}$$

$$\forall P : \text{Program},\ \texttt{verifyAllConstraints}_P(\texttt{analyse}_P) \tag{6}$$

where $\llbracket P \rrbracket$ 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. These two theorems imply the correctness of the analyser:

$$\llbracket P \rrbracket \sim \texttt{analyse}_P$$

When formalising the analysis, several representations of the constraints are possible. It turns out that for the correctness proof (Phase 2) it is more convenient to emphasize the *order relation* aspect of the constraints, whereas the construction of an effective analyser (Phase 3) requires to emphasize the explicit *computational content* of the constraints. This is why an intermediate representation of constraints is defined in Phase 1, which allows for both interpretations and leaves room for reuse in other analyses. Sections 4, 5 and 6 describe the three phases.

4.1 An intermediate representation for constraints

To motivate the format chosen here for representing constraints, note that the constraints have three ingredients: a start adress ad_1 and an end address ad_2 of the data flow, the part of the state that is being affected, and the transformation F that is applied to the data that flows. For example, for the constraint (3) from Section 2.1 that corresponds to the instruction `push` c, 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

$$\begin{array}{lll} \texttt{type Constraint}_P = & | \ \texttt{S2S of Address * Address * } (\widehat{\texttt{Stack}}_P \to \widehat{\texttt{Stack}}_P) & \\ & | \ \texttt{L2L of Address * Address * } (\widehat{\texttt{LocalVar}}_P \to \widehat{\texttt{LocalVar}}_P) \texttt{...} & (7) \end{array}$$

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. A constraint, *i.e.*, an object of type $\texttt{Constraint}_P$, is given the following *relational* interpretation: $\mathcal{R}\llbracket\cdot\rrbracket : \texttt{constraint}_P \to \left(\widehat{\text{State}}_P \to \text{Prop}\right)$.

For the constructors `S2S` and `L2L` the relational interpretation is

$$\begin{array}{lcl} \mathcal{R}\llbracket(\texttt{S2S}\ \ ad_1\ \ ad_2\ \ F)\rrbracket(\hat{H}, \hat{L}, \hat{S}) & := & F(\hat{S}(ad_1)) \sqsubseteq \hat{S}(ad_2) \\ \mathcal{R}\llbracket(\texttt{L2L}\ \ ad_1\ \ ad_2\ \ F)\rrbracket(\hat{H}, \hat{L}, \hat{S}) & := & F(\hat{L}(ad_1)) \sqsubseteq \hat{L}(ad_2) \end{array}$$

4.2 Building the intermediate representation

As seen in Section 2.1 each instruction produces a set of constraints, *e.g.*, Formulas (3) and (4) for the `push` instruction. Thus, we define a function `Flow`, which for each instruction, returns the corresponding list of constraints in a syntax of the form (7).

However, defining this function is slightly complicated by the fact that the constraints make reference to actual program points of the form (m, pc) of the program P being analysed. In a paper-and-pencil proof, every instance of (m, pc) implicitly refers to a valid program point. 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 $\texttt{Flow}_P$ 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).

Continuing with the `push` instruction, the corresponding code part of $\texttt{Flow}_P$ is:

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

We now define what it means for an abstract state $\widehat{\Sigma}$ to verify all the constraints of a program P. With Addr_P denoting the set of addresses appearing in P, and $\mathcal{R}\llbracket\cdot\rrbracket$ the relational interpretation of constraints defined in Section 4.1:

$$\texttt{verifyAllConstraints}_P(\widehat{\Sigma}): \quad \forall (m, pc) : \mathrm{Addr}_P . \forall c \in \texttt{Flow}_P(m, pc).\ \mathcal{R}[\![c]\!](\widehat{\Sigma})$$

5 Correctness

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 (as stated in Theorem (5)). This implication 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

Let $addr(\sigma)$ denote the address (m, pc) of a state σ (cf. Section 2), and

$$\sigma \vdash_P \widehat{\Sigma} : \quad \forall c \in \texttt{Flow}_P(addr(\sigma)).\ \mathcal{R}[\![c]\!](\widehat{\Sigma}) \tag{8}$$

the predicate expressing the fact that the abstract state $\widehat{\Sigma}$ satisfies all the constraints at address $addr(\sigma)$ of program P. Moreover, 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 an arbitrary Carmel instruction I, except `return`, is of the form (9) defined below.

$$\forall P : \mathrm{Program}, \forall \sigma \in [\![P]\!], \forall \widehat{\Sigma} : \widehat{\mathrm{State}}_P, \sigma \sim \widehat{\Sigma} \wedge \sigma \vdash_P \widehat{\Sigma} \Rightarrow \left(\forall \sigma' : \mathrm{State}, \sigma \rightarrow_I \sigma' \Rightarrow \sigma' \sim \widehat{\Sigma} \right) \tag{9}$$

That is, if a state σ is approximated by an abstract state $\widehat{\Sigma}$ that satisfies all the constraints of σ, and if, by performing instruction I, the state σ evolves into σ', then σ' is approximated by $\widehat{\Sigma}$ as well.

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

Formula (9) 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 **and** $\widehat{\Sigma}$ satisfies the constraints corresponding to σ, **then** $\widehat{\Sigma}$ also correctly approximates all *immediate* successors σ' of σ by an instruction.

However, this is not true for the `return` instruction. This is because every `return` is preceded somewhere along the execution by an `invokevirtual` instruction, and, as seen in Section 2, `invokevirtual` not only *pushes* a new frame f' on the call stack, but also makes a *side effect* on the operand stack of (the formerly first) frame f. Hence, the operand stack of frame f, as it was before the call, is not available any more when the `return` instruction occurs.

But, to prove correctness of the analysis, this information is essential. The information must be fetched from the state σ'' that *precedes* the `invokevirtual` call (responsible of the loss of the said information) that could have occurred arbitrarily far in the past. Hence, a *well-founded induction* argument is required. The proof of the resulting theorem is substantially more involved than the proofs of Theorems (9). It should be pointed out that this difficulty had been avoided by a previous pencil-and-paper proof [Han02], where the semantics of the `invokevirtual` instruction has been modified to avoid the side-effect on the call stack. This side-effect is the source of most difficulties in the correctness proof.

6 Resolution

Proofs in the `Coq` system are constructive. Via the Curry-Howard isomorphism, they thus may be seen as programs in a functional language benefiting from a rich type system. The program extraction mechanism provides a tool for automatic translation of these proofs into a functional language with a less rich type system, namely OCAML. The translation removes those parts of the proof that do not contribute to the actual construction of the result but are only concerned with proving that it satisfies its specification

The proof-as-programs 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 approach is conceptually simple and appealing, the development of any non-trivial piece of software rather tends to be done by defining (most of) the function and then showing that it is indeed a witness to the theorem. In our case, this means that the type of the analyser function does not directly encode its correctness. Rather, correctness is proved after the definition of the function. This technique is favoured because it is simpler to control the efficiency of the resulting program.

6.1 Construction of the analyser

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

$$(P : \text{Program}) \to \widehat{\text{State}}_P$$

which must verify Theorem (6). The most difficult part of the work has already been done during the definition of the generic solver `lfp_list` described in Section 3.2, i.e. proof of termination and correctness. The constraints are collected from the lists defined by $\texttt{Flow}_P$ (cf. Section 4) and translated into a *functional form*, using a mapping

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

The constraint solver is then defined by instantiating `lfp_list`:

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

where $\widehat{\text{State}}_P$ denotes the particular abstract state lattice of the analysis (depicted in Figure 1), `collect_func` the list of functional constraints (obtained by composing $\texttt{Flow}_P$ and $\mathcal{F}[\![\cdot]\!]$) and `collect_func_monotone` a proof that theses constraints are monotonic. Details about proofs of monotonicity are available in the full version of this paper. These properties of `lfp_list` ensure that $\texttt{analyse}_P$ is the least abstract state $\widehat{\Sigma}$ in $\widehat{\text{State}}_P$ verifying

$$\forall f \in \texttt{collect\_func}_P \quad f(\widehat{\Sigma}) \sqsubseteq \widehat{\Sigma} \tag{10}$$

To finish making the link between this result and the correctness of $\texttt{analyse}_P$ as defined by properties (6), we prove the equivalence between relational and functional interpretation of constraints:

$$\forall c : \texttt{Constraint}_P, \widehat{\Sigma} : \widehat{\text{State}}_P \ \mathcal{F}[\![c]\!](\widehat{\Sigma}) \sqsubseteq \widehat{\Sigma} \Leftrightarrow \mathcal{R}[\![c]\!]\widehat{\Sigma} \tag{11}$$

Indeed, (10) deals with the functional interpretation, but the predicate `verifyAllConstraints` used in the specification of the analyser interprets constraints as relations.

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 (`S2S`, `L2L`,...). As a consequence, modifications to the system of constraints only affect proofs made during Sections 4 and 5, rather than the construction and the correctness of the analyser.

7 Related work

Proving correctness of program analyses is one of the main applications of the theory of abstract interpretation [CC77]. 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 intepretation with a proof assistant is, as far as we know, that of Monniaux [Mon98] who has built a Coq theory of Galois connections. Prost in his thesis [Pro99] 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.

Several works on mechanical verification of program analyses have dealt with the Java byte code verifier. Barthe *et al.* [BDJ$^+$01] 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 [BDHdS01], 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 [Ber01] 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 [CGQ98], 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 [KN02] have proved the correctness of a Java byte code verifier using the proof assistant Isabelle/HOL. In particular their work include 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.

All these byte code verifiers deal 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 have cast our development in the general setting of Flow Logic [NN98] and constraint-based analysis. The Carmel Flow Logic analysis specified by Hansen also covers exceptions which means it is straightforward to extend our analyser to determine data flow arising from exceptions. It is not evident how such an extension would be done in the formal byte code verifier frameworks cited above.

8 Conclusion

In this paper, we have shown 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 eliminates the gap that often exists between a paper-specification of an analysis and the analyser that is actually implemented. Our approach applies to any analysis 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.

Concerning **efficiency**, the extracted analyser performs well, taking only a few seconds to analyse 1000 lines of bytecode. The resulting extracted program is about 2000 lines of OCAML code. The extracted version of `analyse` have now a type Program $\to \widehat{\text{State}}$ because OCAML does not have dependent types. Future work will proceed in two directions. First, it would be desirable to be able to extract a variety of sophisticated constraint solvers from proofs using the general lattice theory. Whether such detailed programming is possible at the proof level remains to be seen. Second, further automation of the proof obligations in the above-described procedure for building analysers is necessary.

References

[BDHdS01] Gilles Barthe, Guillaume Dufay, Marieke Huisman, and Simão Melo de Sousa. Jakarta: A Toolset for Reasoning about JavaCard. *Lecture Notes in Computer Science*, 2140, 2001.

[BDJ+01] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. *Lecture Notes in Computer Science*, 2028, 2001.

[Ber01] Yves Bertot. Formalizing a JVML Verifier for Initialization in a Theorem Prover. *Lecture Notes in Computer Science*, 2102, 2001.

[CC77] 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.

[CGQ98] 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.

[Han02] René Rydhof Hansen. Flow Logic for Carmel. Technical Report SECSAFE-IMM-001, Danish Technical University, 2002.

[KN02] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[Mar01] Renaud Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005, Trusted Logic SA, May 2001.

[Mon98] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. In French.

[NN98] 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.

[OG98] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[Pro99] Frédéric Prost. *Interprétation de l'analyse statique en théorie des types.* PhD thesis, École normale supérieure de Lyon, 1999. In French.

[Siv04] Igor Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Logic and Automated Reasoning*, 2004. To appear.