Abstract

*

†

∗ To do

† see main.tex for "to do"

Contents

1 Introduction

Virtual machines, such as the Java Virtual Machine, JavaCard 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. A crucial such security function is the bytecode verifier, which performs a static analysis on programs and rejects potentially insecure programs.

Over the last few years, there have been a number of projects to specify and verify such execution platforms (virtual machines and bytecode verifiers); these projects aim at i) detecting ambiguities, incoherences or even flaws in specifications; ii) providing the formal guarantees required by Common Criteria evaluations at highest levels; iii) demonstrating that machine-checked semantics of reaslistic programming languages has become feasible. While such formal developments have been successfully employed for such purposes see e.g. [?], their realization is labour-intensive, and suffers from a lack of methodologies, techniques and tools that would alleviate, through automatic procedures, the task of carrying such endeavours.

This paper describes Jakarta, a dedicated environment for specifying and verifying execution platforms, and illustrates its benefits for the certification of the JavaCard Platform. In a nutshell, Jakarta builds upon a robust methodology to certify virtual machines and bytecode verifiers, and reduces the cost of developing and maintaining such formal developments by providing tool support for applying this methodology.

The methodology The bytecode verifier is designed to enforce statically that programs will not go wrong, by guaranteeing that some invariant P is maintained throughout execution. The methodology requires to define three virtual machines:

- a reference, so-called defensive, virtual machine where values are tagged by their type and where typing information, which is related to P, is verified at run-time;
- an abstract virtual machine that manipulates types and that is used for bytecode verification;
- a "standard", so-called offensive, virtual machine, where values are untyped, and which relies on successful bytecode verification to eliminate type verification at run-time.

Since the type information is related to the property P to be checked, these three machines are sometimes called P*-defensive*, P*-abstract* and P*-offensive*. The advantages of our methodology is three-fold:

1. the offensive and abstract virtual machines can be derived from the defensive virtual machine using abstraction techniques;
2. the correctness of bytecode verification is now crisply stated as: "the offensive and defensive virtual machines coincide on those programs that pass bytecode verification";
3. the correctness of bytecode verification is now a consequence of the correctness of (a) the abstractions of the defensive virtual machine into an offensive and an abstract virtual machine respectively; (b) the derivation of the bytecode verifier from the abstract virtual machine, essentially using a dataflow analysis to perform a fixpoint calculation and compositional techniques to proceed on a method-per-method basis.

Notice that (a) is specific to the virtual machine under consideration but (b) is generic and can be applied to different properties such that type safety, information flow or resource control. Thus the methodology itself is useful for specifying and verifying execution platforms, since it factors out the generic components of the proof, which are treated elsewhere [7], from the application specific ones, which form the subject of this paper.

The environment The methodology provides clear guidelines on how to proceed with the specific parts of the formal development. The purpose of the environment is to provide adequate tool support for applying the guidelines. In particular the environment is designed to:

1. generate automatically the offensive and abstract machines from the specification of the defensive machine, using abstraction techniques;

2. cross-validate (almost) automatically these machines, as in (a) above, using a proof assistant or an automatic theorem prover.

To this end, the environment relies on a dedicated specification language in which virtual machines can be conveniently specified. The language, which we call the *Jakarta Specification Language*, is sufficiently simple so that: i) specifications can be translated automatically to several proof assistants, automatic theorem provers, prototyping environments, and programming languages; ii) transformations and validations of specifications may be mechanized with limited user interaction; iii) specifications may benefit from many techniques that usually fall out of the realm of formal verification, e.g. code generation or test case generation; we do not address this point here, but see Section **??**.

Figure **??** synthetises the process of certifying a platform with Jakarta, using the methodology described above. The input files provided by the user are:

i. The JSL specification of the P-defensive machine, which checks the property P at runtime, using a P-dedicated type system. Writing this machine is the core of user work;

ii. The JSL datatypes of the target machines (i.e. P-offensive and/or P-abstract);

iii. The description of the abstraction, which consists of the JSL definition of the abstraction functions, and of he *abstraction script*. The former specifies how to transform defensive states into target states), while the latter includes commands that 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 corrections must be made to the generic abstraction mechanism in order to correct/optimize the generated machine. For cross-validation purposes, it is also possible to give proof informations in the script.

Assuming we select a proof assistant for the output, the files generated by Jakarta are:

1. the translation of the P-defensive machine (i) into the proof assistant;

2. the translation of the datatypes of the target machine (ii) into the proof assistant;

3. the functions of the target machine, generated from i, ii and iii. This is a JSL specification;

4. The translation of the functions of the target machine (3) into the proof assistant;

5. The proof scripts that establish the defensive and target machines commute. These proof scripts must be executed, and generally partially completed, to provide the final cross-validation proofs.

These tasks are performed by different components:

- the *Jakarta Transformation Kit JTK*, which allows to derive P-offensive and P-abstract machines from a P-defensive one;

- the *Jakarta Prover Interface JPI*, which contains the different translators from JSL to target environments/languages. The current target environments/systems are: OCaml [30], Coq [17], Isabelle [34], PVS [37] and Spike [15]. Translations from Coq and OCaml to JSL are also implemented;

- the *Jakarta Automation Kit JAK*, which provides a mechanism to generate auxiliary commutation lemmas necessary to the cross-validation proofs, and a proof script generation mechanism, that uses purpose-build tactics to discharge the generated lemmas.

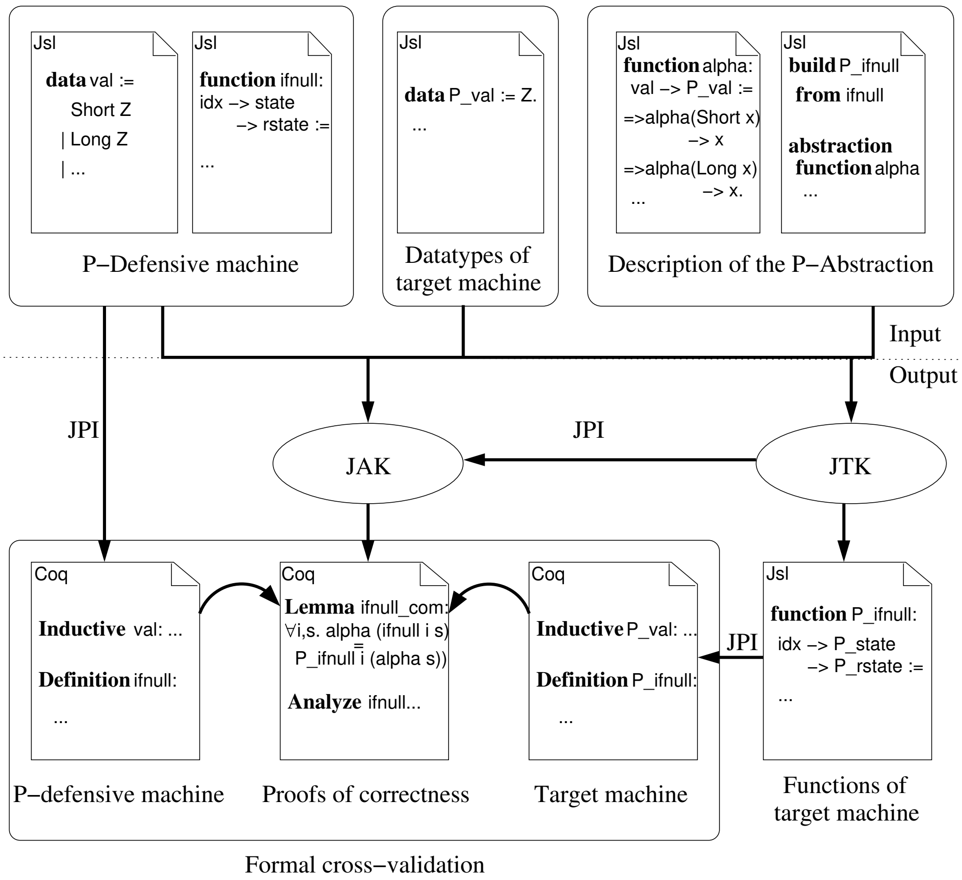

Figure 1: The methodology of Jakarta

Case study: the JavaCard Platform Launched in 1996 by Sun, JavaCard [24] is the standard programming language for new generation smartcards. As emphasised by the JavaCard Forum, JavaCard is an ideal choice for smartcards, since: i) it brings to the resource constrained environment of smartcards many benefits of Java such as object orientation, safety through typing, and portability; ii) it enables multiple applications to coexist securely on the same card, and the possiblity to load new applications after the card has been issued. The JavaCard architecture shares many similarities with Java: JavaCard applets are written in a subset of the Java language that excludes some items such as large datatypes or arrays of arrays, and using JavaCard APIs (and possibly some industry specific APIs such as Open Platform or GSM APIs). Executable JavaCard applets are obtained by compiling (with a standard Java compiler) and converting (with a JavaCard specific converter that tailors the files format to JavaCard specificities) JavaCard programs, and go through a bytecode verifier BCV which checks that applets are correctly formed and well-typed, and that they do not attempt to perform malicious operations during their execution, see [29]. In case bytecode verification is successful, applets are signed and loaded on-card, where they may be executed by the JavaCard Runtime Environment JCRE. The latter comprises the JavaCard Virtual Machine JCVM, that is very close to the Java Virtual Machine JVM.

In earlier work [8, 9], we formalized and verified the JavaCard platform (virtual machine and bytecode verifier), using the methodology described above. The formal development, which we call CertiCartes, contains formal specifications of the three JavaCard Virtual Machines JCVM (defensive, offensive, and abstract), and proofs that the defensive and offensive machines coincide on programs that pass bytecode verification. In order to validate our choices and implementation, we have conducted a similar experiment using the Jakarta environment. The results are positive: both the offensive and abstract virtual machines have been generated with minimal user interaction, and most of the proof obligations required for crossmachine validation are discharged automatically.

Organization of the paper ‡

‡ À faire à la fin

2 An overview of CertiCartes

CertiCartes is an in-depth feasibility study in proving the correctness of bytecode verification for JavaCard 2.1, using the proof assistant Coq [17]. CertiCartes contains executable specifications of the three JCVMs (defensive, offensive and abstract) 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 abstract JCVM. The next paragraphs briefly describe the contents of both modules; [7, 8, 9] provide further details.

2.1 Module JCVM

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. For simplicity, we only deal with closed programs hence (relevant parts of) 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 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 rstate :=
  Normal   : state → rstate |
  Abnormal : exception → rstate.
```

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

Virtual Machine	Defensive	Offensive	Abstract
Value	(Type,Number)	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). This tag is removed from the offensive machine, whereas the concrete values (numbers) are removed from the abstract 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. We thus obtain a type of defensive states `dstate` and return defensive states `rdstate`. Then, we model the defensive semantics of each JavaCard bytecode `b` as a function `dexec_b: dstate → rdstate`. Typically, the function `dexec_b` extracts values from the state, performs type verification on these values, and extends/updates the state with the results of executing the bytecode.

Finally, one-step defensive execution is modeled as a function `dexec: dstate → rdstate` which inspects the state to extracts the JavaCard bytecode `b` to be executed and then calls the corresponding function `dexec_b`.

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 `ostate` and return offensive states `rostate`. Then, one-step offensive execution `oexec: ostate → rostate` is modeled in the same way as one-step defensive execution, but all verifications and operations related to typing are removed.

Modeling the abstract JCVM. The abstract JCVM that is used for bytecode verification operates at the level of types. Formally, we define the type `astate` of abstract states as being the abstract frame. The memory model is further simplified because: (1) the abstract 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.

One-step abstract 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 abstract execution is modeled as a function `aexec: astate → (list rastate)`, where `astate` is the type of return abstract states.

Cross-validation of the JCVMs is a prerequisite for establishing the correctness of the BCV. It involves:

- showing that the offensive and defensive JCVMs coincide on programs that do not raise a type error when executed on the defensive JCVM;
- showing that every program that raises a type error when executed on the defensive JCVM also raises a typing error when executed with the abstract JCVM, i.e. that there is one error state in the list returned by `aexec`.

The proof of these two statements can be made by proving that the offensive and abstract JCVMs are sound abstractions of the defensive JCVM. Formally, we define two abstractions functions:

```
alpha_do : dstate → ostate
alpha_da : dstate → astate
```

together with their lifting to return states:

```
alpha_do_rs : rdstate → rostate
alpha_da_rs : rdstate → rastate
```

and prove that, under suitable constraints, the diagrams of Figures 2 and 3 commute (where the arrow on the right-hand side of Figure 3 means that the abstraction of the return defensive state is, up to sub-typing, an element of the list of return abstract states).

The commutation of both diagrams is proved by case analyses, first on the bytecode to be executed, then on the state of the defensive virtual machine and finally by equational reasoning. The correctness of the offensive machine is a direct consequence of the commutation of diagram 2 under the constraint that the defensive machine do not raise a type error.

$$
\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha\_do}"] & \texttt{rdstate} \arrow[d, "\texttt{alpha\_do\_rs}"] \\
\texttt{ostate} \arrow[r, "\texttt{oexec}"] & \texttt{rostate}
\end{tikzcd}
$$

Figure 2: Commutative diagram of defensive and offensive execution

$$
\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha\_da}"] & \texttt{rdstate} \arrow[d, hook, "\preceq\texttt{alpha\_da\_rs}"] \\
\texttt{astate} \arrow[r, "\texttt{aexec}"] & (\texttt{list rastate})
\end{tikzcd}
$$

Figure 3: Commutative diagram of defensive and abstract execution

The correctness of the abstract machine is a bit more tricky mainly because of the extra-procedural instructions, i.e. that change the current frame. Going into this details is beyond the scope of this paper, but this has the following consequence on the methodology: commutation of diagram 3 must be proved for intra-procedural bytecodes, and only when the defensive machine does not raise an exception.

Summary The module JCVM represents most of the effort behind CertiCartes. However, the construction of the offensive and abstract JCVMs relies on abstract interpretation and can be automated. Likewise, the cross-validation of the JCVM relies on case analyses and equational reasoning and can also be automated.

2.2 Module BCV

The module BCV constructs a certified bytecode verifier from an abstract virtual machine using a formalization of Kildall's algorithm that is similar to the one presented in [26, 33]. Our formalization of Kildall's algorithm takes as input:

- a type `astate` of abstract states and a well-founded relation `lt_astate` over `astate`;
- an abstract virtual machine `aexec: astate` → `(list astate)` which is monotone w.r.t. `lt_astate`;

and produces as output:

- an executable function `bcv: jcprogram` → `bool`;
- a proof that a program `p` such that `bcv p = true` does not raise any typing error when executed on the abstract virtual machine `aexec`.

Additionally, the module BCV contains a justification of method-by-method verification, i.e. a proof that a program is well-typed if all its methods are. Together with the proofs of the commuting diagrams, these results ensure that bytecode verification is correct.

Summary The module BCV is largely independent from the specifics of the virtual machine under consideration, and hence could be applied to other settings with minimal effort.

2.3 Assessment

Our experiment with CertiCartes corroborates the common views that machine checked verification of standard programming languages is becoming viable, and validates an original methodology which is well-suited for reasoning about typed low-level languages, and which can be applied to novel type systems for Java(Card) [4, 21, 27, 38]. Nevertheless, the completion of CertiCartes has proved labour-intensive and tedious, and has sorely highlighted the lack of tool support for specifying and proving in the large. On the positive side, our work has also suggested that it is possible to automate large parts of the specification and proof work. This is the goal of the Jakarta toolset, which we further describe below.

See [6] for details.

3 The Jakarta Specification Language JSL

JSL is a small language for describing virtual machines in a neutral mathematical style. More precisely: JSL types are first-order polymorphic types, JSL expressions are standard first-order algebraic terms, JSL functions are defined by conditional rewrite rules ([3, 12]).

A JSL theory is given by (1) a set of declarations that introduce first-order polymorphic datatypes and their constructors, (2) by a set of function definitions introducing function symbols and their computational meaning via conditional rewrite rules of the form:

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

A specificity of JSL is to require that the right-hand sides of the conditions, i.e the $\mathtt{r}_i$ s, to be patterns with fresh variables (a pattern is a linear term built from variables and constructors). Such a format of rewriting seems unusual but is closely related to pattern-matching in functional programming and proof assistants. Figure 4 and 5 give a flavor of JSL and the corresponding ML definitions. This restricted framework makes the translation of specifications into proof assistants and programming languages straightforward. It is also well adapted to the particular purpose of abstraction:

- The presentation of functions by a list of rules makes the notion of *branch* (i.e. path) of a pattern matching more immediate than the usual tree-like pattern matching. Indeed each rule of a function represents a possible valuation of the parameters of the function. Our abstraction mechanism applies a different transformation for each possible value of each parameter of a function, therefore the presentation with rules is well adapted for its description.
- The presentation with conditional rewrite rules is more flexible than the usual pattern matching. In particular it allows to define partial and/or non deterministic functions by giving several rules with non exhaustive or overlapping conditions. Mechanisms to make such functions total and deterministic are part of the JTK, see section 4.2.

The connection with functional languages is enforced by some syntactical constraints on the conditional rewrite rules described in the next section. Figure 4 gives the JSL specification of the `ifnull` bytecode, which corresponds to the ML function described in figure 5. `ifnull x` is a branching bytecode. If the top value in the stack is a null reference, then the program counter jumps to `x`, otherwise it is incremented normally. The top value of the stack must be a reference. Rules 1, 2 and 5 detect different kinds of errors, and rules 3 and 4 apply branching for positive or negative test (performed by the auxiliary function `res_null` given in section 6.1). The top value is removed from the stack (`update_opstack(tail(opstack...))`).

3.1 JSL syntax and execution model

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

```
data valu_prim = VReturnAddress nat | VBoolean z | VByte z| VShort z | VInt z.
data valu_ref = VRef_null | VRef_array type0 heap_idx | VRef_instance class_idx heap_idx
              | VRef_interface interf_idx heap_idx.
data valu = VPrim valu_prim | VRef valu_ref.
data list 'a = Nil | Cons 'a (list 'a).
data exc 'a = Value 'a | Error.

function  ifnull  : bytecode_idx → dstate → rdstate :=
<ifnull_1>
 (stack_f s) → Nil
    ⇒ (ifnull b s) → (abortCode State_error);

<ifnull_2>
 (stack_f s) → (Cons h lf),
 (head (opstack h)) → (Value v),
 v → (VPrim v0)
    ⇒ (ifnull b s) → (abortCode Type_error);

<ifnull_3>
 (stack_f s) → (Cons h lf),
 (head (opstack h)) → (Value (VRef vr)),
 (res_null vr) → True
    ⇒ (ifnull b s) → (update_frame (update_pc b (update_opstack (tail (opstack h)) h)));

<ifnull_4>
 (stack_f s) → (Cons h lf),
 (head (opstack h)) → (Value (VRef vr)),
 (res_null vr) → False
    ⇒ (ifnull b s) → (update_frame (update_pc (S (p_count h)) (update_opstack (tail (opstack h)) h)));

<ifnull_5>
 (stack_f s) → (Cons h lf),
 (head (opstack h)) → Error
    ⇒ (ifnull b s) → (abortCode Opstack_error).
```

Figure 4: Example of JSL specification: ifnull

```
type valu_prim = VReturnAddress of nat | VBoolean of z |
                   VByte of z| VShort of z | VInt of z
type valu_ref = VRef_null | VRef_array of type0*heap_idx |
                 VRef_instance of class_idx*heap_idx |
                 VRef_interface of interf_idx*heap_idx
type valu = VPrim of valu_prim | VRef of valu_ref
type 'a list = Nil | Cons of 'a list
type 'a exc = Value of 'a | Error

let ifnull (b:bytecode_idx) (s:dstate): rdstate =
 (match stack_f s with
   | Nil → abortCode State_error
   | Cons(h,lf) →
   (match head (opstack h) with
     | Value v →
     (match v with
       | VPrim v0 → abortCode Type_error
       | VRef vr →
         (match res_null vr with
          | true  → update_frame (update_pc b (update_opstack (tail (opstack h)) h))
          | false → update_frame (update_pc (S (p_count h)) (update_opstack (tail (opstack h)) h))))
     | Error → abortCode Opstack_error))
```

Figure 5: The ML version of ifnull

Definition 3.1.1

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

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

  where in $h\vec{e}$*, it is assumed that* $\vec{e}$ *is of length* $\mathsf{ar}(h)$.

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

- **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 $c\ \vec{p}$*, variables are all different (linearity).*

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

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

  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;*
  - $\mathsf{var}(l_k) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup\ \ldots\ \cup \mathsf{var}(r_{k-1})$ *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* $j \neq k$*;*

These constraints make the connection to pattern matching quite straightforward, for example the following rules:

$$\begin{array}{lcl} l \twoheadrightarrow Nil & \Rightarrow & (is\_empty\ l) \rightarrow True \\ l \twoheadrightarrow (Cons\ \ el\ tl) & \Rightarrow & (is\_empty\ l) \rightarrow False \end{array}$$

correspond to the two branches of the following ML function:

```
let is_empty l = match l with [] → True | el::tl → False
```

Conditional rewriting is defined as follows:

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* $r \in R$

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

a position p *in* s *and a substitution* θ *with domain* $\mathsf{dom}(R)$ *such that:*

- $s_{|_p} = \theta g$ *and* $t = s[p \leftarrow \theta d]$*;*
- *for* $1 \leq i \leq n$, $\theta l_i \rightarrow_R^* \theta r_i$*;*

where $\rightarrow_R^*$ *is the reflexive-transitive closure of* $\rightarrow_R$.

Notice that 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 our framework.

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

$$(\text{Tconv})\frac{\Gamma\vdash t:\sigma' \quad \sigma=_\delta\sigma'}{\Gamma\vdash t:\sigma} \qquad (\text{Texpr})\frac{\Gamma\vdash f:\sigma_1\to\ \cdots\ \to\sigma_n\to\sigma \quad \Gamma\vdash t_1:\sigma_1\ \dots\ \Gamma\vdash t_n:\sigma_n}{\Gamma\vdash f\ t_1\dots t_n:\sigma}$$

Figure 6: Typing rules

3.2 JSL types

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

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 $D\vec{T}$, it is assumed that $\vec{T}$ is of length $\mathsf{ar}(D)$.

Constructor/function symbols are assigned (first-order) type schemes, i.e. closed expressions of the form

$$\forall\alpha_1\ \dots\alpha_m.\ \sigma_1\to\quad\to\sigma_n\to\tau$$

where $\alpha_1\ \dots\alpha_m\in\mathcal{V}_\mathcal{T}$ and $\sigma_1\ \dots\sigma_n\in\mathcal{T}$. Formally, we assume given for each constructor/function symbol h a declaration of the form $h:\rho$ where $\rho\in\mathcal{TS}$.

As usual, expressions are type checked relative to a context in which variables are 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 $x_1:\sigma_1,\ \dots\ ,x_k:\sigma_k$ where $x_1\ \dots\ x_k$ are pairwise disjoint variables and $\sigma_1\ \dots\ \sigma_k$ 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 typed checked in the usual way, i.e. a rule $l_1\twoheadrightarrow r_1,\ \dots\ ,l_n\twoheadrightarrow r_n\Rightarrow g\to d$ 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 $1\le i\le n$. Note that the context Δ may be computed from the type declaration of function/constructor symbols.

Finally a function is well-typed if all its rules are well typed.

4 The Jakarta Transformation Kit

The Jakarta Transformation Kit is a tool to transform JSL specifications. Currently the JTK only supports *abstraction*, which is used in the construction and certification of the BCV.

4.1 Principles

The abstraction $\hat{\mathtt{S}}$ of a JSL specification $\mathtt{S}$ is constructed in three successive steps:

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

2. the user builds an abstraction script that guides the abstraction process in places where automatic procedures are inaccurate; the purpose and effect of this script is detailed below.

3. based on the inputs of steps 1 and 2 above, the JTK constructs automatically for every function $\mathtt{f}\colon \mathtt{T}_1 \rightarrow \ldots \rightarrow \mathtt{T}_n \rightarrow \mathtt{U}$ of $\mathtt{S}$ its abstract counterpart $\hat{\mathtt{f}}\colon \hat{\mathtt{T}}_1 \rightarrow \ldots \rightarrow \hat{\mathtt{T}}_n \rightarrow \hat{\mathtt{U}}$. Unless indicated otherwise by the abstraction script, the function $\hat{\mathtt{f}}$ is constructed by a syntactic translation on the rewrite rules defining `f`, followed by a cleaning phase where vacuous conditions and rules are removed.

Most of the abstraction process is handled automatically; however there are a number of situations where user interaction is required. It is the purpose of the transformation script to facilitate such interactions through a carefully designed set of commands.

4.2 Commands of the abstraction script

Each command is of the general form:

$$<COMMAND> ::= <command> <target> list \ [\mathbf{by} <expr>] \ [\mathbf{in} <scope> list]$$

- The optional field *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 field *target* specifies which syntactic compounds trigger the command.
- The optional field *expr* is used in substitution commands to specify the syntactic compounds by which targets must be replaced.

The form of the scope and target fields uses a notion of *position* inside a specification. It is the usual notion of position inside a term extended to the whole 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 ($l_j \rightarrow r_j$ or $g \rightarrow d$ 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:

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

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

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

where $< f > . < pos >$ denotes a position in the function f as described above, and `f@i` denotes the `i`th argument of `f`. Commands fall into three categories: discarding commands, substitution commands and post-processing commands.

Discarding commands (figure 7) specify which patterns in expressions are unwanted and must be deleted. For example, to generate a P-offensive JCVM, discarding commands are used to remove all verifications related to P. 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.*
`delete` *ltarget*	*Removes all (occurrences of) elements described by ltarget.*

Figure 7: Discarding commands

abstraction commands	Effect
`coercion` *lfname1*	*Functions of lfname1 are declared as coercions in case of type errors during by the translation phase.*
`select` *largloc*	`select f@i` *replaces* `(f x`$_1$`...x`$_i$`...x`$_n$`)` *by* `x`$_i$
`replace` *ltarget* `by` *target*	*Replaces elements of ltarget by target.*

Figure 8: Substitution 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 lookup by static lookup in the abstract semantics of method invokation.

Post-processing commands (figure 9) deal with functions which may have become non-deterministic or non-total during abstraction. Post-processing commands are used for example to collect all possible return states into lists during the construction of the abstract JCVM.

abstraction commands	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 (using the lift monad). The transformation is propagated consequently.*

Figure 9: Post-processing commands

4.3 Synthesis of the offensive JCVM

Figure 10 gives excerpts of the abstraction functions used to generate the abstract JCVM from the defensive JCVM. Notice for example how type information contained in the type `valu_prim` of defensive values are removed from `ovalu_prim`.

```
function abstract_valu_prim : valu_prim → ovalu_prim :=
⇒abstract_valu_prim (VReturnAddress v) → (nat2z v) ;
⇒abstract_valu_prim (VBoolean v) → v ;
⇒abstract_valu_prim (VByte v) → v ;
⇒abstract_valu_prim (VShort v) → v ;
⇒abstract_valu_prim (VInt v) → v .

function alpha_do_stack : stack → ostack :=
⇒ alpha_do_stack  Nil → Nil;
⇒ alpha_do_stack (Cons x y) → Cons (alpha_do_frame x) (alpha_do_stack y).

function alpha_do :dstate → ostate :=
⇒ alpha_do (Build_state sh hp s)
     → Build_ostate (alpha_do_sheap sh) (alpha_do_heap hp) (alpha_do_stack s).
```

Figure 10: Abstraction functions (excerpts)

Figure 11 contains the script used to generate from the defensive JCVM (which is over 10000 lines long) the offensive JCVM (5000 lines). The script begins with a preamble, which specifies the function to abstract, the name of the resulting function (at line 3) and the abstraction functions (at line 4). Then the remaining of the script is divided in two blocks of discarding commands (between lines 8 and 17) and substitution commands (between lines 21 and 25); there are no post-processing commands for this abstraction.

The block between lines 8 and 17 uses the discarding commands **`delete`** and **`drop`**.

```
 1 (* Preamble *)
 2
 3 build oexec from dexec using
 4 abstraction functions alpha_do_prim alpha_do ...
 5
 6 (* Discarding commands *)
 7
 8 delete Type_error type_verification signature_verification
 9        types_match test_exception_aastore assignment_compatible
10
11 delete test_exception_putfield_ref.2 test_exception_astore.3
12        test_exception_checkcast.4 test_exception_checkcast.5
13        test_exception_putstatic_ref.2
14
15 delete t0 in putstatic putfield_obj
16
17 drop getfield_obj@3 getstatic@1 tableswitch@1 lookupswitch@1
18
19 (* Substitution  commands *)
20
21 replace get_inst_from_heap by oget_inst_from_heap
22 replace v2t by (Ref Ref_null)
23
24 select vp2z@1 v2z@1 vr2z@1 vr2hi@1 extr_from_opstack@2
25 select tpz2vp@2 in adr_aload res_opstack getfield_obj putfield_obj ...
```

Figure 11: OFFENSIVE ABSTRACTION SCRIPT

- The command **delete** is used to suppress all occurrences of the target within the specified scope. By propagation, these commands may lead to rules being deleted, in particular because rules whose conclusion has been removed are also removed. For example at line 8, the script forces rules returning a type error (i.e. `type_error`...) to be removed everywhere in the specification.
- The command **drop** is used to remove some arguments from the signature of a function. Such a command is needed to clean up code that is made vacuous by the abstraction. For example, the function `getfield_obj` takes as third argument some typing information which becomes vacuous in the offensive JCVM. Therefore we instruct the argument to be removed at line 17.

The block between lines 21 and 25 uses the substitution commands **replace** and **select**.

- **replace** allows to replace the target by the argument specified in the **by** field. This argument can be a function name or an expression. This command allows the user to override the expression computed by the standard abstraction mechanism. We use this command at line 21 to provide a definition of the function `oget_inst_from_heap`, which takes as argument a heap `h` and a value `v`, and extracts from `h` the instance referenced by `v`. In this case, the purpose of the substitution is to provide a direct definition of the abstracted function; it is possible, but less convenient in this case, to use discarding commands for achieving the same definition.
- **select** allows to optimize the specification by projecting a function call onto one of its arguments. Such an optimization is useful when the abstraction produces a function `f` that behaves as a projection function, i.e. such that $\mathtt{f}(x_1, \ldots, x_n) = x_i$ for all $x_1 \ldots x_n$. An example of such function is given by `vp2z`. In the defensive JCVM, this function maps a typed value to its corresponding integer. In the offensive JCVM, the function behaves as the identity, so `(vp2z n)` is replaced by `n` as instructed at line 24. Note that in this case the command affects the computational behavior of the functions, but not their meaning.

4.4 Synthesis of the abstract JCVM

Figure 12 contains some of the abstraction functions used to generate the abstract JCVM from the defensive JCVM. Observe how specifics of the abstract JCVM are captured by the abstraction functions. For example:

- the abstract JCVM must keep track of return addresses to handle subroutines, and forget all other values: the definition of `alpha_da_prim` reflects this desiderata;

```
function alpha_da_prim : valu_prim → avalu_prim :=
⇒ alpha_da_prim (VReturnAddress v) → aReturnAddress v ;
⇒ alpha_da_prim (VBoolean v) → aBoolean ;
⇒ alpha_da_prim (VByte v) → aByte ;
⇒ alpha_da_prim (VShort v) → aShort ;
⇒ alpha_da_prim (VInt v) → aInt .

function alpha_da : dstate → astate :=
⇒ alpha_da (Build_state sh hp s)
    → Build_astate (alpha_da_sheap sh) (alpha_da_stack s).
```

Figure 12: Abstraction functions (excerpts)

- abstract values do not carry any component that corresponds to the heap: this is reflected in the definition of `alpha_da`.

Figure 13 contains (excerpts of) the script used to generate the abstract JCVM (which is around 5000 lines long); the whole script is approximately 120 lines long. The script is longer than the one for the offensive abstraction, which is not surprising since the typed abstraction is more complex: generating the abstract JCVM implies deep modifications in the data structures of the defensive JCVM. For example, the heap is removed, method calls do not create new frames, etc.

The preamble is immediately followed by a set of commands which act on the whole specification. These commands are used to discard operations and verifications involving the heap or values, for example `res_null`, `throwException` and `abortMemory` at line 10. Such commands are sufficient for handling various significant bytecodes, including `load`, `store`, `new`, `astore`, `goto`, `jsr` and `ret`.

```
1   (* Preamble *)
2
3   build aexec from dexec using
4   abstraction functions alpha_da_prim alpha_da ...
5
6   (* General commands *)
7
8   ...
9
10  delete throwException res_null  abortMemory
11
12  ...
13
14  (****** Commands for branching bytecodes ********)
15
16  ...
17
18  determine ifnull ifnonnull
19  drop res_scompare@1,2,3  res_acompare@1,2,3
20  determine res_scompare res_acompare
21  ...
```

Figure 13: Typed abstraction script (excerpts)

The remaining of the script takes care of all bytecodes whose abstract semantics must be fine-tuned. This includes all bytecodes for method invokation and memory management, which are treated by discarding and substitution commands. We concentrate on branching bytecodes, as they are responsible for non-determinism and require the use of post-processing commands.

- The bytecodes `ifnull` and `ifnonnull` pop the top value of the operand stack and check whether it is null to determine the next program counter. The abstraction renders these functions non-deterministic. We handle this non-determinism with the command **`determine`** at line 18, whose effect is to transform the bytecodes into deterministic functions of target type `list astate`. The command, which implements the usual strategy of collecting the return values of a non-deterministic function into a list, also propagates the transformation to the function `aexec` and consequently to all bytecodes.
- The bytecodes `if_scmp_cond`, `if_acmp_cond` and `if_cond` pop two values from the operand stack

and compare them in order to determine the next program counter. This comparison is performed by the auxiliary functions `res_scompare` and `res_acompare`, but the abstraction renders these functions non-deterministic. A similar solution using **`determine`** applies, see line 20, but for technical reasons not discussed here, some arguments of these functions must be dropped before calling **`determine`**; the notation **`drop`** `f@1,2,3` at line 19 is a shorthand for **`drop`** `f@1 f@2 f@3`.

5 Cross-validation of virtual machines

As emphasized in the introduction, the constrained format of JSL facilitates (1) translation to proof systems and programming languages (JPI) (2) reasoning about JSL specifications and developing automatic proof procedures for this purpose (JAK). Both aspect are detailed in what follows.

5.1 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, which represent (recursive) definitions using pattern matching and case expressions instead of rewrite rules. In order to target several such proof assistants, we first translate JSL specifications into an intermediate language JIR (Jakarta Intermediate Representation language), supporting such recursive definitions. Note that the translation may fail if the rewrite rules do not have the correct shape. Further, case expressions that are produced from well-typed JSL specifications may still be considered as ill-formed by the target system, in particular because proof assistants usually reject partial, non-deterministic or non-terminating functions. However, such a limitation, which 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, has not turned out to be a problem in practice.

Currently, Jakarta provides a mean to translate JSL specifications to 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 [30] for an efficient execution of JSL specifications.

5.2 The Jakarta Automation Kit

In order to cross-valide the virtual machines, we have have been developing a repository of tools that: i) generate automatically auxiliary lemmas required to prove that diagrams commute, as shown in Figures **??** and **??**; ii) generate proof scripts for these tactics, using efficient proof tactics to reason about recursive functions. These two points are described below.

5.2.1 Automatic generation of commutation lemmas

In order to establish the main commutation lemmas, we have developed a procedure that generates an auxiliary lemma for each abstracted function upon which the function `dexec` depends. Such abstracted functions include in particular the functions that specify the semantics of each bytecode.

Auxiliary lemmas Let f be a function, $\hat{f}$ its abstraction version and α the abstraction function, the corresponding lemma has the following generic form:

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

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 is instantiated to $\in_{\leq}$ (membership up to subtyping) in the case of the typed abstraction;

- the predicate $Prec_{\hat{f}}$ restricts the commutation of f and $\hat{f}$ to the desired cases. In fact, $Prec_{\hat{f}}$ is a conjunction of *premises*, each of which corresponds to a command in the abstraction script that modifies the generic abstraction process for f.

The following examples illustrate typical situations where premises must be added to lemmas:

1. (*deleted conclusion*) if a rule r is of the form:

$$l_1 \to r_1 \ldots l_n \to r_n \Rightarrow (f\ \vec{x_i}) \to (...t...)$$

   and a command **`delete`** t is applied to remove r, then the following premise must be added to $Prec_{\hat{f}}$:

$$\neg(f\ \vec{x_i}) = (...t...)$$

   This situation occurs in the case of the offensive abstraction, since we delete `TypeError`, and in the typed abstraction, since we delete `ThrowException`. In such cases, the appropriate preconditions, respectively $(f\ x_i) \neq$ (`Abnormal TypeError`) and $(f\ x_i) \neq$ (`ThrowException` e), are thus automatically generated when necessary.

2. (*deleted preconditions*) if a rule r is of the form:

$$l_1 \to r_1 \ldots l_n \to r_n \Rightarrow g \to d$$

   and it is removed during the abstraction, then the following premise is added to $Prec_{\hat{f}}$:

$$\neg(l_1 = r_1 \wedge \ldots \wedge l_n = r_n)$$

   Note that such a rule encompasses the previous one, in the sense . However, we have found convenient to use the TBC

3. (*propagation of the premises*) Finally, 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 h appearing in a rule r (not removed by the abstraction) of the form:

$$l_1 \to r_1, \ldots, l_i \to r_i,\ (...(h\ \vec{x_j})...) \to r_{i+1},\ \ldots \Rightarrow g \to d$$

   the following premise is added to $Prec_{\hat{f}}$:

$$(l_1 = r_1 \wedge \ldots \wedge l_i = r_i) \to (Prec_{\hat{h}}\ \vec{x_j})$$

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 JCVM as a consequence of the lemma; if it is not restrictive enough, we may be unable to prove the lemma itself.

Some example of cases where it is necessary to add a premise are the following:

- In the offensive abstraction, if f returns a `rstate` then $Prec_{\hat{f}}$ should generally contain a condition of the form: $(f\ x_i) \neq$ (`Abnormal TypeError`), since we have to prove that defensive and offensive bytecodes commute when no type error is returned by the defensive version.
- In the typed abstraction, $Prec_{\hat{f}}$ should generally contain a condition of the form: $(f\ x_i) \neq$ (`ThrowException` e), since we have to prove that the defensive and abstract bytecodes commute when no exception is raised by the defensive version (section 2.1)).

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 $Prec_{\hat{f}}$. Our methodology, in which the abstraction process is automated and systematic, allows to automate the generation of lemmas as well:

each operation of the abstraction engine (guided by the abstraction script) may generate a new premise to be added to a lemma.

We have designed a mechanism, independent from the theorem prover used, plugged into the abstraction engine, that generates lemmas corresponding to the operations performed during the abstraction. More precisely we have extended the abstraction engine of Jakarta in order to generate, for each function f of the specification abstracted into $\hat{f}$, the statement of a commutation lemma for f and $\hat{f}$ with the correct preconditions. We enumerate in the following the different situations where the abstraction engine adds a premise to a lemma (See section 6 for examples):

We will see in the example of section 6 that in some cases of deleted conclusions or preconditions, it is wrong to add new premise to $Prec_{\hat{f}}$, i.e. it would make $Prec_{\hat{f}}$ too strong for the purpose of proving the correctness of BCV. Therefore for each command one must decide wether it must generate new premises or not. To this purpose, each abstraction commands described in section 4.2 has a *variant* that do not add a premise. For example the command **`delete`** `t` **`in`** `f` has a variant **`delete_nopred`** `t` **`in`** `f` which has the same effect on the abstraction (delete rules containing `t`) but do not modify $Prec_{\hat{f}}$.

The main theorem The main (auxiliary) lemma generated by the method described above has the following forms for offensive and typed abstractions:

$$\forall x_i.(Prec_{\texttt{oexec}}\ \vec{x_i}) \Rightarrow \texttt{alpha\_do}\ (\texttt{dexec}\ x_i) = (\texttt{oexec}\ \overrightarrow{(\texttt{alpha\_do}\ x_i)})$$

$$\forall x_i.(Prec_{\texttt{aexec}}\ \vec{x_i}) \Rightarrow \texttt{alpha\_da}\ (\texttt{dexec}\ x_i) \in (\texttt{aexec}\ \overrightarrow{(\texttt{alpha\_da}\ x_i)})$$

For the proof to be complete, the main lemma must imply the main commutation theorem. Proving this last property actually reduces to proving that $Prec_{\texttt{oexec}}$ and $Prec_{\texttt{aexec}}$ are not too strong. We illustrate this for both theorems:

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

  $$\forall x_i.(\texttt{dexec}\ x_i) \neq \texttt{TypeError} \Rightarrow \texttt{alpha\_do}\ (\texttt{dexec}\ x_i) = (\texttt{oexec}\ \overrightarrow{(\texttt{alpha\_do}\ x_i)})$$

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

  $$\forall x_i.(\texttt{dexec}\ x_i) \neq \texttt{TypeError} \Rightarrow (Prec_{\texttt{oexec}}\ \vec{x_i})$$

- In the typed abstraction, the main theorem for intra-procedural bytecodes is the following:

  $$\forall x_i.\forall e.(\texttt{dexec}\ x_i) \neq (\texttt{RaiseException}\ e) \Rightarrow \texttt{alpha\_da}\ (\texttt{dexec}\ x_i) \in (\texttt{oexec}\ \overrightarrow{(\texttt{alpha\_da}\ x_i)})$$

  So the following lemma must be proved for intra-procedural bytecodes:

  $$\forall x_i.\forall e.(\texttt{dexec}\ x_i) \neq (\texttt{RaiseException}\ e) \Rightarrow (Prec_{\texttt{aexec}}\ \vec{x_i})$$

  Lemmas for the five extra-procedural bytecodes (method invocation and return) are much more complex. They are stated using special functions written by hand. Showing there precise statements is beyond the scope of this paper and cannot be automated by Jakarta. Notice that there statements and proofs can however be added by hand in the abstraction script.

5.2.2 Reasoning tactics

In order to automate the proofs of these lemmas , we have designed specific reasoning tactics that generate induction principles for reasoning about complex recursive functions. This yields drastic improvements; in particular, we have used these tactics in CertiCartes and proof scripts became significantly easier to construct, and their size was drastically reduced. See [5] for more details. These tactics are specific to Coq but their underlying principles, that we describe below, are applicable to other proof assistants.

Notations We will use Coq notations in the following. The universal quantification $\forall x : t.u$ and the λ-abstraction λ`x:T.t` are denoted respectively by `(x:t)u` and `[x:T]t`. Recursive functions are introduce by the construction **`Fixpoint`** `f` (or **`Fix`** `{f...}`) and finally pattern matching on inductive types by **`Cases...of...`**.

Induction principles are one of the powerful tools provided by proof assistants like Coq. It allows to reason by cases and recurrence on the elements of an inductive type. We see here the definition in Coq of the type `nat` and the type of its associated principle:

```
Inductive nat  : Set := O : nat | S : nat → nat.
nat_ind: (P: nat → Prop) (P O) → ((n:nat)(P n) → (P (S n))) →  (n:nat)(P n).
```

Generally These principles are generated automatically for *relational specifications* but not for *functional specifications*. This makes the former a popular solution even if *functional specifications* have other advantages, like the possibility of testing trough execution, code extraction etc.

Elimination following the shape of a function It is however relatively natural to build an induction principle that follows the shape of a particular function. It is particularly useful for *reasoning by case/induction on the possible branches* of the definition of the function. For example the following function yields an induction principle with four branches.

```
Fixpoint greatest_g [n:nat] : (exc nat) :=
Cases n of
 | O ⇒ (Error ?)
 | (S m) ⇒
  Cases (g n) of
   | true ⇒ (Value ? n)
   | false ⇒ Cases (h n) of
               | true  ⇒ (Value ? n)
               | false ⇒ (greatest_g m)
              end
  end
end.
```

Considering the different paths of a function is a very common way to reason about it, and it is heavily used in CertiCartes (see Section 6 for more examples). We have designed a Coq package to automate this inductive/case analysis process. According to the Coq paradigm, our package builds *proof terms* in the Curry-Howard philosophy. See [5] for a detailed description of the algorithm used to generate such terms by modifying the term of the function. The principle we obtain in the example is the following:

```
greatest_g_ind: (Q:(nat → Prop))
  ((n:nat) (Q O))
 → ((m:nat) (g (S m)) = true → (Q (S m)))
 → ((m:nat) (h (S m)) = true → (g (S m))=false → (Q (S m)))
 → ((m:nat) (Q m) → (h (S m)) = false → (g (S m)) = false → (Q (S m)))
 →  (n:nat) (Q n)
```

Notice how the equalities `(g _)=...` and `(h _)=...` capture the *environment* induced by the branches of the corresponding **`Cases`** expressions, and can be used in proofs. The last branch contains an induction hypothesis `(Q g m)`, which corresponds to the recursive call in the last branch of `greatest_lt`.

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:

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

  where the PO_i's are the proof obligations corresponding to each branches of f, and $\vec{x_i}$'s correspond to the arguments of f. 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* **Analyze** which applies directly to a particular goal. This tactic allows for more automation because we can replace automatically occurrences of case arguments in subgoals by the pattern corresponding to each branch.

The tactics described here are combined with standard tactics of Coq in order to build automatically a default proof script for each lemma generated by the method described in section 5.2.1. The script follows the scheme described in section 2.1: first a case analysis is made using the tactic described above, then equational reasoning is used. Currently 75 percent of these lemmas are proved automatically by the default proof script. The remaining lemmas are more difficult to discharge automatically because they deal with functions for which the commutation is difficult.

6 A complete example: the bytecode `ifnull`

In this section, we illustrate the benefits of our package in establishing correctness results for the `ifnull` JavaCard bytecode, already described in section 3. It is a simple bytecode with only five rules. We will see that it needs however some abstraction commands to drive the abstraction itself, but also to 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) define the abstraction functions, create a first script with no particular command (except the preamble);
2. apply the JTK;
3. (section 6.3) 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;
4. Compile (proof check) the generated Coq files: defensive machine, offensive machine and default proofs of commutation lemmas (one file per function);
5. (section 6.4) 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;
   - (section 6.5) if the lemma cannot be proved because of wrong premises , then add new commands in the script to correct them, reapply 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 5 are kept when regenerating the proofs, so that proofs that are still correct do not need to be done again.

6.1 Defensive machine

The defensive specification of `ifnull` has already been given in figure 4 of section 3. It is translated in Coq as follows:

```
Definition ifnull [b:bytecode_idx] [s:jcvm_state] :=
  Cases (stack_f s) of
    Nil ⇒ (abortCode State_error)
  | (Cons h lf) ⇒
      Cases (head ? (opstack h)) of
        (Value v) ⇒
```

```
          Cases v of
            (VPrim v0) ⇒ (abortCode Type_error)
          | (VRef vr) ⇒
              Cases (res_null vr) of
                  True ⇒ (update_frame (update_pc b (update_opstack (tail ? (opstack h)) h)) s)
                | False ⇒ (update_frame (update_pc (S (p_count h)) (update_opstack (tail ? (opstack h)) h)) s)
              end
          end
      | Error ⇒ (abortCode Opstack_error)
      end
  end.
```

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  : valu_ref → bool :=
 <res_null1> vr → VRef_null ⇒ (res_null vr) → True;
 <res_null2> vr → (VRef_array t hi) ⇒ (res_null vr) → (nat_is_zero hi);
 <res_null3> vr → (VRef_instance ci hi) ⇒ (res_null vr) → (nat_is_zero hi);
 <res_null4> vr → (VRef_interface ii hi) ⇒ (res_null vr) → (nat_is_zero hi).
```

6.2 Abstraction functions and first script

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

```
type abs_valu_prim  = Z.
type abs_valu  =  Z.
type abs_valu_ref  = Z.

function abstract_heap_idx : heap_idx → abs_heap_idx :=
 ⇒ abstract_heap_idx x → (inject_nat x).

function abstract_bytecode_idx : bytecode_idx → abs_bytecode_idx :=
 ⇒ abstract_bytecode_idx x → (inject_nat x).

function abstract_valu_prim : valu_prim → abs_valu_prim :=
 ⇒ abstract_valu_prim (VReturnAddress v) → (inject_nat v);
 ⇒ abstract_valu_prim (VBoolean v) → v;
 ⇒ abstract_valu_prim (VByte v) → v;
 ⇒ abstract_valu_prim (VShort v) → v;
 ⇒ abstract_valu_prim (VInt v) → v.

function abstract_valu_ref :  valu_ref → abs_valu_ref :=
 ⇒ abstract_valu_ref VRef_null → ZERO;
 ⇒ abstract_valu_ref (VRef_array t hp) → (abstract_heap_idx hp);
 ⇒ abstract_valu_ref (VRef_instance c hp)→ (abstract_heap_idx hp);
 ⇒ abstract_valu_ref (VRef_interface i hp) → (abstract_heap_idx hp).

function abstract_valu : valu → abs_valu  :=
 ⇒ abstract_valu (VPrim x) → (abstract_valu_prim x);
 ⇒ abstract_valu (VRef r) → (abstract_valu_ref r).

...

function  abstract_heap : heap → abs_heap :=
 ⇒ abstract_heap  Nil → Nil;
 ⇒ abstract_heap (Cons x y) → Cons (abstract_obj x) (abstract_heap y).


function abstract_frame : frame → abs_frame :=
 ⇒ abstract_frame (Build_frame ops lv cmi pk b bi)
    → Build_abs_frame (abstract_sheap ops) (abstract_sheap lv) cmi pk b (inject_nat bi) .

function abstract_jcvm_state : jcvm_state → abs_jcvm_state  :=
 ⇒ abstract_jcvm_state (Build_jcvm_state sh hp s)
    → Build_abs_jcvm_state (abstract_sheap sh) (abstract_heap hp) (abstract_stack s).

function abstract_returned_state : returned_state →abs_returned_state :=
 ⇒ abstract_returned_state (Normal js) → (Abs_Normal (abstract_jcvm_state js));
 ⇒ abstract_returned_state (Abnormal x) → (Abs_Abnormal x).
```

Notice that:

- Type information is removed, for example the type `valu_prim` is abstracted into `Z`.

- type_prim and type0 have no offensive version, since type information is removed.
- Since all values are abstracted into the unique type Z, references must be abstracted into Z, especially VRef_null which is abstracted into ZERO.
- the type synonyms heap_idx and bytecode_idx are used instead of nat in the functions abstract_heap_idx and abstract_bytecode_idx. It allows to prevent nat values that are not indexes to be abstracted by these functions. This is the main use of type synonyms: allow different abstraction for different use of the same concrete type.

These datatypes and functions are also translated into Coq types, for example, the Coq version of abstract_valu_ref is the following:

```
Definition abstract_valu_ref [x0:valu_ref] :=
  Cases x0 of
    VRef_null ⇒ ZERO
  | (VRef_array t hp) ⇒ (abstract_heap_idx hp)
  | (VRef_instance c hp) ⇒ (abstract_heap_idx hp)
  | (VRef_interface i hp) ⇒ (abstract_heap_idx hp)
  end.
```

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 no other command.

```
build oifnull from ifnull using
abstraction functions abstract_valu_prim abstract_valu
 abstract_valu_ref abstract_frame abstract_stack abstract_jcvm_state
 abstract_sheap abstract_returned_state abstract_heap_idx
 abstract_bytecode_idx abstract_o_data abstract_type_instance
 abstract_obj abstract_heap
```

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) (ifnull and auxiliary functions), and returns two warnings:

```
WARNING!  ores_null is not deterministic because of the following rules couples:
 ores_null_1 and ores_null_2 with substitution  [ (hi , ZERO) ]

WARNING!  oifnull is not deterministic because of the following rules couples:
 oifnull_2 and oifnull_4 with substitution  [EMPTY]
 oifnull_2 and oifnull_3 with substitution  [EMPTY]
```

Overlapping rules have been detected by a unification algorithm in two generated functions ores_null and oifnull. In order to make the abstraction correct, we must tell Jakarta how to make this two functions deterministic. We will thus modify the initial script.

6.3 Second abstraction script

The offensive version ores_null of res_null generated by the JSL from the initial script, is the following (The function z2n is a coercion from Z to nat; such coercions are given in the abstraction script, but we omitted them for clarity.):

```
function  ores_null  : abs_valu_ref → bool :=
<ores_null_1> vr → ZERO ⇒ (ores_null vr) → True;
<ores_null_2> vr → hi ⇒ (ores_null vr) → (nat_is_zero (z2n hi))
```

The two rules of ores_null come from the rules of res_null:

- the first rule of res_null (precondition vr=VRef_null) has been abstracted into a rule having precondition condition vr=ZERO.
- rules 2, 3 and 4 of res_null have been collapsed into a single abstracted rule ores_null_2.

We see that the first rule is redundant with the second because (nat_is_zero(z2n ZERO)) is equal to True. We can therefore remove the first rule without changing the function. We thus add an abstraction command to the script to specify this action: **delete** res_null.1.

When we look at the second non deterministic function oifnull, the offensive version of ifnull:

```
function oifnull : abs_bytecode_idx → abs_jcvm_state → abs_returned_state :=

<oifnull_1> (ostack_f s) → Nil
 ⇒ (oifnull b s) → (oabortCode State_error);

<oifnull_2>
 (ostack s) → (Cons h lf),
 (head (oopstack h)) → (Value v)
  ⇒ (oifnull b s) → (oabortCode Type_error);

<oifnull_3>
   (ostack_f s) → (Cons h lf),
   (head (oopstack h)) → (Value v),
   (ores_null v) → True
      ⇒ (oifnull b s)
        → (oupdate_frame (oupdate_pc b
                                    (oupdate_opstack (tail (oopstack h)) h))
           s);

<oifnull_4>
   (ostack_f s) → (Cons h lf),
   (head (oopstack h)) → (Value v),
   (ores_null v) → False
      ⇒ (oifnull b s)
        → (oupdate_frame (oupdate_pc (succ (op_count h))
                                    (oupdate_opstack (tail (oopstack h)) h))
          s);

<oifnull_5>
   (ostack_f s) → (Cons h lf),
   (head (oopstack h)) → Error
      ⇒ (oifnull b s) → (oabortCode Opstack_error).
```

we see that the overlapping rule 2 corresponds to a type error, which is not to be considered in the offensive JCVM. Thus we add in the abstraction script the general command **remove** `Type_error`, which effect is to remove all rules containing `Type_error`.

The second script is therefore the following:

```
build oifnull from ifnull using
abstraction functions abstract_valu_prim abstract_valu
 abstract_valu_ref abstract_frame abstract_stack abstract_jcvm_state
 abstract_sheap abstract_returned_state abstract_heap_idx
 abstract_bytecode_idx abstract_o_data abstract_type_instance
 abstract_obj abstract_heap
delete res_null.1
remove Type_error
```

and, once the JTK has been applied again, the new offensive versions of `ifnull` and `res_null` are the following:

```
function  ores_null  : abs_valu_ref → bool :=
<ores_null_2> vr → hi ⇒ (ores_null vr) → (nat_is_zero (z2n hi))


function oifnull : abs_bytecode_idx → abs_jcvm_state → abs_returned_state :=

<oifnull_1> (ostack_f s) → Nil
 ⇒ (oifnull b s) → (oabortCode State_error);

<oifnull_3>
   (ostack_f s) → (Cons h lf),
   (head (oopstack h)) → (Value v),
   (ores_null v) → True
      ⇒ (oifnull b s)
         → (oupdate_frame (oupdate_pc b (oupdate_opstack (tail (oopstack h)) h) s);

<oifnull_4>
   (ostack_f s) → (Cons h lf),
   (head (oopstack h)) → (Value v),
   (ores_null v) → False
      ⇒ (oifnull b s)
         → (oupdate_frame (oupdate_pc (succ (op_count h)) (oupdate_opstack (tail (oopstack h)) h)) s);
<oifnull_5>
   (ostack_f s) → (Cons h lf),
   (head (oopstack h)) → Error
      ⇒ (oifnull b s) → (oabortCode Opstack_error).
```

These functions are translated into Coq definitions as follows:

```
Definition ores_null [arg0_res_null:abs_valu_ref] :=
  (nat_is_zero (z2n arg0_res_null)).

Definition oifnull [b:abs_bytecode_idx] [s:abs_jcvm_state] :=
  Cases (ostack_f s) of
    Nil ⇒ (oabortCode State_error)
  | (Cons h lf) ⇒
      Cases (head ? (oopstack h)) of
        (Value v) ⇒
          Cases (ores_null v) of
            True ⇒
              (oupdate_frame
                (oupdate_pc b (oupdate_opstack (tail ? (oopstack h)) h))
                s)
          | False ⇒
              (oupdate_frame
                (oupdate_pc (succ (op_count h)) (oupdate_opstack (tail ? (oopstack h)) h))
                s)
          end
      | Error ⇒ (oabortCode Opstack_error)
      end
  end.
```

6.4 Initial lemmas and predicates

By applying the offensive script shown above, we obtain 16 Coq files containing commutation lemmas as described in section 5.2.1 with their premises (predicates $Prec_{\hat{f}}$ described in section 5.2.1). The predicate and the lemma generated for `res_null` are the following:

```
Definition prec_res_null := [v:valu_ref] ¬(v = VRef_null).

Lemma res_null_eq:(v:valu_ref) (prec_res_null v) → (ores_null (abstract_valu_ref v)) = (res_null v).
 Intro v. ...
 <default proof>
```

We see that the predicate `prec_res_null` specifies that the commutation of `res_null` must be proved only when it is not applied to `VRef_null`. This precondition has been added automatically by the mechanism described in section 5.2.1: the command **`delete`** `res_null.1` has removed the first rule (deleted conclusion) of `ores_null`.

The proof is successful: Coq type-checks and compiles this proof without problem. But we will see that, due to the premise of `res_null_eq`, the predicate generated for `ifnull` (`prec_ifnull`, below) is too strong by propagation. It is generated as a conjonction of three premises as follows:

```
Definition prec_ifnull:=[b:bytecode_idx][j:jcvm_state]

 ¬(ifnull b j) = (abortCode Type_error)

∧   ((((vr:valu_ref)(v:valu)(lf:(list frame))(h:frame)
             (stack_f j) = (Cons ? h lf)
     ∧(head (opstack h)) = (Value ? v)
     ∧                 v = (VRef vr)
     ∧     (res_null vr) = False)
   → (prec_res_null vr))

∧   ((((vr:valu_ref)(v:valu)(lf:(list frame))(h:frame)
             (stack_f j) = (Cons ? h lf)
     ∧(head (opstack h)) = (Value ? v)
     ∧                 v = (VRef vr)
     ∧     (res_null vr) = True)
   → (prec_res_null vr)).

Lemma ifnull_eq:(b:bytecode_idx)(j:jcvm_state)(prec_ifnull b j)
 →  (oifnull (abstract_bytecode_idx b) (abstract_jcvm_state j))
    = (abstract_returned_state (ifnull b j)).
```

The first premise has been automatically generated by the JTK when applying the command **`remove`** `Type_error` to the function `ifnull` (deleted conclusion), as explain in section 5.2.1.

The second and third premises specify that `oifnull` commute with `ifnull` only when calls to `ores_null` verify `prec_res_null`. They have been generated by propagation of the premises of `res_null_eq`. Indeed the following rules of the definition of `if_null` (figure 4) contain calls to `res_null`:

```
<ifnull_3> (stack_f s) → (Cons h lf),
    (head (opstack h)) → (Value v),
                   v → (VRef vr),
         (res_null vr) → True (* Call to res_null *)
 ⇒ (ifnull b s) → (update_frame (update_pc b (update_opstack (tail (opstack h)) h)));

<ifnull_4> (stack_f s) → (Cons h lf),
    (head (opstack h)) → (Value v),
                   v → (VRef vr),
        (res_null vr) → False (* Call to res_null *)
 ⇒ (ifnull b s) → (update_frame (update_pc (S (p_count h)) (update_opstack (tail (opstack h)) h)));
```

But since `if_null` is a bytecode, the main commutation theorem (see section 5.2.1) should be the following:

```
Definition prec_ifnull:=[b:bytecode_idx][j:jcvm_state] ¬(ifnull b j) = (abortCode Type_error)).
```

In other word, the abstraction command **`delete`** `res_null.1` has added an unwanted premise to `prec_res_null`, which has been propagated to `prec_ifnull`, making this main lemma incompatible with the desired main commutation theorem. We therefore need to remove the wrong premise of `prec_res_null` to have the correct lemma statements.

6.5 Final abstraction script

More generally, when adding an abstraction command, *one must decide if it should be mirrored in the generated lemmas*. For example in our example we have to remove the first rule of `res_null` with a **`delete`** command, but we don't want this command to modify the predicate `prec_res_null`. To this purpose, each abstraction command `com` has a dual command `com_noprec` which is used to apply the command `com` but with no consequence on the generation of lemmas. Therefore a correct abstraction script for `if_null` is finally the following:

```
build oifnull from ifnull using
abstraction functions abstract_valu_prim abstract_valu abstract_valu_ref
abstract_frame abstract_stack abstract_jcvm_state abstract_sheap
abstract_returned_state abstract_heap_idx abstract_bytecode_idx abstract_o_data
abstract_type_instance abstract_obj abstract_heap

delete_nopred res_null.1
remove Type_error
```

The generated functions are the same as before, but the lemmas for `res_null` and `if_null` are the following, notice that this time no predicate is generated for `res_null`, and thus no premise propagation is done for `ifnull`:

```
Lemma res_null_eq:(v:valu_ref) (ores_null (abstract_valu_ref v)) = (res_null v).
 Intro v. ...
 <default proof>
 Save.

Definition prec_ifnull:=[b:bytecode_idx][j:jcvm_state] ¬(ifnull b j) = (abortCode Type_error)).


Lemma ifnull_eq:(b:bytecode_idx)(j:jcvm_state)(prec_ifnull b j)
    →  (oifnull (abstract_bytecode_idx b) (abstract_jcvm_state j))
        = (abstract_returned_state (ifnull b j)).
 Intros b j. ...
 <default proof>
 Save.
```

6.6 Conclusion

With this last script, all Coq files are type-checked and compiled without any other interaction, which means that it is formally proved that `oif_null` and `if_null` commute in the sense of the diagram of figure 2.

Unlike for `ifnull`, the abstraction and the commutation proof of most of the bytecodes (75%) do not require any user intervention. We found out however that a theorem prover with an interactive proof process, or at least with a good counterexample mechanism, is very useful during the phase of building the correct defensive machine and/or abstraction script.

7 Conclusion and related work

Formal verification of the JavaCard platform is the key to Common Criteria evaluations at the highest levels and a challenge for proof assistants. A recent survey [23] highlights the importance that formal methods have had in understanding and improving Sun's specifications of the Java(Card) platforms. But at the same time [23] observes a need to build appropriate environments that provide coherent support for formal specification, verification and provably correct implementations of the JavaCard platform. Such environments should incorporate general-purpose tools for developing and managing large-scale specifications and proofs, and specialized tools that are explicitly designed to reason about smartcard platforms. The design of such environments is a largely open avenue, and a promising field of research.

7.1 Related work

Formal methods for smartcards. It is commonly agreed There are many facets to formal verification for smartcards, each of which comes with its own objectives and techniques. For example:

- program verification at source level may help developers to ensure that their applets are correct. One definite advantage of such a scenario is that the developer (or an expert in formal methods working with him) can provide a formal specification of the program, and then check/prove that the applet is correct with respect to its specification. There is a wide range of formalisms to express such specifications, such as predicate, temporal and dynamic logics or interface specification languages [25]. The corresponding verification techniques include run-time checking and unit testing [16], static checking [28], model-checking [18], and interactive theorem proving [10];

- program verification at bytecode level may help to check that it is safe to load and execute an applet whose origin is unknown or doubtful. As one cannot expect code consumers to engage in program verification, it is important that this kind of verification uses automatic techniques such as static analyses and model-checking [13, 36], or relies on a scheme which delegates the verification tasks to the code producer [32].

  There have been a number of related efforts, both to prove the standard bytecode verifier correct [11, 33, 38], and to suggest new bytecode verifiers based on more complex type systems [31]. Most of this work is tailored towards a particular security policy, to the exception of [33], where it is shown how to use a data flow analyzer to construct a bytecode verifier from an abstract virtual machine. In contrast, Jakarta is focused on deriving abstract virtual machines from defensive ones, and proving them correct. We are not aware of any similar effort, despite tremendous activity in the field—see *e.g.* [23] for a recent survey of ongoing work.

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. Many of these difficulties can be solved by using a dedicated prototyping environment that supports some/most administrative aspects of formal semantics. A number of tools have been designed for this purpose, including ASF+SDF [20], Centaur [14], Letos [22] and RML [35]. These tools are rather diverse in their design and functionalities, but they all provide:

- a readable format for giving a formal semantics to programming languages;

- support to execute the semantics, which in particular allows to check that the semantics reflects the intended behavior of the language.

However, prototyping environments lack functionalities to reason formally about the semantics, in contrast to proof assistants such as Coq [17], Isabelle [34] or PVS [37], which feature advanced type systems that support precise specifications and sophisticated reasoning.

In order to deal with real-size programming languages, there is thus considerable interest in integrating prototyping environments and proof assistants. Despite preliminary work in this direction as pointed out in section *Bytecode verification* above, we still lack a tool that reflects manipulations in the specification environment of prototyping systems directly into the reasoning environment of proof assistants.

7.2 Future work

Other type systems and language We have made good our promise to extract the offensive and abstract JCVMs from the defensive JCVM for the type safety property. One line of future research is to do likewise for novel type systems for Java(Card) in order to verify other properties, see e.g. [4, 21, 27, 38]. More generally, we are interested in understanding whether our methodology applies to other typed low-level languages [2, 19, 39]. We are currently building up appropriate defensive virtual machines that verify (1) initialization and (2) information flow.

Support the construction of refined virtual machines Currently, JTK only supports abstractions. We are planning to extend it to support interactive refinements, *i.e.* tools to define a function $f' : \sigma' \to \sigma'$ from a previously defined function $f : \sigma \to \sigma$ and a refinement function $\alpha : \sigma' \to \sigma$; here it is assumed that σ' is more precise than σ, and that α forgets some of the information contained in σ'. In the context of reasoning about the JavaCard platform, refinement is useful *e.g.* to construct a JCVM with a more precise memory model—currently our formalization does not treat transient objects.

Proof Automation We have also automated to a large extent the correctness proofs of the abstraction. Some of the Coq tactics we have designed in this purpose will be integrated in the next Coq distribution. We are working on a generalization of the mechanism for auxiliary lemmas generation. 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 applied to (try to) prove those lemmas.

Once the commutation of a function is proved, we currently use the `AutoRewrite` tactic of Coq: each auxiliary 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 adapting existing connections between Coq and external tools, such as Elan [1].

In a different line of work, Sorin Stratulat has been experimenting cross-validation of the JCVMs using Spike [15], an automatic theorem prover based on implicit induction. Currently Spike has been used to establish automatically the commutation properties for over two thirds of the instruction set§, but we expect that further optimizations on Spike should allow us to handle even more instructions. This experience, which will be reported elsewhere, suggests that first-order theorem proving techniques could be advantageously exploited in the context of platform verification for smartcards.

§ still true?

Extending JSL The paper reports on preliminary results with the implementation and use of Jakarta. Clearly, much work remains to be done, *e.g.* extending JSL with modules, extending JPI to support other proof assistants, programming languages, and modifying JPI to translate JSL specifications as inductive relations... Below, we single out three important directions for future research.

Acknowledgments

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] D. Aspinall and A. Compagnoni. Heap-bound assembly language. Manuscript, 2001.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[4] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.

[5] 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.

[6] G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In H. Kirchner and C. Ringessein, editors, *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer-Verlag, 2002.

[7] G. Barthe and G. Dufay. Verified ByteCode Verifiers Revisited. Manuscript, 2003.

[8] 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.

[9] 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.

[10] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.

[11] Y. Bertot. Formalizing in Coq a type system for object initialization in the Java bytecode language. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 14–24. Springer-Verlag, 2001.

[12] M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002.

[13] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic Purse Applet Certification. In S. Schneider and P. Ryan, editors, *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 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] Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of ECOOP'02*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, 2002.

[17] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.2*, January 2002.

[18] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of ICSE'00*, pages 439–448. ACM Press, 2000.

[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] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: an algebraic specification approach*. AMAST Series in Computing. World Scientific, 1996.

[21] 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.

[22] P. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software–practice and experience*, 29(5):1379–1416, September 1999.

[23] 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.

[24] JavaCard Technology. `http://java.sun.com/products/javacard`.

[25] JML Specification Language. `http://www.jmlspecs.org`.

[26] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. Submitted.

[27] C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 200x. To appear.

[28] K. R. M. Leino. Extended Static Checking: A Ten-Year Perspective. In R. Wilhelm, editor, *Informatics–10 Years Back, 10 Years Ahead*, volume 2000, pages 157–175, 2001.

[29] 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.

[30] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.00*, 2000.

[31] P. Müller and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In S. Drossopoulou, editor, *Proceedings of Formal Techniques for Java Programs*, 2001.

[32] G. C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[33] 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.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[35] M. Petersson. *Compiling Natural Semantics*. PhD thesis, Linköping University, 1995.

[36] J. Posegga and H. Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Proceedings of the ESORICS'98*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.

[37] N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.

[38] 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.

[39] K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Manuscript, 2001.