Tool-Assisted Specification and Verification of Typed Low-Level Languages

Gilles Barthe (gilles.barthe@sophia.inria.fr)
INRIA Sophia Antipolis, France

Pierre Courtieu (pierre.courtieu@univ-orleans.fr)
Université d'Orléans, France

Guillaume Dufay (gdufay@site.uottawa.ca)
University of Ottawa, Canada

Simão Melo de Sousa (desousa@di.ubi.pt)
Universidade da Beira Interior, Portugal

Abstract. Bytecode verification is one of the key security functions of several architectures for mobile and embedded code, including Java, Java Card, and .NET. Over the last few years, its formal correctness has been studied extensively by academia and industry, using general purpose theorem provers. The objective of our work is to facilitate such endeavors by providing a dedicated environment for establishing the correctness of bytecode verification within a proof assistant.

The environment, called Jakarta, exploits a methodology that casts the correctness of bytecode verification relatively to a defensive virtual machine that performs checks at run-time, and an offensive one that does not, and can be summarized as stating that the two machines coincide on programs that pass bytecode verification. Such a methodology has been used successfully to prove the correctness of the Java Card bytecode verifier, and may potentially be applied to many other similar problems. One definite advantage of the methodology is that it is amenable to automation. Indeed, Jakarta automates the construction of an offensive virtual machine and a bytecode verifier from a defensive machine, and the proofs of correctness of the bytecode verifier.

We illustrate the principles of Jakarta on a simple low-level language extended with subroutines, and discuss its usefulness to proving the correctness of the Java Card platform.

1. Introduction

1.1. Background

Virtual machines, such as the Java Virtual Machine, Java Card Virtual Machine, or Microsoft .NET, provide a means to ensure two key properties for mobile code: portability, because executing applets on a virtual machine abstracts away from the specifics of the underlying hardware and operating system; and security, because the virtual machine controls the interaction between the applet and its environment and hence reduces the risk that malicious applets may perform a security attack. In order to enforce security, such architectures rely on several mechanisms, known as security functions. One such crucial security function is the bytecode verifier, which performs a static analysis on programs and rejects potentially insecure programs.

Over the last few years, a number of projects have been targeted at proving formally that bytecode verification algorithms are correct, in the sense that they reject programs which may go wrong with respect to a given property P, typically a safety or security property such as type soundness or correct initialization or correct use of subroutines. These projects have been carried by with different aims, for example: (1) demonstrating that machine-checked semantics of realistic programming languages is feasible, and useful to detect ambiguities, incoherences or even flaws in specifications; (2) achieving a high level certification in the context of security evaluations according to international evaluation schemes for the security of IT products, such as the Common Criteria, which features seven evaluation assurance levels (EAL), the highest of which (levels EAL5 to EAL7) impose the use of formal methods for the modeling, specification and verification of the product being certified.

While such projects have been very successful in their endeavors, providing machine-checked accounts of complex low-level languages is labour-intensive, and could benefit from automated tool support for the most mundane tasks in developing machine-checked specifications and verifications of these languages. The main goal of this paper is to describe Jakarta, a dedicated environment for specifying and verifying formally bytecode verifiers. More precisely, the objective of Jakarta is to provide automated tool support for building and proving the correctness of a bytecode verifier enforcing a given property P. To this end, Jakarta builds upon a two-phase methodology that is common to many existing works, and which is briefly described below.

The first phase of the methodology, which we coin the Virtual Machine phase or VM phase for short, deals with cross-machine validation. It involves defining three virtual machines:

– a *defensive* virtual machine enforcing the property P at run time through a type system;

– an *typed* (or *abstract*) virtual machine, whose execution only manipulates type information related to P (without the usual values of the virtual machines), that will be used to build the bytecode verifier;

– an *offensive* virtual machine that does not verify P at runtime (hence faster and lighter) and that relies on the successful bytecode verification of a program to enforce the P-error-free execution of that program;

and showing that the virtual machines verify the following cross-validation property: *the offensive and defensive virtual machine coincide on those programs that are accepted by the typed virtual machine.* In the next section, we make such notions as "coincide", or "accepted" precise, and illustrate cross-machine validation on a small example that involves a toy assembly language, extended with subroutines. One important feature of the VM phase is that it does not handle exceptions, nor instructions to invoke or return from a method; both of which are handled in the second phase.

The second phase of the methodology, the FrameWork phase or FW phase for short, aims to construct a bytecode verifier from the typed virtual machine, and to establish its correctness relying on cross-machine validation. In order to construct the bytecode verifier, one formalizes Kildall's algorithm, a well-known dataflow analysis that computes fixpoints of monotone functions over lattices, and instantiate the algorithm to the typed virtual machine defined during the VM phase. The correctness of the resulting bytecode verifier is then shown by proving that *the offensive and defensive virtual machine coincide on those programs that are accepted by the bytecode verifier.* This statement is derived from the correctness of Kildall's algorithm, from the correctness of cross-machine validation, and from some properties about method invocations and returns, and exceptions. The latter is discussed in the running example of the next section. As observed by Nipkow [38], the FW phase may be applied to a large variety of bytecode verifiers, including the standard verifiers for Java, Java Card, and .NET, but also enhanced verifiers that guarantee stronger security properties, for example with respect to information flow or resource control.

1.2. Jakarta

The methodology outlined above provides us with clear guidelines for formally verifying bytecode verifiers. Furthermore, the brief description of the methodology indicates that the two phases are of a rather different nature: the VM phase is specific to the execution platform under consideration, while in contrast the FW phase is generic. In fact, the VM phase represents most of the effort required to establish the correctness of a given bytecode verifier. It is also the place where one can achieve a high level of automation by developing appropriate methods and tools, and where the pay-off is highest.

The purpose of Jakarta is to provide tool support for the VM phase, and in particular to provide a very high-level of automation for the mundane tasks inherent to the VM phase. Specifically, Jakarta is designed to generate by *abstraction* from the specification of the defensive machine both the specifications of the offensive machine and of the typed machine, and the cross-validation proofs for these machines. We detail this process to some extent below.

The user provides the specification of the defensive machine. He can write it in several languages: Caml, Coq or the dedicated language of Jakarta, the *Jakarta Specification Language JSL*, a typed language whose execution model is based on conditional rewriting, defined in Section 3. The specification is translated to JSL if necessary before the abstraction takes place. JSL complies with three crucial design objectives: firstly, it is sufficiently expressive so that virtual machines can be conveniently specified. Secondly, it lends itself well to automatic transformations, and in particular to abstractions as required by cross-machine validation. Thirdly, specifications can be translated automatically, via the *Jakarta Prover Interface JPI*, to several proof assistants, automatic theorem provers, prototyping environments, and programming languages. As mentioned above, translation from (subsets of) ML and Coq to JSL are implemented, but we also support translations to Elan, Spike, and PVS.

Then JSL specifications may be abstracted automatically, using an abstraction script that contains the JSL definition of the abstraction functions and the *abstraction commands*. The former specifies how to transform defensive states into target states, while the latter allow users to specify how the abstraction must be conducted, e.g. which functions must be abstracted, which expressions should be removed or modified, and more generally which modifications must be made to the generic abstraction mechanism in order to generate the expected abstracted machine. The abstraction is performed by the *Jakarta Transformation*

Kit JTK. We use the JTK twice, to derive the offensive and the typed machines from the defensive one.

Finally, JSL specifications may be cross-validated in a theorem prover (Coq currently), through a procedure that generates the statement of auxiliary commutation lemmas needed by the cross-validation proofs (Figures 2 and 3 in Section 2.2), as well as the corresponding proof scripts that use purpose-built tactics to discharge these lemmas. As shown in Figure 1, the cross validation is composed of three parts, all in the theorem prover format: the specifications of the defensive and target machines and the statements and proofs of commutation lemmas. The cross-validation is performed by the *Jakarta Automation Kit JAK*. A specific part of the abstraction script gathers all proof information required for this purpose. We use the JAK twice, to cross-validate the offensive and the typed machines against the defensive one.

Figure 1 synthesizes the process of completing the VM phase with Jakarta, using the methodology described above. Coq proof scripts are generated, but the methodology is easily applicable to other provers, although not implemented yet (proof script generation must be implemented for each prover).

In the course of the paper, we shall illustrate the principles of Jakarta, and demonstrate its usefulness on two case studies. The first case study is a toy language which we use as a running example. The second case study is a realistic programming language, namely Java Card; launched in 1996 by Sun, Java Card [27] is the standard programming language for new generation smartcards; its run-time environment, the Java Card Runtime Environment JCRE, comprises the Java Card Virtual Machine JCVM, that is very close to the Java Virtual Machine JVM. In both cases, the property P to be verified is well-typing, and we use the Jakarta environment to perform the first phase of the methodology. The results are positive: almost all proof obligations are discharged automatically, and a big part of the specification is also automatically generated (offensive and typed virtual machines, see next section).

Outline. This paper is organized as follows. Section 2 presents concepts and methodology through a small case study. Section 3 is devoted to the presentation of Jakarta Specification Language (JSL). Sections 4 and 5 present Jakarta tools to abstract JSL specifications and prove the correctness of the generated abstraction. Section 6 is devoted to a realistic and more complex case study with the Java Card Virtual Machine. Section 7 concludes with related works and perspectives.

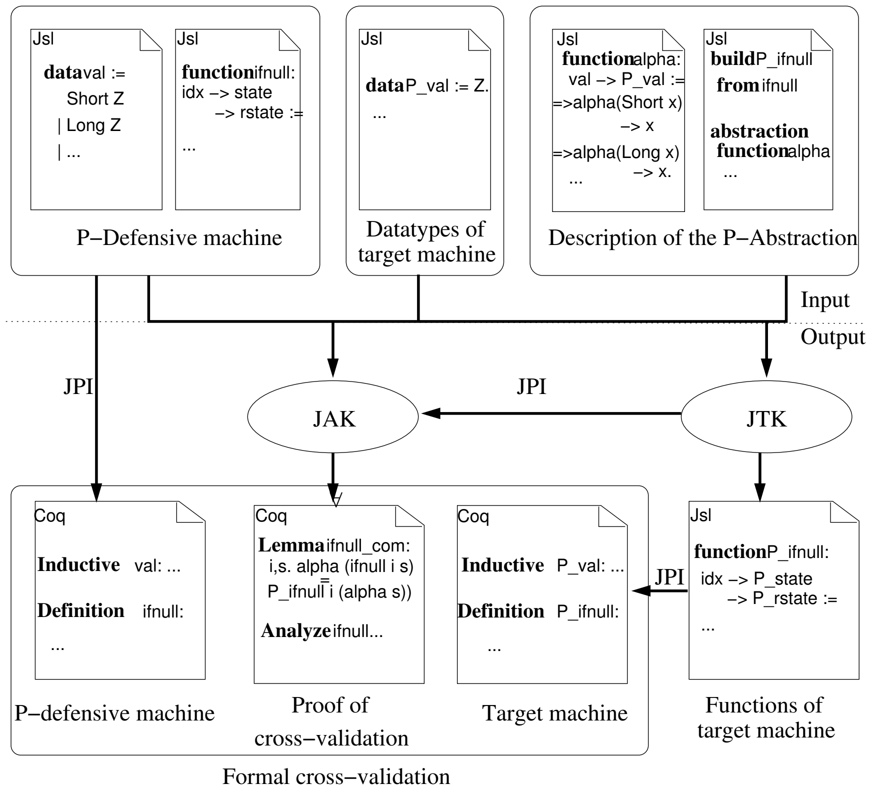

Figure 1. The methodology of Jakarta

2. Running example

We illustrate cross-machine validation on a small example that involves a toy assembly language, extended with subroutines. The assembly language is designed as a target for a simple imperative language with loops and conditionals; we extend it with subroutines [42] in order to make apparent some of the difficulties arising in the formal verification of realistic execution platforms. The goal of this section is to illustrate the methodology and the results expected from Jakarta while following sections will present Jakarta and how it can be used to obtain these results.

2.1. Formalism

The example is written in an intuitive syntax close to Coq with ML style record notations for readability. In fact the syntax is an intermediate format called JIR used to compile Jakarta specifications to proof assistants, see Section 3.3.

The keyword **`Inductive`** (resp. **`Mutual Inductive`**) introduces a inductive (resp. mutual inductive) type. Such definitions also declare all constructors for these types. We give below the definitions of the natural numbers `nat` (with the constructors `O` and `succ`, for successor), of parameterized (polymorphic) lists `list`, and of the parameterized `option` type commonly used to formalize non total functions (lift monad):

```
Inductive nat : Set :=
| O : nat
| succ : nat → nat.

Inductive list (A:Set) : Set :=
| Nil : list A
| Cons : A → list A → list A.

Inductive option (A:Set) : Set :=
| None : option A
| Some : A → option A.
```

The partial functions `head`, `tail` and `nth_elt` respectively return for a given list the first element, the list without its first element and the n-th element of that list.

Records, introduced by keyword **`Record`**, are represented internally as inductive types with one constructor. Accessors of these fields are then expressed as functions named by the corresponding fields of the record. **`match...with...end`** introduces pattern matching as in ML languages.

The general form of a *product* type is the universal quantification: **`forall`** `x:T,U`. In the case of a non dependent product, we use the usual arrow notation: `T → U`.

Convention. In the following, we will introduce datatypes and functions for the defensive virtual machine and the corresponding datatypes and functions for the offensive and typed virtual machines. In order to differentiate these definitions, we use the following conventions: all datatypes and functions concerned with the defensive, offensive and abstract typed virtual machines are prefixed by an `d_`, `o_` and `a_` respectively. In Coq, we rely on qualified names but using them here would clutter notations.

2.2. The Virtual Machine phase

2.2.1. *Language definition*

We formalize programs as lists of instructions, taken from an inductively defined instruction set. In order to factor definitions as much as possible, some instructions are parameterized by the type `vm_types` of the virtual machine types (integers and return addresses).

```
Inductive vm_type : Set :=  tInt : vm_type | tRA : vm_type.

Definition pcs := nat.

Definition locvars_idx := nat.

Inductive instr : Set :=
| ipush : Z → instr
| pop   : instr
| load  : vm_type → locvars_idx → instr
| store : vm_type → locvars_idx → instr
| iadd  : instr
| iif   : pcs → instr
| goto  : pcs → instr
| jsr   : pcs → instr
| ret   : locvars_idx → instr.

Definition program := (list instr).
```

For readability reasons, we have defined two synonyms `pcs` and `locvars_idx` for `nat` used respectively when the value of this type represents a program counter and an index into the list of local variable of a program state (see next Section).

2.2.2. *Defensive semantics*

The defensive virtual machine manipulates typed values (one constructor for each type of the virtual machine), from which the memory model (an operand stack, local variables and a program counter) is built.

```
Inductive d_val : Set :=
| d_Int : Z → d_val
| d_RA  : pcs → d_val.

Record d_state : Set := {
  d_opstack : (list d_val);
  d_locvars : (list d_val);
  d_pc      : pcs
}.
```

Our operational semantics is executable and models one step execution of the virtual machine. In order to account for abnormal termination, the execution function returns a tagged return state.

```
Inductive eReason : Set :=
| program_error : eReason
| opstack_error : eReason
| locvar_error  : eReason
| type_error    : eReason.

Inductive d_rstate : Set :=
| d_Normal   : d_state → d_rstate
| d_Abnormal : eReason → d_rstate.
```

Then, defensive execution is modeled as a function from states to return states, and is implicitly parameterized by a program `prg`, where each function in the right hand side of a match provides the defensive semantics of the corresponding bytecode.

```
Definition d_exec (s : d_state) : d_rstate :=
  match Nth_elt prg (d_pc s) with
  | Some i ⇒
    match i with
    | ipush z   ⇒ d_IPUSH z s
    | pop       ⇒ d_POP s
    | load t l  ⇒ d_LOAD t l s
    | store t l ⇒ d_STORE t l s
    | iadd      ⇒ d_IADD s
    | iif p     ⇒ d_IIF p s
    | goto p    ⇒ d_GOTO p s
    | jsr p     ⇒ d_JSR p s
    | ret l     ⇒ d_RET l s
    end
  | None ⇒ d_Abnormal program_error
  end.
```

In order to illustrate some of the issues involved in the abstraction, we introduce the semantics of the bytecodes `iif` and `ret`, for which we shall later provide an offensive and a typed semantics. The (defensive) semantics of the `iif` is given by the function `d_IIF` that compares the first element of the operand stack to zero and branches accordingly to the program counter given as a parameter or to the next instruction.

```
Definition d_IIF (p : pcs) (s : d_state) :=
  match head (d_opstack s) with
  | Some (d_Int z) ⇒
      match Zeq_bool z 0 with
      | true  ⇒ d_Normal (Mk_d_state (tail (d_opstack s))
                                     (d_locvars s) p)
      | false ⇒ d_Normal (Mk_d_state (tail (d_opstack s))
                                     (d_locvars s)
```

```
                                            (succ (d_pc s)))
      end
  | Some (d_Ra n) ⇒ d_Abnormal type_error
  | None ⇒ d_Abnormal opstack_error
  end.
```

The semantics of the `ret` is given by the function `d_RET` that branches to the return address located in the local variables at the position given as a parameter.

```
Definition d_RET (l : locvars_idx) (s : d_state) :=
  match Nth_elt (d_locvars s) l with
  | Some (d_Ra n) ⇒ d_Normal (Mk_d_state (d_opstack s)
                                          (d_locvars s) n)
  | Some (d_Int z) ⇒ d_Abnormal type_error
  | None ⇒ d_Abnormal locvar_error
  end.
```

2.2.3. *Offensive semantics*

We now turn to the definition of the offensive semantics. The offensive virtual machine manipulates untyped values, and hence the memory model must be modified accordingly. It is sufficient to modify the definition of values so that the definitions of states and return states are modified appropriately. (Note that the offensive virtual machine does not raise any type error hence the reasons for abnormal termination could be modified accordingly; however, doing so would complicate the statements about cross-validation.)

```
Definition o_val := Z.

Record o_state : Set := {
  o_opstack : (list o_val);
  o_locvars : (list o_val);
  o_pc      : pcs
}.

Inductive o_rstate : Set :=
| o_Normal   : o_state → o_rstate
| o_Abnormal : eReason → o_rstate.
```

We now turn to the offensive semantics of `IIF` and `RET`.

```
Definition o_IIF (p : pcs) (s : o_state) :=
  match head (o_opstack s) with
  | Some x ⇒
    match Zeq_bool x 0 with
    | true  ⇒ o_Normal (mk_o_state (tail (o_opstack s))
                                    (o_locvars s) p)
    | false ⇒ o_Normal (mk_o_state (tail (o_opstack s))
                                    (o_locvars s) (succ (o_pc s)))
    end
  | None ⇒ o_Abnormal opstack_error
  end.

Definition o_RET (l : locvars_idx) (s : o_state) :=
  match Nth_elt (o_locvars s) l with
  | Some x ⇒ o_Normal (mk_o_state (o_opstack s)
                                  (o_locvars s) (z2n x))
  | None   ⇒ o_Abnormal locvar_error
  end.
```

Type verifications have disappeared in these definitions. Furthermore, observe that in order for the definition to be type correct, we must insert a coercion `z2n` (absolute value) from integers to natural numbers in the first branch for RET. Since in a correct situation the stack contains a positive integer when RET is called (certifying this is one of the goals of the BCV), the use of `z2n` will not make the offensive RET wrong (since it must commute with the original function only in type safe situations).

2.2.4. *Typed semantics*

We now turn to the definition of the typed semantics. The typed virtual machine manipulates types as values. However, the definition of typed values must keep the value for return addresses (known statically) in order to be able to handle control flow for subroutine. Likewise the offensive machine, the new definition of values is directly taken into account into the definitions of states and return states.

```
Inductive a_val : Set :=
| a_Int : a_val
| a_RA : pcs → a_val.

Record a_state : Set := {
  a_opstack : (list a_val);
  a_locvars : (list a_val);
  a_pc      : pcs
}.

Inductive a_rstate : Set :=
| a_Normal   : a_state → a_rstate
| a_Abnormal : eReason → a_rstate.
```

We now turn to the defensive semantics of IIF and RET.

```
Definition a_IIF (p : pcs) (s : a_state) :=
  match head (a_opstack s) with
  | Some a_Int ⇒
      Cons (a_Normal (mk_a_state (tail (a_opstack s))
                                (a_locvars s) p))
        (Cons (a_Normal (mk_a_state (tail (a_opstack s))
                                        (a_locvars s)
                                        (succ (a_pc s))))
          Nil)
  | Some (a_RA n) ⇒ Cons (a_Abnormal type_error) Nil
  | None ⇒  Cons (a_Abnormal opstack_error) Nil
  end.

Definition a_RET (l : locvars_idx) (s : a_state) :=
  match Nth_elt (a_locvars s) l with
  | Some (a_RA n) ⇒ a_Normal (a_mk_state (a_opstack s)
                                          (a_locvars s)
                                          (a_pc s))
  | Some a_Int ⇒ a_Abnormal type_error
  | None ⇒ a_Abnormal locvar_error
  end.
```

Note that the semantics of aIIF returns a list of return states rather than a return state, as a consequence of the typed semantics being unable to perform tests for conditionals. For aexec to be well typed, this modification should be propagated to other bytecodes.

2.2.5. *Cross-validation*

The objective of cross-validation is to establish that the offensive and defensive machines coincide on programs that do not go wrong with the typed virtual machine. Establishing this result involves:

– showing that the offensive and defensive machines coincide on programs that do not raise a type error when executed on the defensive machine;

– showing that every program that raises a type error when executed on the defensive machine also raises a typing error when executed with the typed machine.

These two statements can be proven by showing that the offensive and typed machines are sound abstractions of the defensive machine. Formally, we define two abstraction functions alpha_do_val and alpha_da_val from defensive to offensive values, and from defensive to typed values respectively:

```
Definition alpha_do_val (dv : d_val) : o_val :=
  match dv with
  | d_Int v ⇒ v
  | d_RA n ⇒ n2z n
  end.

Definition alpha_da_val (dv : d_val) : a_val :=
  match dv with
  | d_Int v ⇒ a_Int
  | d_RA n ⇒ a_RA n
  end.
```

where n2z coerces type nat to z. These functions are then extended to states (alpha_do and alpha_da), and to return states (alpha_do_rs and alpha_da_rs):

```
alpha_do    : d_state  →  o_state
alpha_da    : d_state  →  a_state
alpha_do_rs : d_rstate → o_rstate
alpha_da_rs : d_rstate → a_rstate
```

Then cross-validation involves proving that the diagrams of Figures 2 and 3 commute. In the first diagram, one has to assume that the program does not raise a type error when executed with the defensive machine. In the second diagram, the arrow on the right-hand side means that the abstraction of the return defensive state is an element of the list of return typed states.

$$\begin{tikzcd}
\texttt{d\_state} \arrow[r, "\texttt{d\_exec}"] \arrow[d, "\texttt{alpha\_do}"] & \texttt{d\_rstate} \arrow[d, "\texttt{alpha\_do\_rs}"] \\
\texttt{o\_state} \arrow[r, "\texttt{o\_exec}"] & \texttt{o\_rstate}
\end{tikzcd}$$

Figure 2. Commutative diagram of defensive and offensive execution

$$\begin{tikzcd}
\texttt{d\_state} \arrow[r, "\texttt{d\_exec}"] \arrow[d, "\texttt{alpha\_da}"] & \texttt{d\_rstate} \arrow[d, hook, "\texttt{alpha\_da\_rs}"] \\
\texttt{a\_state} \arrow[r, "\texttt{a\_exec}"] & \texttt{(list a\_rstate)}
\end{tikzcd}$$

Figure 3. Commutative diagram of defensive and typed execution

The commutativity of both diagrams are stated in Coq by the formula:

```
  ∀ (s:d_state), d_exec s ≠ type_error →
                alpha_do (d_exec s) = o_exec (alpha_do s)    (1)
```

for the defensive and offensive virtual machines and by the formula:

```
  ∀ (s:d_state),
    In (alpha_da_rstate (d_exec s)) (a_exec (alpha_da s))    (2)
```

for the defensive and abstract virtual machines. The notion of programs on which the functions d_exec, o_exec and a_exec rely is left as a global implicit parameter of our definitions. Both formulae are proved by a case analysis on the bytecode to be executed. In order to keep proofs manageable, we prove a commutation lemma for each bytecode. In the case of the IIF bytecode, the statements of the commutation lemmas are as follows:

```
Lemma IIF_eq_do : ∀ (p : pcs) (s : d_state),
  d_IIF p s ≠ d_Abnormal type_error →
  o_IIF p (alpha_do s) = alpha_do_rs (d_IIF p s).

Lemma IIF_eq_da : ∀ (p : pcs) (s : d_state),
   In (alpha_da_rs (d_IIF p s)) (a_IIF p (alpha_da s)).
```

where In is list membership. As with all other bytecodes, the proof proceeds by a case analysis on the state of the defensive virtual machine and finally by equational reasoning; see Section 5.4 for a proof of the Lemma IIF_eq_do that uses tactics developed for the purpose of automatic cross-validation proofs.

2.3. The Framework phase

The FW phase aims at building a bytecode verifier from the results obtained during the VM phase. It relies on the formalization, described in full details in [5], of a dataflow analysis for a generic execution function that meets minimal requirements. This dataflow analysis is proven correct in the sense that it offers a sound decision procedure to reject programs that may go wrong during execution. The FW phase instantiates the dataflow analysis to build the bytecode verifier for the typed and then for defensive virtual machine. For the latter, results of cross-validation are used to establish the soundness of the bytecode verifier.

2.3.1. *Abstract definition and construction*

The formalization of the bytecode verifier as described in [5] relies on the modules system of Coq. It offers a refined model of the various notions (transition system, fixpoint structure, bytecode verifier, abstract virtual machine, etc.) involved to obtain the bytecode verifier for the defensive virtual machine. For the sake of this paper, we will not introduce the corresponding modules but only focus on the definition of a generic bytecode verifier, that abstracts away from implementation details.

DEFINITION 2.3.1. *A* bytecode verifier *is given by a type* `state` *of* states*, an* execution relation `exec` *over states, a set* `err` *of error states and a predicate* `check` *such as:*

```
∀ a:state, (check a) → ¬(bad a).
```

where a state is `bad`*, if it is possible to reach from it an error state by successive transitions of the execution relation. Thus the predicate* `check` *rejects all states that lead to an error state.*

The standard way to build such a bytecode verifier is to endow the type of states with a well-founded order for which execution is decreasing (to guarantee termination), and such that error states are downwards closed. If furthermore execution is deterministic, one can compute for every state `a`, the greatest fixpoint `b` below `a`. To do so, we define for every state `a` the greatest fixpoint `gfp a` below it as:

$$\texttt{gfp a} = \begin{cases} \texttt{a} & \texttt{if exec a = a} \\ \texttt{gfp (exec a)} & \texttt{otherwise} \end{cases}$$

Then, we define `check a` as `¬(err (gfp a))`. As execution is monotone and `gfp a` is the greatest fixpoint below `a`, it is clear that such a checking is sufficient to guarantee that `a` is not a bad state.

2.3.2. *Kildall's algorithm*

Kildall's algorithm is a generic algorithm to compute solutions of data-flow problems. In order to construct a bytecode verifier for a virtual machine, we instantiate the above construction, more precisely the `exec` function, to a function that performs one step of Kildall's algorithm. This algorithm relies on a *history structure*, that will correspond to the notion of states in the above construction, to store the last program state(s) reached for each program point. Depending on the desired accuracy (see e.g. [32] for a survey), the history structure may store one program state (monovariant analysis, that only accepts programs with monomorphic subroutines) or a set of program states (polyvariant analysis) for each program point.

For bytecode verification, the algorithm is applied method by method and the history structure is initialized to the initial state of the method being verified for the first program point, and to a default state for the other program points. One step of execution proceeds by iterating the execution function of the virtual machine over the states of the history structure. Each non-default state is chosen once and the result of the execution function of the virtual machine on this state is propagated to its possible successors in the history structure.

Propagated result states are *merged* with the previous state stored into the history structure at the corresponding program point and stored back at this location. If the history structure contains a default state then the result of the merge is the propagated state itself. Otherwise, the merged state is obtained from the propagated state and the state from the history structure, by taking pointwise the most general unifier (on the type lattice of the virtual machine) of the types appearing in the two states. In particular, if these two states do not have the same number of local variables or the same number of elements in the operand stack, the resulting merged state is an error state.

2.4. Adding method invocations and returns and exceptions

The running example above deals with a simple language without method invocations and method returns, and without exceptions. The purpose of this section is to explain how the methodology is adapted to handle these features. We focus on method invocations and returns; exceptions are treated in a similar manner. Note that the validation between the defensive and offensive virtual machines can be handled with the methodology described above and that only the validation between the defensive and typed virtual machines need to be cared for.

Suppose that we extend our language with instructions for invoking a method and returning from it. In this setting, a method is defined as a record consisting of list of instructions, i.e. its body, its number of local variables and its number of parameters. Due to the extreme simplicity of the type system, the arguments and result type of the methods are all of type `tInt`. Then, a program is simply defined as a list of methods.

```
Record method : Set := {
  body      : (list instr);
  nb_local  : nat;
  nb_params : nat;
}.

Definition program := (list method).
```

The notion of state is modified to handle method invocations and returns; the idea is that states now consist of stacks of frames, each frame corresponding to a method invocation, and defined as a record of an operand stack, a local variable map, an index to the method being executed, and a program counter:

```
Record d_frame : Set := {
  d_opstack : (list d_val);
  d_locvars : (list d_val);
  d_method  : method;
  d_pc      : pcs;
}.
```

```
Definition d_state : Set := (list d_frame).
```

The defensive semantics of method invocation, which adds a new frame at the top of the stack of frame for the invoked method, is:

```
Definition d_INVOKE (nargs : nat) (mn : method) (s : d_state) :=
match s with
Nil ⇒ d_Abnormal stack_error |
Cons f lf ⇒
  match (l_take nargs (d_opstack f)) with
  None ⇒ d_Abnormal opstack_error |
  (Some l) ⇒
    match (l_drop nargs (d_opstack f)) with
    None ⇒ d_Abnormal opstack_error |
    (Some l') ⇒
      match (args_tInt l (nb_params mn)) with
      false ⇒ d_Abnormal type_error |
      true ⇒ (d_Normal (Cons (Build_d_frame (Nil d_val)
                              (make_locvars l (nb_local mn))
                              mn
                              (0))
                        (Cons (Build_d_frame l'
                               (d_locvars f)
                               (d_method f)
                               (d_pc f))
                         lf)))
end end end end.
```

where `l_take` and `l_drop` respectively return and remove the given number of arguments in a list, `args_tInt` verifies that the given values are all of type `tInt` and `make_locvars` initializes the set of local variables with the arguments of the method.

The defensive semantics of method return, which pops the current frame and pushes the return type of the invoked method in the operand stack of the invoker method, is:

```
Definition d_RETURN (s : d_state) :=
match s with
Nil ⇒ d_Abnormal stack_error |
Cons f Nil ⇒ d_Abnormal stack_error |
Cons f (Cons f' lf) ⇒
  match (Nth_elt (methods cap) (method_loc f)) with
  None ⇒ d_Abnormal program_error |
  (Some mt) ⇒
    match (d_opstack f) with
    Nil ⇒ d_Abnormal opstack_error
    Cons x lv ⇒
      match x with
      (d_Int _) ⇒ (d_Normal (Cons (Build_d_frame
                                    (Cons x (d_opstack f')
                                           (d_locvars f')
                                           (d_method f')
                                           ((d_pc f') + 1)))
                                      lf)) |
      _ ⇒ d_Abnormal type_error
end end end end.
```

In contrast, the typed virtual machine proceeds on a method-per-method basis, as done by bytecode verification (the definition `a_state` remains unchanged). In this setting, the operation of adding or removing a frame is not relevant for the typed virtual machine. Instead, the typed version of the semantics of method invocation is given by a function `a_INVOKE` that simulates the result on the current frame of the method invocation and the return from that method. More precisely, it removes the argument on the operand stack, pushes the return type of the invoked method on the operand stack and increments the program counter. Thus, `a_INVOKE` does not create a new frame but reflects the result of a complete method invocation on the current frame and the execution can then continue to the next instruction.

```
Definition a_INVOKE (nargs : nat) (mn : method) (s : a_state) :=
match (l_take nargs (a_opstack s)) with
None ⇒ a_Abnormal opstack_error |
(Some l) ⇒
  match (l_drop nargs (a_opstack s)) with
  None ⇒ a_Abnormal opstack_error |
  (Some l') ⇒
    match (args_tInt l) with
    false ⇒ a_Abnormal type_error |
    true ⇒ (a_Normal (Build_a_frame (Cons a_Int l')
                                      (a_locvars s)
                                      ((a_pc s) + 1)))
end end end.
```

Let us now turn to cross-validation. In order to state cross-validation between the defensive and typed virtual machine, we define an abstraction function `alpha` from defensive frame `d_frame` to a typed state `a_state`, using the function `alpha_da_val` in the obvious way. We will also suppose given a function `init` that takes a method and returns the initial typed state corresponding to that method. This initial typed state will have an empty operand stack, the types of its arguments in the local variables and the program counter pointing to the first instruction of the method.

In order to prove the correctness of the bytecode verifier, we introduce the notion of *safe* states. A defensive state is safe if its abstraction is equal to the state computed at the same location by the bytecode verifier on the type virtual machine. As a main result of the bytecode verifier framework, we prove that this notion of safety is an invariant of the defensive virtual machine execution. Thus, at any time of the execution, the abstraction of the considered defensive state is equal to the corresponding typed state computed by the bytecode verifier on the type virtual machine. With the assumption that bytecode verification has been successful (no typed state is an error state), it ensures that the defensive execution never reaches an error state.

The proof of the invariance of the notion of safety distinguishes the scope of each instruction, as commented in Section 2.4. We assume a state `s` to be safe. In order to prove that `d_exec s` is safe we need to prove the following statements.

– If the instruction pointed by `s` is an intra-procedural instruction (i.e. remains in the same frame) then `s` and `d_exec s` must have the form `(Cons f lf)` and `(Cons f' lf)` respectively, and we have to prove that:

  ```
  a_exec (alpha f) = (alpha f')
  ```

  These results are provided by the cross-validation (see Section 2.2.5).

– To guarantee the soundness of typed method invocation or return instructions w.r.t corresponding defensive virtual machine instructions, we have to introduce two functions `a_INVOKE_add` and `a_INVOKE_ret` that correspond to the behavior of `a_INVOKE`, which simulates the modifications made by the defensive virtual machine on a frame when the control flow is given to the invoked method and when it returns to the invoker method. These functions must ensure `a_INVOKE f = a_INVOKE_ret (a_INVOKE_add f)`.

  Then, for the `d_INVOKE` instruction, `s` must have the form `(Cons f lf)`, `d_INVOKE (s)` the form `(Cons f' (Cons f'' lf))` with `alpha f'' = (a_INVOKE_add (alpha f))`, and we have to prove that:

  ```
  (init (d_method f'))) = (alpha f')
  ```

– For the `d_RETURN` instruction, then `s` must have the form `Cons f' (Cons f lf))`, `d_RETURN (s)` the form `(Cons f'' lf)` and we must prove that:

  ```
  (a_INVOKE_ret (alpha f)) = (alpha f'')
  ```

At this point, the architecture of the FW phase can now be summarized as in Figure 4. Further complications arise in an object-oriented setting where the latter statement must be relaxed to account for subtyping, and where an explicit treatment of exceptions must be provided, see [5].

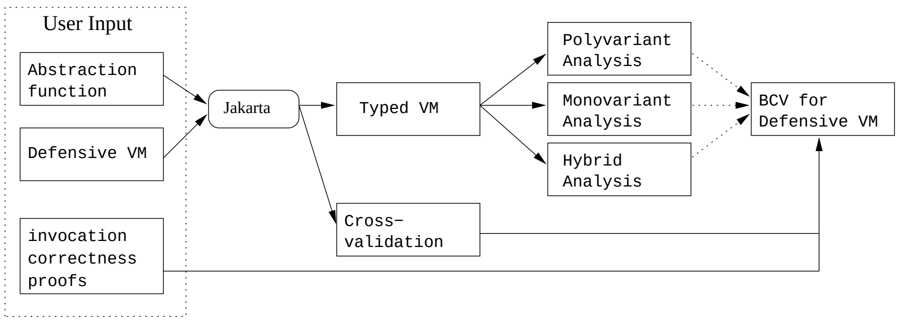

Figure 4. ARCHITECTURE OF THE FW PHASE

3. The Jakarta Specification Language JSL

We now start our presentation of the tool Jakarta, that provides tools support for the VM phase, with the description of its specification language.

JSL is a small language for describing virtual machines. In a nutshell, a JSL theory is given by a set of declarations that introduce first-order polymorphic datatypes and their constructors and by a set of function definitions introducing function symbols and their computational meaning via conditional rewrite rules of the form [3, 11]:

$$\mathtt{l}_1\to\mathtt{r}_1,\ldots,\mathtt{l}_n\to\mathtt{r}_n\Rightarrow\ \mathtt{g}\ \to\ \mathtt{d}$$

A specificity of JSL is to require the right-hand sides of the conditions to be patterns with fresh variables (a pattern is a linear term built from variables and constructors); left-hand sides of the conditions are arbitrary JSL expressions. Such a format of rewriting seems unusual but is closely related to pattern-matching in functional programming and proof assistants and is also well adapted for automating abstractions, as shall be discussed in Section 3.3.

Figure 5 gives an example of a JSL specification with the IIF bytecode, which corresponds to the function described on page 9 with pattern-matching.

3.1. JSL SYNTAX AND EXECUTION MODEL

JSL terms are first-order terms built from variables (from a set $\mathcal{V}$ of variables) and constant symbols. The latter are either constructor symbols (from a set $\mathcal{C}$ of function symbols) introduced by datatype declarations, or defined symbols (from a set $\mathcal{D}$ of function symbols) introduced by function definitions. Finally, every constructor/function symbol comes equipped with its arity, i.e. we assume an arity function $\mathsf{ar} : (\mathcal{C}\cup\mathcal{D}) \to \mathbb{N}$.

```
function d_IIF : pcs → d_state → d_rstate :=

<d_IIF_rule_1>
head (d_opstack s) → Some (d_Int z0),
zeq_bool z0 ZERO → True
  ⇒ d_IIF p s → d_Normal (Mk_d_state (tail (d_opstack s))
                                     (d_locvars s) p);

<d_IIF_rule_2>
head (d_opstack s) → Some (d_Int z0),
zeq_bool z0 ZERO → False
   ⇒ d_IIF p s → d_Normal (Mk_d_state (tail (d_opstack s))
                                      (d_locvars s)
                                      (succ (d_pc s)));

<d_IIF_rule_3>
head (d_opstack s) → Some (d_RA n)
  ⇒ d_IIF p s → d_Abnormal type_error;

<d_IIF_rule_4>
head (d_opstack s) → None
  ⇒ d_IIF p s → d_Abnormal opstack_error.
```

Figure 5. EXAMPLE OF JSL SPECIFICATION: D_IIF

We assume the reader to be familiar with standard notions such as subterms, occurrences, substitutions, and the associated notations; in particular $\mathsf{var}(e)$ denotes the set of variables of an expression.

DEFINITION 3.1.1.

- **Expression.** *The set $\mathcal{E}$ of* expressions *is defined by:*

$$\mathcal{E} := \mathcal{V} \mid \mathcal{C}\ \mathcal{E}^* \mid \mathcal{D}\ \mathcal{E}^*$$

where for $h\ \vec{e} \in \mathcal{E}$ with $h \in \mathcal{C} \cup \mathcal{D}$, it is assumed that $\vec{e}$ is of length $\mathsf{ar}(h)$.

- **Pattern.** *The set $\mathcal{P}$ of patterns is the subset of $\mathcal{E}$ defined by:*

$$\mathcal{P} := \mathcal{V} \mid \mathcal{C}\ \mathcal{P}^*$$

where, in $c\ \vec{p} \in \mathcal{P}$ with $c \in \mathcal{C}$, variables are pairwise distinct (linearity condition) and such that $\vec{p}$ is of length $\mathsf{ar}(c)$.

- **Rewrite rule.** *Given $f \in \mathcal{D}$, an f-*rule *is a rule of the form:*

$$l_1 \rightarrow r_1,\ \ldots\ ,\ l_n \rightarrow r_n \Rightarrow g \rightarrow d$$

where:

- $\vec{r_i} \in \mathcal{P}$, $\vec{l_i}, g, d \in \mathcal{E}$, *and* $g = f\ \vec{x}$ *where* $\vec{x} \in \mathcal{V}$ *are pairwise distinct;*
- $\forall 1 \leq k \leq n\ \mathsf{var}(l_k) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup\ \ldots\ \cup \mathsf{var}(r_{k-1})$ *and* $\mathsf{var}(d) \subseteq \mathsf{var}(g) \cup \mathsf{var}(\vec{r_i})$*;*
- $\mathsf{var}(r_k) \cap \mathsf{var}(g) = \emptyset$ *and* $\mathsf{var}(r_j) \cap \mathsf{var}(r_k) = \emptyset$ *if* $j \neq k$*;*

Conditional rewriting is defined in the following way.

DEFINITION 3.1.2 (JSL Execution model). *Let R be a set of rewrite rules. An expression s R-rewrites to t, written $s \rightarrow_R t$, if there exists a rule $r \in R$ of the form: $l_1 \rightarrow r_1,\ \ldots\ , l_n \rightarrow r_n \Rightarrow g \rightarrow d$, a position p in s and a substitution θ such that:*

$$s_{|_p} = \theta g \text{ and } t = s[p \leftarrow \theta d] \text{ and } \theta l_i \rightarrow^*_R \theta r_i \text{ for } 1 \leq i \leq n$$

where $\rightarrow^*_R$ is the reflexive-transitive closure of $\rightarrow_R$, $s_{|_p}$ denotes the subterm of s at position p, and $s[p \leftarrow x]$ denotes s where $s_{|_p}$ has been replaced by x.

When the conditions $\theta l_i \rightarrow^*_R \theta r_i$ for $1 \leq i \leq n$ are satisfied, we say that substitution θ matches *the conditions of r.*

In the following, we define particular pairs of rules that may appear during abstractions (see Section 4).

DEFINITION 3.1.3. *Given two rules r_1 and r_2 of a function f:*

– *r_1 and r_2 are* overlapping *if there exists a substitution of the parameters of f that matches the conditions of both r_1 and r_2 (critical pairs);*

– *r_1 is* redundant *with r_2 if r_1 and r_2 are overlapping and return the same result (trivial critical pairs), this relation is symmetric;*

– *r_1* subsumes *r_2 if r_1 is redundant with r_2 and any substitution matching the conditions of r_1 also matches the conditions of r_2, this relation is not symmetric.*

3.2. JSL types

JSL types are built from type variables ($\mathcal{V_T}$), datatype symbols ($\mathcal{T}_d$), abstract type symbols ($\mathcal{T}_a$) and synonym type symbols ($\mathcal{T}_s$). Type symbols come equipped with an arity, i.e. we assume a function $ar_{\mathcal{T}}$: $(\mathcal{T}_d \cup \mathcal{T}_a \cup \mathcal{T}_s) \rightarrow \mathbb{N}$.

DEFINITION 3.2.1. *The set $\mathcal{T}$ of types is defined as follows:*

$$\mathcal{T} ::= \mathcal{V}_{\mathcal{T}} \mid \ \mathcal{T}_d \ \mathcal{T}^* \mid \ \mathcal{T}_a \ \mathcal{T}^* \mid \mathcal{T}_s \ \mathcal{T}^*$$

where in $d\ \vec{t}$, with $d \in \mathcal{T}_d \cup \mathcal{T}_a \cup \mathcal{T}_s$, it is assumed that $\vec{t}$ is of length $\mathsf{ar}_{\mathcal{T}}(d)$.

To constructor and function symbols are assigned (first-order) type schemes, i.e. closed expressions of the form $\forall \alpha_1 \ldots \alpha_m.\ \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ where $\alpha_1 \ldots \alpha_m \in \mathcal{V}_{\mathcal{T}}$ and $\sigma_1 \ldots \sigma_n \in \mathcal{T}$. Formally, we assume given for each constructor/function symbol h a declaration of the form $h : \rho$ where ρ is a type scheme.

As usual, expressions are type checked relative to a context in which variables have assigned types. Because of type synonyms, the type checking is done modulo the type synonyms relation, denoted by $=_\delta$.

DEFINITION 3.2.2. *A* context *is a sequence of the form $x_1 : \sigma_1,\ \ldots, x_k : \sigma_k$ where $x_1 \ldots x_k$ are pairwise disjoint variables and $\sigma_1 \ldots \sigma_k$ are types. A* type judgment *is a triple the form $\Gamma \vdash e : t$, meaning that e is of type t in the context Γ. The typing relation is given by the rules of Figure 6.*

Rules are type-checked in the usual way, i.e. a rule $l_1 \rightarrow r_1,\ \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ is well-typed in a context Γ iff there exists a context Δ containing all the free variables occurring in g and the r_is, and such that $\Gamma, \Delta \vdash g, d : \sigma$ and $\Gamma, \Delta \vdash l_i, r_i : \tau_i$ for $1 \leq i \leq n$. Note that the context Δ may be computed from the type declaration of function/constructor symbols. The notation $t\{\vec{\alpha_i} := \vec{\tau_i}\}$ in (Tinst), where t is a type, means t where α_i's have been substituted by τ_i's. The rule (Tinst) allows to instantiate the type of a polymorphic functions with concrete types. Finally a function is well-typed if all its rules are well typed.

3.3. The Jakarta Prover Interface

JSL specifications may be translated almost directly to prototyping environments based on rewriting such as ELAN or automatic theorem provers such as SPIKE.

It is also possible to translate JSL specifications into the format of proof assistants, that represent (recursive) definitions using pattern matching and case expressions instead of rewrite rules. Indeed, the syntactical constraints imposed on conditional rewrite rules yield a direct connection with functional languages and proof assistants, whose specification languages feature a specific construct for case analysis. Intuitively, a nested case analysis in a functional programming language may be viewed as a tree where each non-leaf node is labelled by a

$$(\text{Tbase})\frac{(x:\sigma)\in\Gamma}{\Gamma\vdash x:\sigma}$$

$$(\text{Tinst})\frac{f:\forall\alpha_1\ldots\alpha_m.\sigma_1\to\cdots\to\sigma_n\to\sigma\quad\vec{\tau_i}\in\mathcal{T}}{\Gamma\vdash f:(\sigma_1\to\cdots\to\sigma_n\to\sigma)\{\vec{\alpha_i}:=\vec{\tau_i}\}}$$

$$(\text{Tconv})\frac{\Gamma\vdash t:\sigma'\quad\sigma=_\delta\sigma'}{\Gamma\vdash t:\sigma}$$

$$(\text{Texpr})\frac{\Gamma\vdash f:\sigma_1\to\cdots\to\sigma_n\to\sigma\quad\Gamma\vdash t_1:\sigma_1\ \ldots\ \Gamma\vdash t_n:\sigma_n}{\Gamma\vdash f\ t_1\ldots t_n:\sigma}$$

Figure 6. Typing rules

condition of the form $e \to p$ where p is a pattern, and where leaf nodes are labelled by rewrite rules. Jakarta associates a conditional rewrite rule to each path (from root to leaf) in the tree.

On the other hand, no constraint enforces exhaustiveness or confluence of the rewrite rules defining a function, which means that partial and non deterministic functions can be defined in JSL. In particular, not all JSL specifications can be mapped back into case-expressions. For some JSL specifications, we may apply standard program transformations to obtain another specification which may be translated into a case expression. In particular, we apply the lift monad to make partial functions from A to B into total functions from A to $B_\perp$, or the list monad to make non-deterministic functions from A to B into deterministic functions from A to $B^\star$. For some specifications however, the translation to case expressions is just impossible; such a limitation is inherent to the translation of term-rewriting systems into languages where recursive definitions are given through the combination of fixpoint definitions and case analysis. Further, case expressions that are produced from well-typed JSL specifications may still be considered ill-formed by the target system, in particular because proof assistants usually reject non-terminating functions. However, such limitations have not turned out to be a problem in practice.

On a practical level, the translation from JSL to programming languages and proof assistants is performed via the intermediate language JIR described in Section 2.1 from which we target several proof assistants. Currently, Jakarta provides a means to translate JSL specifications to the proof assistants Coq, Isabelle, and PVS. Furthermore, we provide a translation from Coq to JSL so as to reuse existing Coq developments and a translation from JSL to OCaml [34] for an efficient execution of JSL specifications.

3.4. Running example in JSL

In this section, we provide excerpts from the JSL specification of the running example of the introduction. The JSL specification is generated automatically from the JIR specification using Jakarta built-in tools (see Section 3.3). Notice that, as exlained in Section 2, the definition of states as a record has been translated into an inductive type with one constructor whose arguments correspond to the fields of the record. Accessor functions (`d_opstack`, `d_locvars` and `d_pc`) are defined as follows:

```
function d_opstack : d_state→(list d_val) :=
  ⇒(d_opstack (Mk_d_state opstack0 locvars0 pc0))→opstack0.

function d_locvars : d_state→(list d_val) :=
  ⇒(d_locvars (Mk_d_state opstack0 locvars0 pc0))→locvars0.

function d_pc : d_state→pcs :=
  ⇒(pc (Mk_d_state opstack0 locvars0 pc0))→pc0.

function d_update_ops : (list d_val)→d_state→d_state :=
  ⇒(d_update_ops ops s) →
      (Mk_d_state ops (locvars s) (succ (d_pc s))).
```

To complement the JSL function `d_IIF` of Section 3 and illustrate the abstractions of the following sections, we also give as an example the definition of the bytecode `tLOAD` that pushes in the operand stack the content of a local variable.

```
function d_tLOAD : vm_type →locvars_idx →d_state →d_rstate :=

<d_tLOAD_rule_1>
nth_elt (d_locvars s) l →(Some x) , x →(d_Int z0)
  ⇒d_tLOAD tInt l s →
      d_Normal (d_update_ops (Cons x (d_opstack s)) s);

<d_tLOAD_rule_2>
nth_elt (d_locvars s) l →(Some x) , x →(d_RA p)
  ⇒d_tLOAD tInt l s →d_Abnormal type_error);

<d_tLOAD_rule_3>
nth_elt (d_locvars s) l →(Some x) , x →(d_Int z0)
  ⇒d_tLOAD tRA l s →d_Abnormal type_error);

<d_tLOAD_rule_4>
nth_elt (d_locvars s) l →(Some x) , x →(d_RA n)
  ⇒d_tLOAD tRA l s →
```

```
     d_Normal (d_update_ops (Cons x (d_opstack s)) s));
```

<d_tLOAD_rule_5>

```
nth_elt (d_locvars s) l →None
  ⇒d_tLOAD t l s →d_Abnormal locvar_error).
```

The full JSL specification for this example (including representation and operations on primitive datatypes such as `z`) is around 800 lines long.

3.5. Discussion

The main justification for the format of JSL specifications is that the automatic transformation of specifications is greatly facilitated if the specification language is simple and so designed that specifications lend themselves well to the transformations. The latter consideration is the main justification for the format of JSL specifications. Concretely, pattern matching is less adapted than rewrite rules for abstraction because: (1) an abstraction needs to treat differently each *path* in a pattern-matching, which make the rule presentation easier to manipulate since each rule corresponds to one path; (2) an abstraction may introduce non-deterministic and non-total functions, which are again easier to express with rewrite rules rather than with pattern matching.

Readability is another important consideration when designing a specification language. Of course, readability is not directly influenced by the simplicity of the specification language in which the specifications are written—quite on the contrary, richer specification languages are often more likely to provide enough features for giving specifications in a concise and intuitive form. One particular drawback of JSL in comparison with specifications based on pattern-matching is its verbosity, which is apparent when comparing the JSL and JIR semantics of the IIF bytecode. Nevertheless, there are some potential benefits in providing JSL specifications instead than specifications based on pattern-matching: firstly, the presentation of a language semantics with rules is closer to informal practice. Secondly, the presentation may omit error cases, which is not possible if exhaustive pattern-matching is required. However, Jakarta is a proof-of-concept software whose aim is to establish the possibility of automating the synthesis of the offensive and typed machine and the proofs of cross-machine validation. Therefore, we have not put the necessary effort to turn JSL into a more readable specification language.

Instead, we exploit translation mechanisms between JIR/(fragment of) ML/(fragment of) Coq and JSL, and proceed as follows: first, we write the specification in JIR, ML or Coq, and then translate it to JSL for abstraction, and finally translate the generated specifications back, using information collected during the abstraction to generate Coq scripts that establish cross-machine validation.

4. The Jakarta Transformation Kit JTK

The Jakarta Transformation Kit is a tool to transform JSL specifications. Currently the JTK only supports transformations by *abstraction*, which are needed in the construction and certification of the BCV. The input of the JTK is the specification to transform and a description of this transformation written in an *abstraction script*.

4.1. Principles

The objective of the JTK is to compute, with minimal user interaction, the abstraction $\hat{\texttt{S}}$ of a JSL specification `S`. The abstracted specification $\hat{\texttt{S}}$ is constructed in three successive steps:

1. the user provides for each datatype `T` needed in the abstraction of `S` a corresponding abstract datatype $\hat{\texttt{T}}$ together with an abstraction function $\alpha_{\texttt{T}}$ from `T` to $\hat{\texttt{T}}$; abstraction functions are written in (or translated to) JSL, but must only use unconditional rewriting;

2. the user builds an abstraction script that guides the abstraction process in places where automatic procedures are inappropriate. The script consists of a set of commands, whose purpose and effect is detailed below;

3. based on the inputs of the previous steps, the JTK constructs automatically for each function $\texttt{f}\colon \texttt{T}_1 \rightarrow \ldots \rightarrow \texttt{T}_n \rightarrow \texttt{U}$ of `S` its abstract counterpart $\hat{\texttt{f}}\colon\hat{\texttt{T}}_1 \rightarrow \ldots \rightarrow \hat{\texttt{T}}_n \rightarrow \hat{\texttt{U}}$. Unless indicated otherwise by the abstraction script, the function $\hat{\texttt{f}}$ is constructed by a syntactic translation on the rewrite rules defining `f`, followed by a cleaning phase where vacuous conditions and rules (e.g. tests over expressions deleted by the abstraction) are removed.

We illustrate in Section 4.5 the abstraction process with the synthesis of offensive and typed virtual machines of the running example.

4.2. The default abstraction mechanism

The default abstraction mechanism takes as input a JSL specification S, which consists of datatypes, type definitions, and function definitions, and for each datatype and type definition T of S:

- a corresponding datatype and type definition $\hat{T}$ for the output specification;
- an abstraction function $\alpha : T \rightarrow \hat{T}$.

Note that we require the user to provide an abstract counterpart for type definitions in those cases where the type definition needs to be abstracted differently from its *definiens*.

Abstracting types We require that abstraction functions of parameterized datatypes, such as lists, are also parameterized. Hence we can define inductively a corresponding abstract type $\hat{T}$ and an abstraction function $\alpha : T \rightarrow \hat{T}$ for every type of the input specification S.

Abstracting expressions In order to abstract expressions, we use the abstraction functions $\alpha : T \rightarrow \hat{T}$ between datatypes. In order to guarantee that closed patterns are abstracted into closed patterns, we require that abstraction functions α are defined in JSL by rules of the form

$$\Rightarrow \alpha(\texttt{c x\_1 ... x\_n}) \rightarrow \text{t}$$

where c is a constructor and t a JSL term such that for every substitution σ mapping $x_1 \dots x_n$ to closed patterns, σt reduces to a pattern. The simplest case is when t itself is a pattern. In other cases, such as the definition of the offensive abstraction, we use auxiliary functions and these functions must be evaluated for returning a pattern. Therefore, abstraction of expressions is performed inductively, with two main cases:

- if c is a constructor whose abstraction rule is

  $$\Rightarrow \alpha(\texttt{c x\_1 ... x\_n}) \rightarrow \texttt{t}$$

  and `t` is a pattern, then the abstraction of `c u_1 ... u_n` is `t [x_1:= u_1 ... x_n:= u_n]`, where `.[.:=.]` is used to denote substitution;

- if c is a constructor whose abstraction rule is

  $$\Rightarrow \alpha(\texttt{c x\_1 ... x\_n}) \rightarrow \texttt{t}$$

  and `t` is not a pattern, then the abstraction of `c u_1 ... u_n` is the result of partially evaluating `t [x_1:= u_1 ... x_n:= u_n]`. The idea here is to evaluate `t` but not the `u_i`.

In the sequel, we let $\lceil \texttt{t} \rceil$ denote the abstraction of `t`.

Abstracting rules and functions The abstraction operator on expressions is extended to rules and functions in the obvious way:

$$\lceil l_1 \to r_1, \ldots, l_n \to r_n \Rightarrow l \to r \rceil \rightsquigarrow$$
$$\lceil l_1 \rceil \to \lceil r_1 \rceil, \ldots, \lceil l_n \rceil \to \lceil r_n \rceil \Rightarrow \lceil l \rceil \to \lceil r \rceil$$

$$\lceil \texttt{function}\ f \ : \overrightarrow{T} := d. \rceil \rightsquigarrow \texttt{function}\ \hat{f} \ : \hat{\overrightarrow{T}} := \lceil d \rceil.$$

Cleaning phase After this rewriting stage, the specification is entirely checked. If an expression is not well formed then it is removed and can lead to rule removal or even function removal. If an expression is not well typed then coercions are used if possible, otherwise an error is raised. Coercions are used for example in the offensive abstraction to map return addresses to integers.

Note that the need for coercions arises from the possibility of abstracting types that are equal w.r.t. δ-conversion into completely different types. One possibility would be to proceed in two passes. The first pass would make explicit in expressions the use of δ-conversion in typing derivations, in the same way that order-sorted algebra can be reduced to many-sorted algebra. The second pass would then be the application of the abstraction mechanism without coercions.

4.3. Commands of the abstraction script

The Jakarta Transformation Kit relies on abstraction commands to control the abstraction process. Each command has general form:

<*COMMAND*> ::= <*command*> <*target*>*list* [**by** <*expr*>] [**in** <*scope*>*list*]

The field *target* specifies which syntactic compounds trigger the command. The optional field **by** *expr* is used in substitution commands to specify the syntactic compounds by which targets must be replaced. The optional field **in** *scope* determines in which part of the JSL specification the command <*command*> must be performed; by default the scope of a command is the whole specification.

The form of the scope and target fields uses a notion of *position* inside a specification. Positions will be denoted by expressions of the form $f.i.j.k$, where f is the name of a function, i a rule number, j a condition number ($l_j \to r_j$ or $g \to d$ if $j = 0$), $k \in \{1, 2\}$ designate left or right member of a condition. Thus the i^{th} rule of function f is given by the scheme:

$$f.i.1.1 \to f.i.1.2, \ \ldots \ , \ f.i.n.1 \to f.i.n.2 \Rightarrow f.i.0.1 \to f.i.0.2$$

The forms of target and scope are given by the following:

$$
\begin{array}{lll}
<\!\mathit{target}\!> & ::= & <\!f\!> \mid <\!f\!>.<\!\mathit{pos}\!> \mid <\!f\!>\texttt{@}<\!\mathit{nat}\!> \mid <\!\mathit{expr}\!> \\
<\!\mathit{scope}\!> & ::= & <\!f\!> \mid <\!f\!>.<\!\mathit{pos}\!> \\
<\!\mathit{pos}\!> & ::= & <\!\mathit{nat}\!> \mid <\!\mathit{nat}\!>.<\!\mathit{nat}\!> \\
 & & \mid <\!\mathit{nat}\!>.<\!\mathit{nat}\!>.1 \mid <\!\mathit{nat}\!>.<\!\mathit{nat}\!>.2
\end{array}
$$

where $<\!f\!>.<\!\mathit{pos}\!>$ denotes a position in the function f as described above, and `f@i` denotes the $\mathtt{i}^{th}$ argument of `f` (see commands `drop` and `select` in Figures 7 and 8).

Commands fall into three categories: discarding commands, substitution commands and post-processing commands. Figure 7, 8 and 9 describe informally the effect of each command. A more precise description of main commands is given at section 4.4.

Discarding commands (Figure 7) specify which patterns in expressions are unwanted and must be deleted. For example, to generate an offensive VM, discarding commands are used to remove all type verifications. Another use of these commands is for removing overlapping rules in the abstracted specification.

Abstraction command	**Effect**
`drop` largloc	Removes the arguments listed in largloc. **`drop`** `f@i` replaces $(\mathtt{f}\ \mathtt{x}_1\ldots\mathtt{x}_i\ldots\mathtt{x}_n)$ by $(\mathtt{f}\ \mathtt{x}_1\ldots\mathtt{x}_{i-1}\ \mathtt{x}_{i+1}\ldots\mathtt{x}_n)$
`delete` ltarget	Removes all (occurrences of) elements described by ltarget.
`reject` lexp	If rules with same preconditions and different conclusion are encountered, remove rules containing exp $\in$ lexp.

Figure 7. Discarding commands

Substitution commands (Figure 8) specify coercions and substitutions that should be used during the translation of rewrite rules. Substitution commands are used for example to replace dynamic method lookup by static lookup in the typed semantics of Java Card method invocation.

Post-processing commands (Figure 9) deal with functions which may have become non-deterministic or non-total during abstraction. Post-

Abstraction command	**Effect**
`coercion` lfname	Functions of lfname are declared as coercions in case of typing errors during by the translation phase.
`select` largloc	**`select`** `f@i` replaces `(f` $\mathtt{x}_1 \ldots \mathtt{x}_i \ldots \mathtt{x}_n$`)` by $\mathtt{x}_i$
`replace` ltarget `by` target	Replaces elements of ltarget by target.

Figure 8. Substitution commands

processing commands are used for example to collect all possible return states into lists during the construction of the typed VM.

Abstraction command	**Effect**
`determine` lfname	Transforms the functions in lfname in order to make them deterministic (by collecting results in a list). The transformation is propagated consequently.
`totalize` lfname	Transforms the functions in lfname in order to make them total (lifting types). The transformation is propagated consequently.

Figure 9. Post-processing commands

4.4. Towards a rewriting semantics of abstraction commands

A function definition can be seen as a set of rewrite rules, and abstraction commands, which specify how to transform function definitions, are actions that operate upon sets of rewrite rules. With the exclusion of **`coercion`**, **`determine`** and **`totalize`**, these actions can be given a precise semantics in terms of rewriting rules. These rules are applied to the specification (or to a part of the specification if the scope field is specified) during the abstraction.

The command **`replace`** `t` **`by`** `u` corresponds to the following rewriting rule added to the default mechanism described above. Such added rules are applied instead of the default mechanism when possible.

$$\lceil t \rceil \rightsquigarrow u$$

`delete` `t` is expressed by:

$$\lceil t \rceil \rightsquigarrow \varepsilon$$

where ε is the empty expression. Expressions containing ε are ill-formed and will be removed by the cleaning phase explained above. Removing an expression can lead to remove the whole rule if it becomes itself ill-formed (for instance if ε appears in the conclusion).
`select` `f@i` corresponds to:

$$\lceil f\ \overrightarrow{t_{1,n}} \rceil \rightsquigarrow \lceil t_i \rceil$$

We see that an effect of this rule is that f will not be called in the abstracted specification, therefore it will not be abstracted at all.
`drop` `f@i` is expressed by:

$$\begin{array}{rcl} \lceil f\ \overrightarrow{t_{1,n}} \rceil & \rightsquigarrow & \hat{f}\ \overrightarrow{\lceil t_{1,i-1} \rceil} \overrightarrow{\lceil t_{i+1,n} \rceil} \\ \lceil \texttt{function}\ f\ : \overrightarrow{T_{1,n}} := d. \rceil & \rightsquigarrow & \texttt{function}\ \hat{f}\ : \overrightarrow{\lceil T_{1,i-1} \rceil} \to \overrightarrow{\lceil T_{i+1,n} \rceil} := \lceil d \rceil. \end{array}$$

where the second rule rewrites the type of function f, and $\overrightarrow{T_{n,m}}$ stands for the arrow type $T_n \to ... \to T_m$. **`reject`** `e` is expressed by a rewrite rule applying to *sets of rules*:

$$\lceil\ X \Rightarrow t \to C[e]\ \ ;\ \ X \Rightarrow t \to u\ \ \rceil \rightsquigarrow \lceil X \Rightarrow t \to u \rceil$$

where $C[e]$ is a *context*, i.e. a term containing an occurrence of e, and X is a list of conditions. In this rule the symbol ";" is associative-commutative and therefore, the place and order in which the two rules appear in the function is not important.

4.5. Running example in JTK

In this section, we will continue to illustrate the action of Jakarta on our running example, describing abstraction functions and scripts used to generate the offensive and typed virtual machines.

4.5.1. *Offensive abstraction*

In order to obtain the offensive virtual machine from the defensive virtual machine, we first need to describe datatypes of the target virtual machine as well as abstraction and coercion functions. For this virtual machine, the notion of value is restricted to a numerical value. The corresponding notion of states and abstractions functions are easily defined. Notice that return addresses values are now assigned the same type `z` as values, since it is not possible any more to distinguish these values from their virtual machine type.

```
function alpha_do_val : d_val →o_val :=
⇒alpha_do_val (d_Int v) →v ;
⇒alpha_do_val (d_RA n) →(n2z n).

function alpha_do_lval : (list d_val) →(list o_val) :=
⇒alpha_do_lval Nil →Nil;
⇒alpha_do_lval (Cons x y) →
    Cons (alpha_do_val x) (alpha_do_lval y).

function alpha_do : d_state →o_state :=
⇒alpha_do (Mk_d_state ops loc p)
  → (Mk_o_state (alpha_do_lval ops) (alpha_do_lval loc) p).

function alpha_da_rstate : d_rstate →o_rstate :=
⇒alpha_do_rstate (d_Normal js) →(o_Normal (alpha_do js));
⇒alpha_do_rstate (d_Abnormal x) →(o_Abnormal x).
```

As discussed in Section 4.6, datatypes and functions following the definition `alpha_do_val` could be automatically generated since they are just iterators.

The script for the abstraction, given below, introduces with the keyword **abstract** the function to abstract, here `d_exec` (all depending sub-functions will also be abstracted), the functions to use for the abstraction, the prefix to add to the generated functions, some commands for the abstraction, some coercions to apply in case of type mismatch detected by the type checker and names of output files. In our case, these commands will be limited to remove rules leading to the error case `type_error`, since it becomes irrelevant for the offensive virtual machine. Without the command **coercion**, functions manipulating `RA` values would produce typing errors during the abstraction process.

```
abstract d_exec

with alpha_do_val alpha_do_lval alpha_do alpha_do_rstate
prefix o_

delete type_error

coercion n2z z2n

into jcvm_off_functions log jcvm_log
```

We now turn to the generated functions for `IIF` and `tLOAD`. We notice that they indeed use the new datatypes and that irrelevant rules have been deleted.

```
function o_IIF : pcs→o_state→o_rstate :=
<o_IIF_rule_1>
o_head (o_opstack s) →(Some z0),
zeq_bool z0 ZERO →True
  ⇒o_IIF p s →(o_Normal (Mk_o_state (o_tail (o_opstack s))
                         (o_locvars s) (z2n p)));

<o_IIF_rule_2>
o_head (o_opstack s) →(Some z0),
zeq_bool z0 ZERO →False
  ⇒o_IIF p s →(o_Normal (Mk_o_state (o_tail (o_opstack s))
                         (o_locvars s) (succ (z2n (o_pc s)))));

<o_IIF_rule_4>
o_head (o_opstack s) →None
  ⇒o_IIF p s →(o_Abnormal opstack_error).

function o_tLOAD : vm_type→locvars_idx→o_state→o_rstate :=
<o_tLOAD_rule_1>
o_nth_elt (o_locvars s) l →(Some x)
  ⇒o_tLOAD t l s →
      (o_Normal (o_update_ops (Cons x (o_opstack s)) s));

<o_tLOAD_rule_5>
o_nth_elt (olocvars s) l →None
  ⇒o_tLOAD t l s →(o_Abnormal locvar_error).
```

4.5.2. *Typed abstraction*

Datatypes and abstraction functions for the typed abstraction, given below, are similar to the ones of the offensive abstraction. The numerical value of integer value is lost, however the value associated to return addresses must be kept since it is needed for subroutines control flow.

```
function alpha_da_val : d_val →a_val :=
⇒alpha_da_val (d_Int v) →a_Int ;
⇒alpha_da_val (d_RA n) →(a_RA n).

function alpha_da_lval : (list d_val) →(list a_val) :=
⇒alpha_da_lval Nil →Nil;
⇒alpha_da_lval (Cons x y) →
    Cons (alpha_da_val x) (alpha_da_lval y).

function alpha_da : d_state →a_state :=
⇒alpha_da (Mk_d_state ops loc p)
  → (Mk_a_state (alpha_da_lval ops) (alpha_da_lval loc) p).

function alpha_do_rstate : d_rstate →a_rstate :=
⇒alpha_da_rstate (d_Normal js) →(a_Normal (alpha_da js));
⇒alpha_da_rstate (d_Abnormal x) →(a_Abnormal x)
```

The script for the type abstraction includes a command **determine** that collects all possible results for `a_IIF` into a list. Note that without this command, the abstraction of `d_IIF` would be non-deterministic, because numerical values on which `d_IIF` operates is lost and Jakarta would raise a warning (observable in the program output and in the log file).

```
abstract d_exec

with alpha_da_val alpha_da_lval alpha_da alpha_da_rstate
prefix a_

determine a_IIF

into jcvm_a_functions log jcvm_log
```

The functions generated by the abstraction follow. Notice in particular the list returned by the function `a_IIF` as an effect of the command **determine**.

```
function a_IIF : pcs→a_state→(list a_rstate) :=

a_head (a_opstack s)→(Some a_Int)
  ⇒(a_IIF p s)→
  (Cons (a_Normal (Mk_a_state (a_tail (a_opstacks))
                        (a_locvars s) p))
    (Cons (a_Normal (Mk_a_state (a_tail (a_ops tack s))
                          (a_locvars s) (succ (a_pc s))))
    Nil));

a_head (a_opstack s)→(Some (a_RA n))
  ⇒a_IIF p s→(Cons (a_Abnormal type_error) Nil).

a_head (a_opstack s)→None
  ⇒a_IIF p s→(Cons (a_Abnormal opstack_error) Nil);

function a_tLOAD : vm_type→locvars_idx→a_state→a_rstate :=

<a_tLOAD_rule_1>
a_nth_elt (a_locvars s) l →(Some x) , x →a_Int
  ⇒a_tLOAD tInt l s →
      (a_Normal (a_update_ops (Cons x (a_opstack s)) s));

<a_tLOAD_rule_2>
a_nth_elt (a_locvars s) l →(Some x), x →(a_RA p)
  ⇒a_tLOAD tInt l s →(a_Abnormal type_error);

<a_tLOAD_rule_3>
a_nth_elt (a_locvars s) l →(Some x) , x →a_Int
  ⇒a_tLOAD tRA l s →(a_Abnormal type_error);
```

<a_tLOAD_rule_4>
```
a_nth_elt (a_locvars s) l →(Some x) , x→(a_RA p)
 ⇒a_tLOAD tRA l s →
     (a_Normal (a_update_ops (Cons x (a_opstack s)) s));
```

<a_tLOAD_rule_5>
```
a_nth_elt (a_locvars s) l →None
 ⇒a_tLOAD t l s →(a_Abnormal locvar_error).
```

4.6. Limitations

The abstraction engine is at an experimental stage of development, and suffers from a number of limitations that affect the size of scripts negatively. Below, we detail some of the limitations, that appear most often in abstractions, and discuss possible simple solutions.

Lack of default abstractions The abstraction functions `alpha_do_lval`, `alpha_do` and `alpha_do_rs` of the offensive abstraction script page 4.5.1 are just iterators of the function `alpha_do_val` for types `list`, `state` and `rstate`. It would be desirable to implement a mechanism that builds abstraction functions automatically for any inductive type containing `val` using standard techniques.

Lack of code optimization Another limitation of Jakarta is the lack of optimizations in the specifications produced by the abstraction. In particular, if all rules of a pattern matching return the same result, the pattern matching could be deleted and the different rules collapsed into a single one.

Detecting redundant rules The abstraction process may generate overlapping rules. In the general case, overlapping rules are a source of non-determinism that should be handled by the user through the abstraction script. However, in practice, it is often the case that overlapping rules are ordered by the subsumption relation between rules (one rule is a special case of the other, see Section 3.1.3), and do not cause any non-determinism. In order to obtain a non-overlapping rewriting system, one should keep the most general rewrite rule, and eliminate the other subsumed rules. Unfortunately, the detection of redundant rules is currently done by pure syntactical rule comparison, and the user is forced to insert a command in the script to delete other redundant rules. It would be preferable to resort to unification tests that would detect more redundant rules automatically—of course it is undecidable in general whether a set of rules is redundant, in particular it is hard to know if two overlapping rules return the same result. This would however lead to a significant simplification of the abstraction scripts. See Section 6.2.3 for an example of redundant rules undetected by Jakarta (function `res_null`).

Type inference and type synonyms In a large specification, it is often convenient to define types, say $T_1, \ldots, T_n$, as synonyms to a type T. Such a situation occurs when elements of different sets have to be manipulated in the same way. For instance, it is very convenient to specify the different types of indexes as synonyms of the type of naturals. The abstraction process may fail if these types have to be abstracted to different types. The main reason lies in the fact that it is not simple, when analyzing an expression to abstract, to infer the right type and decide which abstraction function to use. A work around is to explicitly disambiguate such situation by adding abstraction commands in the script. However, it would be desirable to improve the interaction with the type-checker in order to solve such ambiguities automatically.

5. The Jakarta Automation Kit JAK

In order to cross-validate the virtual machines, we have been developing a repository of tools that generate the statements and proofs of auxiliary lemmas required in order to show that diagrams commute, as shown in Figures 2 and 3. Generated proofs are provided as proof scripts that must be compiled, and use purpose-built tactics to reason about recursive functions and to perform rewriting.

For the moment only Coq output has been completely implemented. The amount of work needed to extend to another prover depends on which tactics are available in the targeted prover. For example the automatic first-order prover SPIKE already has the kind of induction (via so-called *implicit induction*) needed. Isabelle has been equipped with such tactics too (using K. Slind's TFL [40]). However the main part of the mechanism (lemma generation) is prover independent: all information on auxiliary lemmas is stored in a prover independent data structure. Of course, the way the automation kit has to formulate and prove lemmas strongly depends on the target prover and the kind of proofs we want to achieve. We discuss briefly the interest of an independent intermediate language for this purpose, which is ongoing work, in our conclusion.

Currently the Coq specific part is implemented in a quite *ad-hoc* manner (see Section 5.4), i.e. the user has some limited control on the generated lemmas via several commands in the abstraction script but some of these commands are not prover independent. A list of the current features available in the abstraction script for lemmas alteration follows:

- Add preconditions to lemmas in a prover independent syntax using the keyword **`addprecond`**;
- Replace the proof script of a lemma directly by arbitrary text if the generic mechanism is not adapted (this was very rare in our examples);
- Add arbitrary text (for example additional lemmas in Coq syntax) to proof scripts.

We will only mention **`addprecond`** (Section 5.1 and Figure 14) in the following, as others are hardly used in our examples.

5.1. Automatic generation of commutation lemmas

In order to prove the commutation diagrams of Figures 2 and 3, we have developed a procedure, plugged into the abstraction engine, that generates an auxiliary lemma for each abstracted function of the specification. More precisely, let f be a function, $\hat{f}$ its abstraction version and α the global abstraction function, the corresponding lemma has the following generic form:

$$\forall \overrightarrow{x_i}.(Prec_{\hat{f}}\ \overrightarrow{x_i}) \quad \Rightarrow \quad (\mathcal{R}\ \ \alpha(f\ \overrightarrow{x_i})\ \ \hat{f}(\alpha\ \overrightarrow{x_i}))$$

where:

- $\mathcal{R}$ is a binary relation definable in the target system. In the case of the offensive abstraction, $\mathcal{R}$ is instantiated to $=$, whereas it can be instantiated to $\in_{\leq}$ (membership up to subtyping on the types of the virtual machine) in the case of the typed abstraction of the Java Card virtual machine (see Section 6);
- The predicate $Prec_{\hat{f}}$ restricts the commutation of f and $\hat{f}$ to the desired cases. It is crucial that $Prec_{\hat{f}}$ has the right formulation: if $Prec_{\hat{f}}$ is too restrictive, then we may be unable to prove the global correctness of the virtual machine as a consequence of the lemma; if it is not restrictive enough, we may be unable to prove the lemma itself.

  In fact, $Prec_{\hat{f}}$ is a conjunction of *premises* excluding situations where commutation is false, typically cases where a defensive function returns a type error must be excluded from commutation lemmas of offensive abstraction. More generally, the form of the premise $Prec_{\hat{f}}$ is determined by the way the function f is abstracted: each case where the abstraction makes f and $\hat{f}$ different must be added to $Prec_{\hat{f}}$ because f and $\hat{f}$ will not commute in such cases. Connection between predicate $Prec_{\hat{f}}$ and the global cross-validation theorem is established at Section 5.2.

$Prec_{\hat{f}}$ is also used to excluded cases that we will not prove with Jakarta, as explained in Section 2.2.5. For example, we use this feature in the typed VM to exclude cases where an exception is thrown, since the correctness of the BCV in such cases is not expressed in term of commutation.

Notice that we must prove this lemma for each abstracted function in the specification. This eventually include the main (one-step) execution function `d_exec`. See Section 5.2 on how the commutation of `d_exec` is used to prove the main commutation theorem.

Conditionally add a precondition. Abstraction operations should not always generate new preconditions. For example the **`delete`** command is sometimes applied to delete redundant rules (see Sections 3.1.3 and 4.6) in order to clean an abstracted function. In this case, **`delete`** should not add a new premise to $Prec_{\hat{f}}$. On the other hand, **`delete`** is also used to forget rules that are obsolete in the abstracted machine: type-checking rules when doing offensive abstraction or exception-raising rules when doing typed abstraction. These rules should be added to $Prec_{\hat{f}}$ since they correspond to non-commuting states.

In Jakarta, deletion of a rule, implied by a command c (**`delete`** or **`reject`**), generates a new premise by default (see below how this is done) unless the keyword `[redundant]` has been used in c (see Section 6.2.3 for an example).

There are four situations where the abstraction engine adds a premise to a lemma: when deleting a conclusion or a precondition, when propagating preconditions of a function g to a function f calling g, and finally when the user adds a precondition by hand. We describe them more precisely in the following, see Section 6.2 for examples.

Deleted conclusion. If r is a rule of the form: $l_1 \rightarrow r_1, \ \ldots \ , l_n \rightarrow r_n \Rightarrow (f \ \overrightarrow{x_i}) \rightarrow (...t...)$ and a command **`delete`** t is applied (without the `[redundant]` variant) to remove t, then the following premise must be added to $Prec_{\hat{f}}$: $(f \ \overrightarrow{x_i}) \neq (...t...)$.
Some examples illustrating this case are the following:

- In the offensive abstraction, rules of the form: $\cdots \Rightarrow (f\ \overrightarrow{x_i}) \rightarrow$ (`Abnormal type_error`) are deleted by the command **`delete`** `type_error`. So the precondition $(f\ \overrightarrow{x_i}) \neq$ (`Abnormal type_error`) is added to $Prec_{\hat{f}}$. This kind of precondition is also generated when a **`reject`** rule is applied.
- Similarly, in the typed abstraction of the case study from Section 6, preconditions of the form $(f\ \overrightarrow{x_i}) \neq$ (`ThrowException` e) are generated because instructions that do not stay in the same frame do not commute in the typed abstraction. As explained in Section 6.1, Jakarta is not used for proof of correctness of the BCV for these instructions.

Deleted preconditions. If r is a rule of the form: $l_1 \rightarrow r_1,\ \ldots\ , l_n \rightarrow r_n \Rightarrow g \rightarrow d$ and a command **`delete`** is applied (without the `[redundant]` variant) to remove t, then if t appears in one l_i, and the rule r is deleted, then the following premise is added to $Prec_{\hat{f}}$: $\neg(l_1 = r_1 \wedge \cdots \wedge l_n = r_n)$. Indeed, since r is deleted the result of the function will not commute in the corresponding case. This kind of precondition is also generated when a rule is deleted directly by its name without `redundant` (**`delete`** `f.i.j.k`), for the same reason.

Propagation of the premises. Since commutation of f and $\hat{f}$ is a consequence of the commutation of the auxiliary functions of f, $Prec_{\hat{f}}$ may refer to $Prec_{\hat{g}}$ if g is in the dependency graph of f. More precisely, for each function g appearing in a rule r of f of the form: $l_1 \rightarrow r_1,\ \ldots\ , l_i \rightarrow r_i,\ (...(g\ \overrightarrow{x_j})...) \rightarrow r_{i+1},\ \cdots \Rightarrow \ldots$ if g is not removed by the abstraction, the following premise is added to $Prec_{\hat{f}}$: $(l_1 = r_1 \wedge \cdots \wedge l_i = r_i) \rightarrow (Prec_{\hat{g}}\ \overrightarrow{x_j})$.

Preconditions added by hand. We give the user the ability to add preconditions by hand in the abstraction script. It is useful in rare cases where the default preconditions are incomplete (see Section 6.3). These preconditions are propagated as described in the previous paragraph. The command adding a new precondition to the lemma corresponding to the function `f` is the following:

`addprecond` <*logicexpr*> **`to`** `f`

where `logicexpr` is a property expressed in a basic logical language with universal quantification, which is translated to the targeted theorem prover during abstraction.

5.2. Automatic generation of the main theorem

The final lemma generated by the method described above expresses the commutation of the main execution function d_exec with its abstracted version. This gives the following for offensive and typed abstractions:

$$\forall \overrightarrow{x_i}.(Prec_{\texttt{o\_exec}}\ \overrightarrow{x_i}) \Rightarrow \\ \texttt{alpha\_do}\ (\texttt{d\_exec}\ \overrightarrow{x_i}) = (\texttt{o\_exec}\ (\texttt{alpha\_do}\ \overrightarrow{x_i})) \quad (3)$$

$$\forall \overrightarrow{x_i}.(Prec_{\texttt{a\_exec}}\ \overrightarrow{x_i}) \Rightarrow \\ \texttt{alpha\_da}\ (\texttt{d\_exec}\ \overrightarrow{x_i}) = (\texttt{a\_exec}\ (\texttt{alpha\_da}\ \overrightarrow{x_i})) \quad (4)$$

For the proof to be complete, the final lemma must imply the main commutation theorem expressed by the commuting diagrams of Figures 2 and 3. Proving this last property actually reduces to proving that generated preconditions $Prec_{\texttt{o\_exec}}$ and $Prec_{\texttt{a\_exec}}$ are not too strong. We illustrate this for both theorems:

- In the offensive abstraction, the main commutation theorem is the following.

  $$\forall \overrightarrow{x_i}.(\texttt{d\_exec}\ \overrightarrow{x_i}) \neq \texttt{type\_error} \Rightarrow \\ \texttt{alpha\_do}\ (\texttt{d\_exec}\ \overrightarrow{x_i}) = (\texttt{o\_exec}\ (\texttt{alpha\_do}\ \overrightarrow{x_i})) \quad (5)$$

  Therefore, in order to have a complete proof of the commutation of offensive and defensive virtual machines, it is necessary to prove:

  $$\forall \overrightarrow{x_i}.(\texttt{d\_exec}\ \overrightarrow{x_i}) \neq \texttt{type\_error} \Rightarrow (Prec_{\texttt{o\_exec}}\ \overrightarrow{x_i})$$

  This lemma is trivial, since generated preconditions $Prec_{\texttt{o\_exec}}$ exactly express the fact the execution does not lead to type_error.

- In the typed abstraction, no precondition is generated and the main theorem is the following:

  $$\forall \overrightarrow{x_i}.\texttt{alpha\_da}\ (\texttt{d\_exec}\ \overrightarrow{x_i}) \in (\texttt{a\_exec}\ (\texttt{alpha\_da}\ \overrightarrow{x_i}))$$

  However, for other abstractions, such as for CertiCartes (see Section 6), preconditions might express the fact that no exception is raised during the execution or the execution remains in the same method.

5.3. Reasoning tactics

In order to automate the proofs of these lemmas, we have designed and exploited Coq tactics that generate induction principles for recursive functions. While being built for the purpose of cross-validation and for Coq, these tactics are of general interest and their underlying principles are applicable to other proof assistants, see [4] for more details. In order to implement completely our methodology for another prover, it is necessary to have similar tactics/strategies. Like we said above, this is the case in SPIKE and Isabelle. In the following, we explain the purpose of these tactics and how they have been implemented in Coq specifically.

Induction principles are one of the powerful tools provided by proof assistants like Coq. These principles are generated automatically for relational specifications, and allow to reason by cases and by induction. We have developed a similar mechanism for functional specifications. In a nutshell, the induction principle of a function is built following the shape of this particular function, and is particularly useful for reasoning by case/induction on the possible branches of the definition of the function. For example, the induction principle associated to the IIF function has type:

```
∀ (Q : pcs → d_state → Prop),
  (∀ (p : pcs) (s : d_state) (z0 : z),
    (zeq_bool z0 ZERO)=True →
    (head (d_opstack s))=(Some (d_Int z0)) → (Q p s)) →
  (∀ (p : pcs) (s : d_state) (z0 : z),
    (zeq_bool z0 ZERO)=False →
    (head (d_opstack s))=(Some (d_Int z0)) → (Q p s)) →
  (∀ (p : pcs) (s : d_state) (n : pcs),
    (head (d_opstack s))=(Some (d_RA n)) → (Q p s)) →
  (∀ (p : pcs) (s : d_state),
     (head (d_opstack s))=None → (Q p s)) →
  (∀ (p : pcs) (s : d_state), (Q p s))
```

Notice how the equalities capture the *environment* induced by the branches of the corresponding **match** expressions, and can be used in proofs. The IIF function is not recursive but our mechanism applies equally well to recursive functions. Consider for example the function which divides a natural number by 2. The function is given by the following Coq definition (where S and O are the constructors of type nat in Coq):

```
Fixpoint div2 (n : nat) : nat :=
  match n with
  | O ⇒ 0
  | S m ⇒
    match m with
    | O ⇒ 0
    | S n′ ⇒ S (div2 n′)
    end
  end.
```

Its associated induction principle is of type:

```
∀ (Q : nat →  Prop),
  (Q 0) →
  (Q 1) →
  (∀ (n′ : nat), (Q n′) →  (Q (S (S n′)))) →
    ∀ (n :nat), Q n
```

The last branch contains an induction hypothesis `(Q n′)`, which corresponds to the recursive call in the last branch of `div2`.

The main components of the package are:

– A Coq *command* **`Functional Scheme`** which builds a general induction principle from the definition of a function f, i.e. a theorem of the form:

$$(Q : \forall \overrightarrow{x_i} : \overrightarrow{T_i}.Prop)\ \ (H_1 : PO_1)...(H_n : PO_n)\ \rightarrow\ \forall \overrightarrow{x_i} : \overrightarrow{T_i}.(Q\ \overrightarrow{x_i})$$

where the PO_i's are the proof obligations corresponding to each branches of f, corresponding to `(Q 0)`, `(Q 1)` and `(∀ (n′:nat), (Q n′) → (Q(succ(succ n′))))` in the example above, and $\overrightarrow{x_i}$'s correspond to the arguments of f (`n` above). To make an elimination, the user just applies the theorem. The advantage of this method is that the structure of the function (and its type checking) is not duplicated each time we make an elimination;

– A *tactic* **`functional induction`** which applies directly to a particular goal. This tactic allows far more automation because we can replace automatically occurrences of case arguments in subgoals by the pattern corresponding to each branch. We implemented a complementary tactic called **`Rewall`** which performs rewriting using all previous lemmas and hypothesis generated by **`functional induction`**.

5.4. Generation of proof scripts

The tactics described above are combined with standard tactics of Coq in order to build a default proof script for each lemma generated by the method described in Section 5.1. Figure 10 shows the Coq proof script generated for the commutation proof of the function `d_IIF`

(Section 2.2.2) and its offensive version `o_IIF`. The script includes the definition of the predicate containing the premises of the lemma, here the predicate has one premise corresponding to the deleted conclusion due to the command **`delete`** `type_error`. Then comes the lemma itself with the tactics following the scheme described in Section 6.1: first a case analysis is made using the tactic described above, then equational reasoning is used (tactic `Rewall`).

```
(* Predicate *)
Definition IIF_pred (p : pcs) (s : d_state) :=
  (d_IIF p s) ≠ (d_Abnormal type_error).
(* Unfold the predicate in automatic tactics *)
Hints Unfold IIF_pred : jcvmdb.

(* Commutation lemma *)
Lemma IIF_eq : ∀(p : pcs) (s:d_state), (IIF_pred p s) →
  (o_IIF p (alpha_do s)) = (alpha_do_rstate (d_IIF p s)).

intros p s.
 (* Induction/case step *)
  functional induction d_IIF params p s;
  trivial;intros __precond1;
 (* Get rid of contradictory cases *)
  try (elim __precond1;unfold o_IIF d_IIF;Rewall;auto;fail);
 (* Equational reasoning using previous lemmas *)
  repeat progress (Rewall;simpl;trivial);
 (* Finish *)
  auto with jcvmdb;eauto with jcvmdb.
Save.

(* Put this new lemma into databases for further equational reasoning *)
Hint Rewrite [IIF_eq] in jcvmdb
  using solve [auto|simpl;auto|eauto 10].
Hint Resolve [IIF_eq] in jcvmdb.
```

Figure 10. DEFAULT COQ SCRIPT FOR IIF

From the running example from Section 2, this default script is sufficiently powerful to solve all commutation proofs without modification, provided that we prove and register one auxiliary lemma stating that for all `p`, `(z2n (abstract_do_pcs p))` = `p` for `p >= 0`. While it has not been done, we argue that generating a similar default script for provers that have tactics similar to **`functional induction`** is simple: once case analysis and induction have been done by **`functional induction`**, only equational rewriting is needed.

5.5. Running example in JAK

In the remaining of this section, we will present how Jakarta is used to obtain the expected cross-validation results of Section 2 from the JSL definitions of the running example (excerpts given in Sections 3.4 and 4.5 and Figure 5).

We will describe the commands to be added to the scripts given in Section 4.5 to obtain cross-validation results. Scripts for these two examples are very concise, however we will see that the level of automation is really high.

5.5.1. *Offensive abstraction*

During the abstraction for the offensive virtual machine, lemmas stating the correctness of the abstraction (cross-validation) are generated automatically. Preconditions `IIF_pred` and `tLOAD_pred` are added, as described in Section 5.1, in order to exclude cases corresponding to the command `remove`.

```
Definition IIF_pred (p:pcs) (s:d_state) :=
  (d_IIF p s) ≠ (d_Abnormal type_error).

Lemma IIF_eq : ∀ p:pcs, ∀ s:d_state, (IIF_pred p s)
 → (o_IIF (abstract_do_pcs p) (alpha_do s)) =
      (alpha_do_rstate (d_IIF p s)).

Definition tLOAD_pred (t:vm_type) (l:locvars_idx) (s:d_state) :=
  (d_tLOAD t l s) ≠ (d_Abnormal type_error).

Lemma tLOAD_eq : ∀ t:vm_type, ∀ l:locvars_idx, ∀ s:d_state,
(tLOAD_pred t l s) →
  (o_tLOAD t l (alpha_do s)) = (alpha_do_rstate (d_tLOAD t l s)).
```

All the lemmas needed for the commutation of offensive and defensive VMs are generated and proved automatically, except for the `tLOAD` bytecode. The following script (together with the two auxiliary lemmas) is appended to the script given in Section 4.5.1 in order to have all proofs generated and compiled (accepted) by Coq directly:

```
transparent map_list map_option map_prod map_sum alpha_do
 alpha_do_rstate alpha_do_val alpha_do_lval abstract_do_pcs

proof tLOAD
"
 intros t l s.
 unfold o_tLOAD d_tLOAD;
   functional induction d_tLOAD params t l s;
   trivial;intros __precond1;
   Try (elim __precond1;unfold o_tLOAD d_tLOAD;Rewall;auto;fail);
   repeat progress (Rewall;simpl;trivial);
   auto with jcvmdb;eauto with jcvmdb.
  replace z0 with (alpha_do_val (d_Int z0));auto with jcvmdb.
  replace (n2z n) with (alpha_do_val (d_RA n));auto with jcvmdb.
"

into jcvm_off_functions log jcvm_log
```

The command `transparent name1 name2...` indicates to the Coq script generator to make the constants `name1 name2...` transparent (i.e. their definitions can be unfolded during the Coq proof). The command `proof f string` is used to replace the default proof script of the commutation lemma of the function `f` by `string`. Here we see that commutation lemma of function `tLOAD` needs 2 additional lines (the 6 first lines have been generated by JAK as explained in Section 5.4). `tLOAD` is the only function needing such replacement.

5.5.2. *Typed abstraction*

Lemmas needed to establish cross-validation are also generated during the abstraction of the typed virtual machine. They do not include any precondition, and the statement for `IIF` reflects the new return types for this function (`In` corresponds to list membership).

```
Lemma IIF_eq : ∀p:pcs, ∀s:d_state
(In (alpha_da_rstate (d_IIF p s)) (a_IIF p (alpha_da s))).

Lemma tLOAD_eq : ∀t:vm_type, ∀l:locvars_idx, ∀s:d_state,
(a_tLOAD t l (alpha_da s)) = (alpha_da_rstate (d_tLOAD t l s)).
```

In this abstraction again, very little addition to the abstraction script is needed to make all proofs compilable by Coq. Only one auxiliary lemma and a particular proof information for `tLOAD`, as in previous section, are needed:

```
transparent map_list map_option map_prod map_sum alpha_da
  alpha_da_rstate alpha_da_val alpha_da_lval abstract_da_pcs

proof tLOAD
"
  intros t l s.
   unfold a_tLOAD d_tLOAD;
    functional induction d_tLOAD params t l s;trivial;
    repeat progress (Rewall;simpl;trivial);
    auto with jcvmdb;eauto with jcvmdb.
  replace a_Int with (alpha_da_val (d_Int z0));auto with jcvmdb.
  replace (a_RA n) with (alpha_da_val (d_RA n));auto with jcvmdb.
"

into jcvm_a_functions log jcvm_log
```

6. CertiCartes as a case study

CertiCartes is an in-depth feasibility study in proving the correctness of bytecode verification for Java Card 2.1, using the proof assistant Coq [17]. CertiCartes contains executable specifications of the three JCVMs (defensive, offensive and typed) and of the BCV, and a proof of the correctness of the BCV. It is structured in two separate modules: a first module JCVM, which includes the construction of three virtual machines and their cross-validation, and a second module BCV, which includes the construction and validation of the BCV from the typed JCVM.

In this section, we show how Jakarta can be applied to build offensive and typed machines and provide the proof of cross validation for the JCVM. [5, 6, 7] provide further details.

6.1. The Java Card VM

The Java Card Virtual Machine involves more refined datatypes and structures than the ones in our running example. We will describe below how this virtual machine has been formalized.

Modeling programs. Programs are formalized in a neutral mathematical style based on (first-order, non-dependent) datatypes and record types, and the corresponding machinery: case analysis and structural recursion for datatypes, formation and selection for record types. For example, programs are represented by the record type:

```
Record jcprogram := {
  interfaces : (list Interface);
  classes    : (list Class);
  methods    : (list Method);
  sheap_type : (list type)
}.
```

where the types `Interface`, `Class`, `Method` and `type` are themselves defined as record types. `sheap_type` is used for initialization purposes, to determine types of variables declared as static in the program. For simplicity, we only deal with closed programs hence the packages `java.lang` and `javacard.framework` are an integral part of programs.

Modeling memory. Memory is modeled in a similar way. For example, states are formalized as a record consisting of the heap (containing the objects created during execution), the static fields image (containing static fields of classes) and a stack of frames (environments for executing methods). Formally:

```
Record d_state := {
  hp : heap;
  sh : sheap;
  st : stack
}.
```

In order to account for abrupt termination (that may arise because of uncaught exceptions or because the program being executed is ill-formed), we also introduce a type of return states, defined as a sum type:

```
Inductive d_rstate :=
| d_Normal   : d_state → d_rstate
| d_Abnormal : exception → d_rstate.
```

The definition of (return) states and of all components of the memory model are parameterized by two generic notions of value, which we leave unspecified here. It will later be instantiated together with the memory model for defensive, offensive and typed JCVMs. The following table gives the notion of value attached to each JCVM:

Virtual Machine	Defensive	Offensive	Typed
Value	Number with Type	Number	Type

We see that defensive values contain a type tag, that represents the property checked at runtime by the defensive machine (and thus checked statically by the BCV). Type tags are removed in order to obtain the offensive machine, whereas the concrete values (numbers) are removed in order to derive the typed machine. We describe more precisely this mechanism in the following.

Modeling the defensive JCVM. First, we instantiate the memory model with a notion of typed value. In CertiCartes, types of defensive values are represented by the constructors of inductive types.

```
Inductive d_val_prim :=
| d_VReturnAddress : bytecode_idx → d_val_prim
| d_VBoolean       : Z → d_val_prim
| d_VByte          : Z → d_val_prim
| d_VShort         : Z → d_val_prim
| d_VInt           : Z → d_val_prim.

Inductive d_val_ref :=
| d_VRef_null      : d_val_ref
| d_VRef_array     : type → heap_idx → d_val_ref
| d_VRef_instance  : class_idx → heap_idx → d_val_ref.
```

```
Inductive d_val :=
| d_VPrim          : d_val_prim → d_val
| d_VRef           : d_val_ref → d_val.
```

where all types suffixed by _idx are actually nat. For example a defensive value 8 of type short will be represented by (d_vPrim (d_vShort (8))), and a null reference by (d_vRef d_vRef_null).

We thus obtain a type of defensive states d_state and return defensive states d_rstate. Then, we model the defensive semantics of each Java Card bytecode as a function d_state → d_rstate. Typically, this function extracts values from the state, performs type verification on these values, and extends/updates the state with the results of executing the bytecode. An example is given in Section 6.2.1 (in Coq syntax) for the bytecode ifnull.

Finally, one-step defensive execution is modeled as a function d_exec: d_state → d_rstate which inspects the state to extract the Java Card bytecode to be executed and then calls the function yielding its semantics.

Modeling the offensive JCVM. First, we instantiate the memory model with a notion of untyped value so as to obtain a type of offensive states o_state and return offensive states o_rstate. Which gives in Certicartes:

```
Definition o_val_prim := Z.
Definition o_val_ref := Z.
Definition o_val := Z.
```

Notice that a mapping from d_val_prim to o_val_prim will coerce nat to Z because of the type of the argument of d_VReturnAddress.

Then, one-step offensive execution o_exec: o_state → o_rstate is modeled in the same way as one-step defensive execution, but all verifications and operations related to typing are removed.

Modeling the typed JCVM. The typed JCVM that is used for bytecode verification operates at the level of types. Formally, we define the type a_state of typed states as being the typed frame. The memory model is further simplified because: (1) the typed JCVM operates on a method per method basis, so only one frame is needed; (2) since values are removed from this machine, the heap is not needed: types of objects are stored directly in the operand stack; (3) the removal of values transforms the static heap into a list of types. Because of the field sheap_type of the record type jcprogram that contain exactly this information, the typed static heap becomes redundant.

To obtain typed values, we remove numerical values of the defensive values, except for the numerical value of return addresses which is a static information and is used to determine the control flow for subroutines instructions. We give below the sum types of typed values.

```
Inductive a_val_prim :=
| a_ReturnAddress : bytecode_idx → a_val_prim
| a_Void          : a_val_prim
| a_Boolean       : a_val_prim
| a_Byte          : a_val_prim
| a_Short         : a_val_prim
| a_Int           : a_val_prim.

Mutual Inductive a_val :=
| a_Prim          : a_val_prim → a_val
| a_Ref           : a_val_ref → a_val
with a_val_ref :=
| a_Ref_null      : a_val_ref
| a_Ref_array     : vmtype → a_val_ref
| a_Ref_instance  : class_idx → a_val_ref
| a_Ref_interface : interf_idx → a_val_ref.
```

One-step typed execution is non-deterministic because branching instructions lead to different program points and hence different states. We use a list to collect all return states. Therefore one-step typed execution is modeled as a function `a_exec: a_state → (list a_rstate)`, where `a_rstate` is the type of return typed states.

Cross-validation of the JCVMs Likewise the example from Section 2, cross-validation is a prerequisite for establishing the correctness of the BCV for the JCVM. The soundness of the offensive abstraction is expressed by the same commuting diagram as Figure 2. However, for the typed JCVM, the commutation of the diagram (Figure 11) must also take into account the notion of subtyping that appears between defensive and typed states, due to the use of dynamic types and method invocations.

$$\begin{tikzcd}
\texttt{d\_state} \arrow[r, "\texttt{d\_exec}"] \arrow[d, "\texttt{alpha\_da}"] & \texttt{d\_rstate} \arrow[d, hook, "\preceq\texttt{alpha\_da\_rs}"] \\
\texttt{a\_state} \arrow[r, "\texttt{a\_exec}"] & \texttt{(list a\_rstate)}
\end{tikzcd}$$

Figure 11. Commutative diagram of defensive and typed execution

Further, this diagram is restricted to instructions that remain in the same frame. Method invocations or return instructions, as well as exceptions, are handled separately in our framework (without the use of

Jakarta) and their correctness is expressed differently (see [5] for more details). Corresponding proofs for correctness are incorporated in the Framework phase (see Section 2.3).

6.2. An example: the bytecode `ifnull`

In this section, we illustrate the benefits of our package in establishing correctness results between defensive and offensive VM for the `ifnull` Java Card bytecode. It is a simple bytecode with only five rules, yet it needs some commands to drive the abstraction and make the correctness proof succeed.

We describe the whole process of obtaining a correct (certified) offensive version of this bytecode. We use Coq in the following, but the methodology applies to any proof system. The steps will be the following:

1. (Section 6.2.2) Define the abstraction functions, create a first script with no particular command (except the preamble).
2. Apply the JTK.
3. (Section 6.2.3) Based on the result (16 abstracted functions in our example) and warnings returned by the JTK, add new commands to the script to suppress warnings and go back to 2.
4. (Section 6.2.4) Compile (proof check) the generated Coq files: defensive machine, offensive machine and default proofs of commutation lemmas (one file per function).
5. If a default proof does not succeed for some file/lemma, then launch Coq on the corresponding file and try to finish the proof interactively, which leads to several possibilities:
   - If we manage to prove the lemma, then the process is finished.
   - If the lemma cannot be proved because of wrong premises, then add or modify commands in the script to correct them. In particular use the `redundant` variant of commands. Re-apply the JTK and go back to 4.
   - If the lemma cannot be proved because functions actually do not commute correctly, then make the necessary corrections either in the defensive machine, in the abstraction functions or in the abstraction script, go back to 2.

   Modifications made to proof scripts during step 5 are kept when regenerating the proofs, thanks to a mechanism based on the `diff` command, so that proofs that are still correct do not need to be modified again. It would also be possible to follow the approach of the Why tool [22] that distinguishes more clearly the generated parts of a file from the user input.

6.2.1. *Defensive machine*

The instruction `ifnull` is a branching bytecode that compares the reference at the top of the operand stack to the null reference and jumps to the bytecode index given as operand (`b` in the code below) if the comparison succeeds or to the following instruction (`succ (d_pc h)`) otherwise.

The defensive specification of `ifnull`, that tests if values in the stack are of the expected type, is written in JSL as follows:

```
function d_ifnull : bytecode_idx → d_state → d_rstate :=

stack_f state → Nil
  ⇒ d_ifnull b state → d_Abnormal state_error state;

stack_f state → Cons h lf,
head (d_opstack h) → Value (d_VPrim v0),
  ⇒ d_ifnull b state → d_Abnormal type_error state;

stack_f state → Cons h lf,
head (d_opstack h) → Value (d_VRef vr),
res_null vr → True
  ⇒ d_ifnull b state →
        d_update_frame (d_update_pc b
          (d_update_opstack (tail (d_opstack h)) h)) state;

 stack_f state → Cons h lf,
 head (d_opstack h) → Value (d_VRef vr),
 res_null vr → False
   ⇒ d_ifnull b state →
        d_update_frame (d_update_pc (succ (d_pc h))
                       (d_update_opstack
                         (tail (d_opstack h)) h)) state;

 stack_f state → Cons h lf,
 head (d_opstack h) → Error
   ⇒ d_ifnull b state → d_Abnormal opstack_error state.
```

We give also the auxiliary function `res_null`, as we will illustrate the use of abstraction commands on this function. `res_null` takes a reference and returns `true` if it is the null reference and `false` otherwise.

```
function res_null : d_val_ref →bool :=
 vr →d_VRef_null ⇒res_null vr →True;
 vr →d_VRef_array t hi ⇒res_null vr →nat_is_zero hi;
 vr →d_VRef_instance ci hi ⇒res_null vr →nat_is_zero hi;
 vr →d_VRef_interface ii hi ⇒res_null vr →nat_is_zero hi.
```

6.2.2. *Abstraction functions and first script*

Datatypes for the offensive JCVM specification, and the abstraction functions to produce offensive types from defensive ones are given by the user in JSL format:

```
type o_val_prim = Z.
type o_val = Z.
type o_val_ref = Z.

function alpha_do_val_prim : d_val_prim →o_val_prim :=
 ⇒alpha_do_val_prim (d_VReturnAddress v) →(nat2z v);
 ⇒alpha_do_val_prim (d_VBoolean v) →v;
 ⇒alpha_do_val_prim (d_VByte v) →v;
 ⇒alpha_do_val_prim (d_VShort v) →v;
 ⇒alpha_do_val_prim (d_VInt v) →v.

function alpha_do_val_ref : d_val_ref →o_val_ref :=
 ⇒alpha_do_val_ref d_VRef_null →ZERO;
 ⇒alpha_do_val_ref (d_VRef_array t hp) →hp;
 ⇒alpha_do_val_ref (d_VRef_instance c hp)→ hp;
 ⇒alpha_do_val_ref (d_VRef_interface i hp) →hp.

function alpha_do_val : d_val →o_val :=
 ⇒alpha_do_val (d_VPrim x) →alpha_do_val_prim x;
 ⇒alpha_do_val (d_VRef r) →alpha_do_val_ref r.

...

function alpha_do_heap : d_heap →o_heap := ...

function alpha_do_sheap : d_sheap →o_sheap := ...

function alpha_do_stack : d_stack →o_stack := ...

function alpha_do : d_state →o_state :=
 ⇒alpha_do (Build_d_state sh hp s)
   → Build_o_state (alpha_do_sheap sh)
                (alpha_do_heap hp) (alpha_do_stack s).

function alpha_do_rs : d_rstate →o_rstate :=
 ⇒alpha_do_rs (d_Normal js) →o_Normal (alpha_do js);
 ⇒alpha_do_rs (d_Abnormal x) →o_Abnormal x.
```

The initial script for abstracting `ifnull` is given below. It contains the name of the function to abstract, the name of the abstraction functions and the command to remove all occurrences of `type_error`.

```
abstract ifnull with alpha_do_val_prim alpha_do_val
```

```
 alpha_do_val_ref alpha_do_frame alpha_do_stack alpha_do
 alpha_do_sheap alpha_do_rs alpha_do_heap
prefix o_

delete type_error
```

At this point, all the information needed to perform the abstraction is given: JSL defensive machine, JSL offensive datatypes, JSL abstraction functions and JSL abstraction script. The abstraction generates a set of 16 function definitions (in 16 files) for `ifnull` and auxiliary functions, and returns a warning:

```
WARNING! o_res_null is not deterministic because
of the following rules couples:
 o_res_null_1 and o_res_null_2 with substitution [ (hi , ZERO) ]
```

Overlapping rules have been detected in the generated function `o_res_null`. We must refine the basic abstraction process for this functions by looking at the generated function and modifying the initial script.

6.2.3. *Second abstraction script*

The script above has generated the following definition for `o_res_null`:

```
function o_res_null : o_val_ref →bool :=
  vr →ZERO ⇒o_res_null vr →True;
  vr →hi ⇒o_res_null vr →nat_is_zero (z2n hi).
```

The first rule of `res_null` (precondition `vr=VRef_null`) has been abstracted into a rule having precondition vr=`ZERO`. Rules 2, 3 and 4 of `res_null` have been collapsed into a single abstracted rule `o_res_null_2`.

We see (but Jakarta did not, see limitation of Jakarta in Section 4.6) that the first rule is subsumed by the second because `(nat_is_zero(z2n ZERO))` is equal to `True`. Therefore a good solution is to remove the first rule by adding a **`delete`** `[redundant]` command to the script. It will not add a new premise since the semantics of `o_res_null` is not altered. Finally, the script is the following:

```
abstract ifnull with alpha_do_val_prim alpha_do_val
 alpha_do_val_ref alpha_do_frame alpha_do_stack alpha_do
 alpha_do_sheap alpha_do_rs alpha_do_heap
prefix o_

delete type_error
delete [redundant] res_null.1
```

and, once the JTK has been applied again, the new offensive versions of `res_null` is correct, we give the translation of `res_null` and `ifnull` into Coq in Figure 12.

```
Definition o_res_null (vr : o_val_ref) := nat_is_zero (z2n vr).

Definition o_ifnull (b : bytecode_idx) (s : o_state) :=
 match o_stack_f s with
  | Nil ⇒o_Abnormal state_error
  | Cons h lf ⇒
     match head (o_opstack h) with
     | Some v ⇒
        match o_res_null v with
        | True ⇒
          o_update_frame (o_update_pc b
                         (o_update_opstack
                           (tail (o_opstack h)) h)) s
        | False ⇒
          o_update_frame (o_update_pc (succ (o_pc h))
                         (o_update_opstack
                           (tail (o_opstack h)) h)) s
        end
     | None ⇒o_Abnormal opstack_error
     end
 end.
```

Figure 12. Final Coq version of ifnull and res_null

6.2.4. *Lemmas and predicates*

By applying the offensive script shown above, we obtain 16 Coq files (one per abstracted function) containing commutation lemmas with their premises as in Figure 10. The predicates and the lemmas generated for `res_null` and `ifnull` are the following (we do not show the default proof script):

```
(* res_null *)
Lemma res_null_eq : ∀(v : d_val_ref),
(o_res_null (alpha_do_val_ref v)) = (res_null v).
intro v.
<default proof>
Save.

(* Storing of the lemma in database for further proofs automation *)
Hint Rewrite [res_nul_eq] in jcvmdb.
Hint Resolve [res_nul_eq] in jcvmdb.

(* ifnull *)
Definition prec_ifnull (b : bytecode_idx) (j : d_state) :=
¬(d_ifnull b j) = (d_Abnormal type_error)).

Lemma ifnull_eq : ∀(b : bytecode_idx) (j : d_state),
(prec_ifnull b j) →
  (o_ifnull b (alpha_do j)) = (alpha_do_rs (d_ifnull b j)).
```

```
intros b j.
<default proof>
Save.
```

Notice that no predicate is generated for `res_null`.

With this last script, it is thus formally proved that `o_ifnull` and `ifnull` commute in the sense of the diagram of Figure 2. Unlike for `ifnull`, the abstraction and the commutation proof of most of the byte-codes (85%) do not require any user interaction. Remaining lemmas are intrinsically more complex to discharge automatically and fall beyond the scope of the tool. Nevertheless, Jakarta leaves the user with the lemma and its default proof script, and generally it is enough to add a few more tactics to make it work. These modifications to default scripts are then kept when regenerating proofs.

6.3. Synthesis of the offensive and typed JCVM

Using the techniques described in the previous example, we have been able to generate from the defensive JCVM (around 6,000 lines long) the offensive JCVM (5,000 lines) and the typed JCVM (4,000 lines) as well as cross validation proofs for these abstractions.

6.3.1. *The offensive JCVM*

Figure 13 contains the script (around 30 lines long) needed to obtain the offensive JCVM. The beginning of the script is similar to the script for `ifnull` (Section 6.2.3). It begins with a preamble, which specifies the function to abstract, the abstraction functions and the prefix used for the generated functions. Then, the remaining of the script is divided in two blocks of discarding commands and substitution commands.

In that script, the command **`drop`** is used to remove some arguments from the signature of a function. For example, the function `d_GETSTATIC` takes as a first argument some typing information that becomes vacuous in the offensive JCVM.

In the following block, we used substitution commands, such as coercions, replacement of terms or optimization with the **`select`** command to replace functions that behave as a projection function after abstraction. An example of such function is given by `vp2z`. In the defensive JCVM, this function maps a types value to its corresponding numerical value. In the offensive JCVM, this function behave as the identity.

6.3.2. *The typed JCVM*

The abstraction for the type-abstract JCVM is much more complex. Indeed, there is no more numerical values, heap, stack (only the top-most frame is kept) or static heap. The datatypes and the abstraction functions for the typed JCVM are the following:

```
(∗ Preamble ∗)

abstract exec_jcvm with alpha_do_val_prim alpha_do_val ...
prefix o_

(∗ Discarding commands ∗)

delete type_error
delete [redundant] res_null.1
delete ‘t in d_LOAD.9.3.1 d_LOAD.17.3.1 d_LOAD.22.3.1
 d_NEWARRAY d_INC d_CONST d_STORE.9.5.1
 d_STORE.17.5.1 d_STORE.21.5.1 d_STORE.27.5.1
delete ‘t0 in d_PUTSTATIC putfield_obj aSTORE

drop &d_GETSTATIC@1 &d_TABLESWITCH@1 &d_LOOKUPSWITCH@1

(∗ Substitution commands ∗)

coercion n2z z2n

replace ((trhi2vr (vr2tr vr) (absolu (v2z v)))) by
  ‘v in d_putfield_obj.36.10.2

select v2z@1 vp2z@1 vr2z@1 vr2hi@1 tpz2vp@2
```

Figure 13. Offensive abstraction script

```
function alpha_da_val_prim : d_val_prim →a_val_prim :=
⇒alpha_da_val_prim (d_VReturnAddress v) →a_ReturnAddress v;
⇒alpha_da_val_prim (d_VBoolean v) →a_Boolean;
⇒alpha_da_val_prim (d_VByte v) →a_Byte;
⇒alpha_da_val_prim (d_VShort v) →a_Short;
⇒alpha_da_val_prim (d_VInt v) → a_Int.

function alpha_da_val_ref : d_val_ref →a_val_ref := ...
function alpha_da_val : d_val →a_val := ...
function alpha_da_lval : list d_val →list a_val := ...
function alpha_da_oval : option d_val →option a_val := ...
function alpha_da_loval :
 list (option d_val) →list (option a_val) := ...

function alpha_da_frame : d_frame →a_frame :=
⇒alpha_da_frame
   (Mk_d_frame ops lv c p b m)
 → Build_a_frame (alpha_da_lval ops)
              (alpha_da_loval lv) c p b m.

function alpha_da_stack : d_stack →a_stack :=
⇒alpha_da_stack Nil  → Nil;
```

```
⇒alpha_da_stack (Cons x y) →(Cons (alpha_da_frame x) Nil).

function alpha_da_jcvm_state : d_state →a_state :=
⇒alpha_da_jcvm_state (Mk_d_state sh hp s) →(alpha_da_stack s).
```

Due to this significant change, the script for the type abstraction, given in Figure 14, is bigger (around 150 lines).

```
(* Preamble *)

abstract exec_jcvm with alpha_da_val_prim alpha_da_val ...
prefix a_

(* Discarding commands *)

delete 'ThrowException abortMemory res_null
delete test_exception_putstatic_ref
delete test_security_exception_checkcast
delete ...

drop &result_astore@{5,6} &d_ARITH@1
drop &res_pc@1 &res_pc2@1
drop ...

(* Substitution commands *)

coercion tv2t tvr2tr tvp2tp t2tv

replace (get_obj_class_idx nhp) by (get_a_val_ref_class_idx vr)
 in d_INVOKEINTERFACE.{5-11}.8.1 d_INVOKEVIRTUAL.{5-10}.7.1
replace d_INVOKEVIRTUAL.5.12.2 d_INVOKEINTERFACE.5.14.2
 d_INVOKESPECIAL.5.11.2 by
 (a_Normal
  (Cons (Build_a_frame (app_return_type l' (snd (signature m)))
       (a_locvars h) (a_method_loc h) (succ (a_pc h)))
   Nil))
replace ...

determine a_res_pc a_res_pc2 both_bool a_ICMP

(* Proofs commands *)

addprecond < t ≠ ReturnAddress > to tpz2vp
addprecond ...
```

Figure 14. Typed abstraction script (Excerpts)

In the discarding block, we remove all aspects related to numerical values such as exceptions and memory errors. In the substitution block, we replace dynamic lookups of the execution by static ones. For instance we provide directly the result for method invocation. As for script for our running example, we use the command **determine** to make deterministic the generated function for conditional instructions. Finally, in the proof command block, we use the command **addprecond** to add preconditions (that will be propagated to calling functions). The concerned functions are used in the formalization under certain conditions and the added preconditions restrict the use of these functions to these particular cases. For instance, the function `tpz2vp` is never called in the code with the argument `ReturnAddress`, the command **addprecond** expresses this fact, that is necessary for the cross validation.

7. Conclusions and related work

Formal specification and formal verification help understanding and improving informal specifications of a programming language semantics. Unfortunately, providing an extensive formal specification of a realistic programming language, let alone proving formally some of its properties such as type soundness, remains labour-intensive. It is thus important to build appropriate environments that provide coherent support for formal specification, formal verification and provably correct implementations of programming languages.

This paper focuses on an environment for proving the correctness of bytecode verification in low-level programming languages for mobile and embedded code, such as the JVM. Our environment Jakarta incorporates general-purpose tools such as proof assistants as well as specialized tools that are explicitly designed to reason about typed low-level programming languages. In addition to presenting its underlying principles, we have shown how Jakarta can be used to good effect for certifying the Java Card platform.

7.1. Related work

Abstract interpretation The idea of deriving abstract functions from a concrete function and an abstraction function already appears in the seminal work on abstract interpretation by Cousot and Cousot [18]. Over the last 25 years, the framework of abstract interpretation has developed extensively and has been successfully exploited in too many contexts to be listed here.

Part of the cross validation may be viewed as a simple instance of abstract interpretation. Indeed, for the construction of the typed machine from the defensive machine, one may build a Galois connection between the powerset of defensive states and the lattice of typed states from the abstraction function [37, page 235]. On the other hand it remains unclear how to derive the offensive virtual machine from the defensive virtual machine using the framework of abstract interpretation.

Prototyping environments and proof assistants Formal semantics provide an unambiguous reference description of the expected operational behavior of languages. Yet such semantics are large, technically involved and difficult to build for the non-expert. Some of these difficulties can be solved by using a dedicated prototyping environment that helps addressing some/most administrative aspects of formal semantics. A number of tools have been designed for this purpose, including ASF+SDF [21], ASMGofer [12], Centaur [14], Letos [25] and RML [39]. While very diverse in their design and functionalities, these tools all provide a readable format for specifying the formal semantics of a programming language, and support to execute the semantics, which in particular allows to check that the semantics reflects the intended behavior of the language. However, prototyping environments most often lack functionalities to reason formally about the semantics, in contrast to proof assistants which support sophisticated reasoning. There is thus some interest in integrating prototyping environments and proof assistants; yet despite preliminary work in this direction [44], no current tool integrates smoothly prototyping environments and proof assistants. Consequently, formal semantics is written directly in the specification language of the proof assistant, which often lacks constructs for modular and incremental specifications. On the positive side, some proof assistants feature a program extraction mechanism which enables specifications to be executed.

Formal specification and verification of the Java Card platform There is a large body of machine-checked specifications of the J(C)VM, see [26, 33] for a survey. Many of these works use the methodology instrumented in our work, see e.g. [2, 7, 10, 30, 35]. However, the methodology is generally not made explicit and the offensive virtual machine is not considered. Further, there are a few extensive accounts of the J(C)VM that do not adopt the methodology instrumented by Jakarta, including the work of Klein, Nipkow, and Wildermoser [28, 29]. In the J-Book [41], Börger, Schmid, and Stärk discuss the relationship between the offensive and defensive JVM, and in particular the derivation of a defensive JVM from an offensive one, but the discussion remains informal.

There are also machine-checked proofs of type soundness for .NET [24, 43]. This work is more closely related to ours in the sense that [43] explicitly aims at developing tools to automate type soundness proofs. The major difference with our work is that they do not pursue cross-machine validation, and opt instead for a standard type soundness proof.

7.2. Future work

Incremental and modular specifications While expressive enough to specify the JCVMs, the JSL lacks constructs for modular specifications. Hence JSL specifications are monolithic and cannot be constructed in an incremental fashion.

In order to follow the principles of specification in-the-large, we intend to extend the JSL with subtyping, a module system, and primitives for incremental and modular specifications. We are in particular interested in mechanisms that allow a separation of concerns between the operational semantics captured by the offensive virtual machine, and the safety and security checks performed by the typed virtual machine. Put it otherwise, we would like to be able to synthesize, from an offensive virtual machine, and several security automata that perform the necessary checks, the corresponding defensive virtual machine, and the cross-validation proofs. Technically, it amounts to define mechanisms to define a function $f' : \sigma' \rightarrow \sigma'$ from a previously defined function $f : \sigma \rightarrow \sigma$, where σ and σ' are record types with $\sigma' \leq \sigma$ (i.e. σ' has more fields than σ).

Such a mechanism would prove most useful in the context of reasoning about the Java Card platform. *E.g.* it would allow to construct from our current defensive JCVM another defensive JCVM that also performs checks w.r.t. confidentiality of resource control, or to make our existing specification more precise w.r.t. its memory model.

Proof automation Commutation lemmas are proved by case analysis and equational reasoning. For the latter, we use the `AutoRewrite` tactic of Coq: once the commutation of a function is proved, the corresponding lemma is put in a rewriting database. This database is then heavily used in the equational reasoning in the proof of the following lemmas. This proof mechanism is rather primitive and slow. We expect that more efficient proof automation is possible, and in particular that equational reasoning can be automated by exploiting connections between Coq and external tools, such as Elan [13] or Spike [15]. In fact, we have been experimenting with Elan, using the interface to Coq reported in [36, 1], and with Spike, using a JPI module that translates JSL specifications to Spike. The first experiment, which was carried with Q.-H. Nguyen, is currently inconclusive because, in order to be used in our context, the interface between Coq and Elan should be packaged as a tactic, which is not the case for the moment for technical reasons. The second experiment, which was carried with S. Stratulat, was more successful in that Spike was able to establish automatically the commutation properties for a large set of instructions. This experiment, which is reported in [9], suggests that first-order theorem proving techniques could be advantageously exploited in the context of platform verification for smartcards.

On a more general level, we intend to generalize our mechanism for generating auxiliary lemmas (proof and statements). It seems possible to provide a little language for describing the generic form of the auxiliary lemmas needed to prove the correctness of a particular abstraction. This language should also describe the form of the standard tactics which should be applied to (try to) prove those lemmas. This would give more flexibility to the user and allow him to generate very accurate proof scripts for different abstractions.

Moreover automatic script generation as described in this paper can be considered as an extension to a given tactic language [20]. One can specify general strategies like “what default tactics should be used to prove a lemma”, “how to use previous lemmas in the following proofs”. This is hardly automated in theorem provers like Coq, and a generic format to describe such strategies and a tool to generate scripts from this description could be a great improvement in domains where some regularity can be found in the proof methods. This issue is closely related to proof planning [16].

Other case studies One line of future research is to exploit our environment to validate enhanced type systems for Java(Card). Examples of such type systems can be found e.g. in [8, 23, 31, 42], and address properties such as confidentiality, initialization, or locking/inlocking. More generally, we are interested in understanding whether our methodology applies to other typed low-level languages such as TAL [19] or .NET.

Acknowledgments Part of this work was supported by the IST project VerifiCard.

References

1. C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in COQ. In J. Despeyroux, editor, *Proceedings of LFM'00*, 2000. Rapport Technique INRIA.
2. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 335 – 351. Springer-Verlag, 2003.

3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.
5. G. Barthe and G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. In *Proceedings of FASE'04*, volume 2984 of *Lecture Notes in Computer Science*, pages 99–113. Springer-Verlag, 2004.
6. G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 32–45. Springer-Verlag, 2002.
7. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
8. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05.* ACM Press, 2005. To appear.
9. G. Barthe and S. Stratulat. Using Implicit Induction Techniques for the Validation of the JavaCard Platform. In R. Nieuwenhuis, editor, *Proceedings of RTA'03*, volume 2706 of *Lecture Notes in Computer Science*, pages 337 – 351. Springer-Verlag, 2003.
10. G. Betarte, B. Chetali, E. Giménez, C. Loiseaux, and O. Ly. Formal Modeling and Verification of the Java Card Security Architecture: from Static Checkings to Embedded Applet Execution. In *Proceedings of ESMART'02*, 2002.
11. M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.
12. E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.
13. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *The Elan V3.4. Manual*, 2000.
14. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM Press, 1988.
15. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, January 1997.
16. A. Bundy. The use of explicit plans to guide proofs. In *Proceeding of CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120. Springer-Verlag, 1988.
17. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
18. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL'77*, pages 238–252. ACM Press, 1977.
19. K. Crary and G. Morrisett. Type structure for low-level programming languages. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proceedings of ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, 1999.

20. D. Delahaye. A Tactic Language for the System Coq. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR'00*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer-Verlag, 2000.
21. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: an algebraic specification approach.* AMAST Series in Computing. World Scientific, 1996.
22. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
23. S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, November 1999.
24. A.D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of POPL'01*, pages 248–260. ACM Press, 2001.
25. P. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software–practice and experience*, 29(5):1379–1416, September 1999.
26. P. Hartel and L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.
27. JavaCard Technology. `http://java.sun.com/products/javacard`.
28. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
29. G. Klein and M. Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 30(3-4):363–398, December 2003.
30. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
31. C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290(1):741–778, October 2002.
32. X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
33. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
34. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.00*, 2000.
35. J. Strother Moore, R. Krug, H. Liu, and G. Porter. Formal Models of Java at the JVM Level A Survey from the ACL2 Perspective. In S. Drossopoulou, editor, *Proceedings of Formal Techniques for Java Programs*, 2001.
36. Q.-H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002.
37. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
38. T. Nipkow. Verified Bytecode Verifiers. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, 2001.
39. M. Petersson. *Compiling Natural Semantics.* PhD thesis, Linköping University, 1995.
40. K. Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, TU Münich, 1999.
41. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer-Verlag, 2001.

42. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.
43. D. Syme and A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In M. Baaz and A. Voronkov, editors, *Proceedings of LPAR'02*, volume 2514 of *Lecture Notes in Computer Science*, pages 418–434, 2002.
44. D. Terrasse. *Vers un environnement d'aide au développement de preuves en Sémantique Naturelle*. PhD thesis, Ecole Nationale des Ponts et Chaussées, 1995.