A Uniform and Certified Approach for Two Static Analyses

Solange Coupet-Grimal and William Delobel

Rapport/Report 24-2005

26 April 2005

A Uniform and Certified Approach for Two Static Analyses

Solange Coupet-Grimal and William Delobel

LIF – Laboratoire d'Informatique Fondamentale de Marseille
UMR 6166
CNRS – Université de Provence – Université de la Méditerranée

CMI, 39 rue Frédéric Joliot-Curie 13453 Marseille Cedex 13

{Solange.Coupet, Delobel}@lif.univ-mrs.fr

Abstract/Résumé

We give a formal model for a first order functional language to be executed on a stack machine and for a bytecode verifier that performs two kinds of static verifications : a type analysis and a shape analysis, that are part of a system used to ensure resource bounds. Both are instances of a general data flow analyzer due to Kildall. The generic algorithm and both of its instances are certified with the Coq proof assistant.

Keywords: Mobile code, formal verification, Calculus of Constructions.

Nous présentons un modèle formel pour un langage fonctionnel du premier ordre exécuté sur une machine à pile et pour un vérificateur de bytecode qui effectue deux types de vérifications statiques : une analyse de types et une analyse de formes. Les deux sont utilisées dans un système visant à garantir une borne des ressources et sont des instances d'un algorithme générique du à Kildall. L'algorithme et ses deux instances, sont certifiés dans l'assistant de preuves Coq.

Mots-clés : Code mobile, vérification formelle, Calcul des Constructions.

Relecteurs/Reviewers: Roberto Amadio and Silvano Dal Zilio.

1 Introduction

Over the last decade, research on mobile code has been a hot topic and intensive efforts have been made to reduce the risk of malicious (Java) applets performing a security attack. For this, a crucial functionality of the Java Platform is the bytecode verifier which performs a static type analysis on programs. This kind of analysis ensures integrity properties of the execution environment such as the absence of memory faults. Consequently, there has been considerable interest in specifying formally the Java Virtual Machine and proving the correctness of its bytecode verifier (see for instance [BDJMdS02, BDJ$^+$01, Nip01, KN03] ...).

More recently, these investigations have been extended to establishing an additional property that contributes to guarantee the safety of bytecode by ensuring bounds on the computational resources needed by its execution. Within this context, a project has been undertaken [ACGDZJ04] which focuses on a rather standard first-order functional programming language with inductive types, pattern matching, and call-by-value, to be executed on a simple stack machine. The language comes with various bytecode static analyses: a standard type analysis, an analysis on the algebraic shape of the values in the stack, an analysis of the size of these values, and an analysis that insures the termination. The last three analyses, and in particular their combination, are instrumental in predicting the space and time required for the execution of a program.

This paper deals with the formal specification of the virtual machine related to this language and the certification in the Coq proof assistant of an extended bytecode verifier which performs the first two phases of the analysis, that is the type and shape verifications. Our contribution is threefold. First, we present a verifier designed in a uniform way, so as both verification processes become special cases of a general framework for data flow analysis based on the well known algorithm due to Kildall [Kil73]. Second, we propose a functional specification in Coq of Kildall's algorithm and we prove its correctness. This is related to Klein and Nipkow's work with the system Isabelle/HOL [Nip01, KN03]. However our formalization differs from theirs by the heavy use of Coq dependent types and the way of encoding a recursive function which is not structurally recursive. With this approach, properties common to both analyses are established once and for all. Note that this approach also suits size analysis, although this remains work to be done in future. Third, these generic properties are used, for proving not only the correctness of type verification which is now quite standard, (see [Gol98, Nip01]), but also, and this is the novelty, that of algebraic shape verification.

The paper is organized as follows. The first section is a quick description of the problem: we present the language, the bytecode instructions, and both kinds of analyses. Section 3 is dedicated to encoding Kildall's algorithm in Coq and proving its correctness. Section 4 is related to both type and shape analyses which are formally specified and proved correct. In section 5 we make a comparison with related work and give a conclusion. Some proofs are detailed in the appendix.

2 The functional language and the bytecode instructions

A formal and rigorous description of the source language, the bytecode instructions, their operational semantics, and of the type and shape analyses can be found in [ACGDZJ04]. For lack of space, we have chosen to give here an informal but intuitive presentation that we illustrate by examples to give a quick understanding of the problem. The source language is a first order functional language, with (mutually) inductive types. Functions are defined by a sequence of pattern matching rules of the form $f(p_1, \ldots, p_n) = e$, where e is an expression and $p_1, \ldots, p_n$ are linear patterns (a variable occurs at most once). As an illustration, Fig.1 displays a program that evaluates boolean expressions. Each function is compiled to bytecode to be executed by a virtual machine. At run time, a frame *(f, pc, P)* is created at each function call: f is the function's name, *pc* is the program counter that indicates the index of the current instruction (initially 0), and P is a stack of values that is initialized by the arguments of the function. This frame is pushed on the top of the current configuration, that is the stack constituted by all the frames of the functions currently active. Fig.2 shows a possible bytecode program for function *member* in Fig.1, as well

```
type  bool = T | F ;;
type  nat  = Z  | S of nat ;;
type  env  = Nil | C of nat * env ;;
type  form = Var of nat| Not of form | Or of form * form | Ex of nat * form ;;

not (T) = F        or (T,y) = T       eq (Z,Z)       = T         eq (S(x),Z)   = F
not (F) = T        or (F,T) = T       eq (S(x),S(y)) = eq (x,y)  eq (Z,S(x))   = F
                   or (F,F) = F
member (x,Nil)     = F
member (x,C(y,l)) = or(eq(x,y),member(x,l))

check (Var(x), l)    = member (x,l)                              qbf (f) = check(f,Nil)
check (Not(f1), l)   = not (check(f1,l))
check (Or(f1,f2), l) = or (check(f1,l),check(f2,l))
check (Ex(x,f1), l)  = or (check(f1,l),check(f1, C(x,l)))
```

Figure 1: A program for evaluating boolean formulae

as a symbolic execution of the function on natural number x and environment l. Each line shows the value pc of the program counter, the related instruction, and the expression stack on which the instruction of rank pc is executed (the top of the stack is on the left). Let us describe the

```
member:
  0:  load(1);                                                 [l  x]    Id
  1:  branch("Nil",4);                                 [l       l  x]    Id
  2:  build("F",0);                                          [Nil  x]    l=Nil
  3:  return;                                          [F     Nil  x]    l=Nil
  4:  branch("C",13);                                  [l       l  x]    Id
  5:  load(0);                                      [t  h  C(h,t)  x]    l = C(h,t)
  6:  load(2);                                   [x  t  h  C(h,t)  x]    l = C(h,t)
  7:  call("eq",2);                    [h         x  t  h  C(h,t)  x]    l = C(h,t)
  8:  load(0);                             [eq(x,h)  t  h  C(h,t)  x]    l = C(h,t)
  9:  load(3);                         [x   eq(x,h)  t  h  C(h,t)  x]    l = C(h,t)
  10: call("member",2);            [t   x   eq(x,h)  t  h  C(h,t)  x]    l = C(h,t)
  11: call("or",2);          [member(x,t)   eq(x,h)  t  h  C(h,t)  x]    l = C(h,t)
  12: return;              [or(eq(x,h),member(x,t))  t  h  C(h,t)  x]    l = C(h,t)
  13: stop;                                            [l       l  x]    Id
  14: return;           ⊥
```

Figure 2: Symbolic execution of function *member*

operational semantics of the instructions that appear in this piece of bytecode.

- Instruction *load*(j) pushes on top of P the element of rank j, counting from the bottom of the stack. pc is incremented. Note that the bottom of P is rank 0.
- Instruction *branch*(c, j) matches element e on the top of the stack with constructor c of arity m. If the matching succeeds, e is popped out from the stack, it is deconstructed and its m arguments are pushed on the stack. pc is incremented. If the matching fails, the stack is left unchanged and pc is set to j.
- Instruction *build*(c, m), where c is a constructor of arity m, discards the m values $v_1, \ldots, v_m$ on the top of P and pushes $c(v_1, \ldots, v_m)$. Program counter pc is incremented.

- Instruction *return* returns to the environment the result on the top of the stack. The current function is deactivated, that is its frame is popped out from the current configuration.
- Instruction *stop* stops the execution.
- Instruction $call(g, m)$ pushes, over the frame of the calling function, a new frame for executing function g of arity m. In this new frame, the program counter is set to 0 and the value stack is constituted by m values, popped out from the value stack of the calling function.

2.0.1 Shape analysis as a symbolic execution

Fig.2 is nothing but the result of a symbolic execution of function *member* in the sense that the stack contains algebraic expressions, built with constructors, functions names and variables, instead of values built from constructors only (as in a true execution). In particular, the arguments of the function have been replaced by two variables x and l. Within this context of symbolic execution, each instruction $branch(c,\ j)$ gives raise to a substitution that matches the expression on the top of the stack with a pattern $c(x_1, \ldots, x_m)$. Here, m is the arity of constructor c and $x_1, \ldots, x_m$ are fresh variables. Substitutions in the last column of Fig.2, keep track of these pattern matchings. The current substitution is updated when encountering a *branch* instruction, by composing it with that resulting from the new pattern matching. Symbolic execution furnishes information on the shape of the values in the stack during an actual execution. Although it is out of the scope of this paper, let us mention that this kind of analysis can be used to ensure bounds on the resources required by a program execution. For that, it is assumed that the bytecode comes with polynomial annotations for the constructors and the functions. These polynomials (introduced by Marion [Mar00] under the name of quasi-interpretations) are intended to provide bounds on the size of the values built with the constructors or returned by the functions. A bound on the overall memory used by the program is then deduced from considerations on termination. One can refer to [ACGDZJ04] for more details.

2.0.2 Type analysis as an abstract execution

The bytecode is also supposed to be accompanied by type annotations, that is functions' and constructors' signatures. Type analysis comes before shape analysis. It consists in an abstract execution of the bytecode, in which the values are replaced by their types. The type verification checks the compatibility between the types in the stack and the current instruction, with respect to the signatures. When executing an instruction $branch(c,\ j)$ for instance, one checks that the value on the top of the stack is the type of constructor c. In case of error, an error state $\top$ is produced. One proves, and this is now quite standard, that if the bytecode is correctly typed, there will be no out of bounds access to the value stack, all function calls will apply to well typed arguments, and each new value is built with a constructor and arguments the types of which are consistent ...

Both abstract and symbolic executions can be viewed as instances of a generic data flow analyzer due to Kildall ([Kil73]) that we present in the next section.

2.0.3 Notations

In the sequel, we will use the following notations :
$-e :: l$ is the list with head e and tail l
$-l@l'$ is the concatenation of lists l and l'
$-|l|$ is the size of list l
$-[v_0; \ldots; v_k]$ is the list of elements $v_0, \ldots, v_k$. So $[\,]$ is the empty list.
$-l[i]$ the i^{th} element of the list, elements being indexed from 0
$-e \in l$ means e is an element of list l

3 Kildall's algorithm

3.1 Parameterizing the generic data flow framework

Kildall's algorithm traverses the control flow graph of a function. It is a graph whose vertices are the instructions' indices in the bytecode program. There is an edge between p and q if instruction of index q can be executed immediately after that of index p. This graph has an only source, 0, that corresponds to the entry point of the function. The generic data flow framework is parameterized by:

- the number n of instructions of the bytecode, that characterizes the set $\{0, \ldots, (n-1)\}$ of the vertices also called *instructions* when it is clear from the context.
- a function *succs*, which associates with each instruction in $\{0, \ldots, (n-1)\}$ the list of instructions that can immediately follow it. It characterizes the set of the edges.

Moreover, with each vertex of the graph is associated a *state*, which generalizes the symbolic stack in front of each instruction in Fig.2, or a type stack in case of the type analysis. Thus, we introduce:

- σ, the type of the states. σ is equipped with a relation $>_\sigma$, a supremum function sup_σ, and a top element $\top$. Moreover, we assume that all ascending chains in σ are finite. Intuitively, the states can be seen as constraints on the value stacks handled by the virtual machine during the evaluation of a function. Relation $>_\sigma$ compares constraint strength: if instruction p can be executed in state s, it has to be so for every state s' such that $s >_\sigma s'$.

The length-n lists ss of elements of σ are called $function\ states$. In the case of symbolic analysis for example, they correspond to the lists of n symbolic stacks as those displayed in Fig.2.

The algorithm also relies on a flow function *step* that takes as arguments an instruction p and a state s. *(step p s)* is the list of all states (one for each possible successor of instruction p) resulting from the execution of p in state s. Functions *step* and *succs* are combined to define function *step*$'$ such that, if $(step\ p\ s) = [t_1, \ldots,\ t_k]$ and $(succs\ p\ s) = [q_1, \ldots,\ q_k]$ then $(step'\ p\ s) = [(q_1,\ t_1),\ \ldots,\ (q_k, t_k)]$.

3.2 Description of the algorithm

As already mentioned, the algorithm traverses the function's flow graph. A function state ss associates state $ss[p]$ in σ with each vertex p. In practice, initially all the vertices except 0 will have a special state that represents a constraint always satisfiable. It is encoded as the least element of σ (which is not mentionned in the previous section since it is not used for specifying and proving the algorithm). For vertex 0, the initial constraint depends on the kind of analysis that is performed. When an instruction p is reached, a call $(step'\ p\ ss[p])$ computes a new state t for each successor q of p. The current state $ss[q]$ is updated by $(sup_\sigma\ t\ ss[q])$. If $ss[q]$ has actually been modified, q is moved to the *working list* of the instructions to be examined again. The process goes on until stability. Therefore, the stability of an instruction p with respect to a function state ss is defined as follows:

$$(stable\ ss\ p) := \forall (q,t) \in (step'\ p\ ss[p]),\ ss[q] \geq_\sigma t$$

Kildall's algorithm starts with any function state ss and the list (called *(worklist ss)*) of all instructions p that are not stable for ss. It calls a main loop, *iterate*, which takes as arguments a function state ss and a working list w. Iterate examines each element p in w to make it stable. This is achieved through a call to the propagation function *propagate*, which, for all successors q of p, updates $ss[q]$ and adds q to w if $ss[q]$ has been changed.

```
$(kildall\ ss) := (iterate\ (ss, (worklist\ ss)))$
$(iterate\ (ss,\ w))$ := **match** $w$ **with**
                    $[\,] \Rightarrow ss|$
                    $p :: w' \Rightarrow (iterate\ (propagate\ ss\ w'\ (step'\ p\ ss[p])))$
$(propagate\ ss\ w\ l)$ := **match** $l$ **with**
                    $[\,] \Rightarrow (ss, w)|$
                    $(q, t) :: l' \Rightarrow$ **if** $(sup_\sigma\ t\ ss[q]) = ss[q]$ **then** $(propagate\ ss\ w\ l')$
                                   **else** $(propagate\ ss[q \leftarrow (sup_\sigma\ t\ ss[q])]\ \ q :: w\ \ l')$
```

It can be noticed that the propagation function is defined by a recursion on the structure of its third argument. In contrast, function *iterate* is not defined by structural recursion. This requires us to exhibit a well-founded order on pairs (ss, w), and a non-trivial building of *iterate* from a termination proof. This is detailed in the second part of section 3.3. Lastly, remark that although quite straightforward, the definition of *propagate* requires the equality to be decidable on σ.

3.3 Encoding in Coq Kildall's algorithm

As far as the algorithm's specification in Coq is concerned, and comparatively to Klein and Nipkow's work with Isabelle [Nip01, KN03], two main points must be emphasized: the use of dependent types and the way of encoding function *iterate*. These are the essential differences and we shall briefly discuss the advantages of each approach.

Specifying with dependent types As we have seen, *function states* are lists of states whose length is meant to remain constant and equal to the number n of instructions in the function's bytecode. This data type is encoded quite naturally in Coq by a dependent type:

```
Inductive sized_list : nat -> Set:=
  sd_nil : (sized_list 0)|
  sd_cons : (∀ n:nat), α -> (sized_list n)-> (sized_list (S n)).
```

Notice that these lists are polymorphic: they depend on set α that is supposed to be declared as a parameter in the current section. Outside the section, α is discharged and thus must appear explicitly in the type. This specification avoids the presence of hypotheses of the form $|ss| = n$ in a great number of lemmas, all throughout the development. Similarly, we define inductively the lexicographic order and the componentwise order on lists with a type that expresses that only same-length lists can be compared:

```
 lex_n, <_n :  (sized_list n) -> (sized_list n) -> Prop
```

With this approach, we can prove by induction on parameter n that the lexicographic order is well-founded provided the underlying order on the elements is well-founded. Since for all natural numbers n, $\texttt{lex}_n$ is weaker than $<_n$, we can conclude that $<_n$ is well-founded too.

Function *succs* computes the list of the successors of an instruction p. Instruction p and its successors all must be natural numbers less than n. Instead of taking the hypothesis:

$$(\forall p : nat),\ p < n \rightarrow (\forall q : nat),\ q \in (succs\ p) \rightarrow q < n \qquad .$$

we define the type *dep_list* of the lists the elements of which satisfy a certain predicate:

```
dep_list : (∀ α: Set)(α -> Prop) -> Set.
```

So, we obtain the type `d_list` of the lists of natural numbers less than n as an instance of this data type :

```
d_list : =λn:nat.(dep_list  nat  λp:nat.p<n)
```

Consequently, the types of functions *succs*, *step*, and *step'* are the following:

```
succs : (∀ p : nat), p<n -> (d_list n)
step  : (∀ p : nat), p<n -> σ -> (list σ)
step' : (∀ p : nat), p<n -> σ -> (dep_list  nat*σ  λ(q,t):nat*σ.q<n)
```

As an example, a version without dependent types leads to establishing

$$propagate(ss, w, l) = (ss', w') \to |ss'| = |ss|$$

while this is implicitly stated in the type of function *propagate* in our development.

```
propagate: (sized_list σ n) -> (d_list n)-> (dep_list nat*σ λ(q,t):nat*σ.q<n) ->
           (sized_list σ n) * (d_list n)
```

To be fair, this has been achieved through an increased effort on preliminary results, mainly concerning lists (consequent libraries on lists with dependent types have been built). But doing so, we "factorize" some proofs by moving them from specialized parts of the development to generic ones. Thus they become reusable and they are performed once and for all. Moreover, let us point out that not only the statements of the lemmas are simplified but also the proofs and the use of the lemmas, since in their applications, fewer hypotheses must be shown to be satisfied. However, using dependent types raises some difficulties. In such specifications there is a strong interdependence between logical parts, namely proof terms elegantly expressing constraints on data types, and purely computational parts. In practice one has to establish that these logical parts are irrelevant as far as computational aspects are concerned. For example, as the elements of lists of type *dep_list* are pairs made of an element and a certificate, one must define a projection :

```
dep_list_to_list : (∀ α:Set)(∀P: α -> Prop) (dep_list α P) -> (list α).
```

and an equivalence on dependent lists by:

```
l ≡ l' := (dep_list_to_list l) = (dep_list_to_list l')
```

The specification and the verification of Kildall's algorithm are performed under the following hypotheses:

```
(H1): (∀ p : nat)(∀ C: p < n)(∀ s:σ), |(succs p C)| = |(step p C s)|
(H2): (∀ p : nat)(∀ C,C': p < n),  (succs p C) ≡ (succs p C')
(H3): (∀ s : σ)  (∀ p: nat) (∀ C,C': p < n), (step p C s) = (step p C' s)
(H4): (∀ p : nat)(∀ C: p < n)(∀ s, t:σ), s≤_σt -> (step p C s)≤(step p C t)
```

In $(H4)$, $\leq$ stands for the componentwise relation on the standard lists. This hypothesis expresses the monotonicity of function *step*.

Defining function *iterate* by well-founded induction The specification of function *iterate* in Coq is not straightforward since this system only supports total functions defined by structural recursion. This specification must include a proof of termination within its structure. The approach that we take here is due to Yves Bertot and Antonia Balaa [BB02]. The term is built by approximations in a style inspired by Tarski's fixpoint theorem. Here are the main steps for constructing the term *iterate*.

1. We define a family of relations $\prec_n$, that we prove to be well-founded, on the set of pairs
   `(ss, w):  (sized_list σ n)*(d_list n)`
   by: `(ss',w')`$\prec_n$`(ss,w) := (ss'`$>_n$`ss)` $\vee$ `(ss=ss'` $\wedge$ `(|w'| < |w|)`
2. We prove that for all n-length function states ss, instructions p, proofs $C : p < n$, and instruction lists w:
   `(propagate ss w (step' p C ss[p]))` $\prec_n$`(ss, p::w)`

   Thus, the recursive call in the evaluation of (*iterate* $(ss,\ w)$) is on an argument strictly less than $(ss,\ w)$ with respect to the well-founded relation $\prec_n$.

3. Let F be the functional defined by:
```
(F f) = λ(ss,w) if w = [] then ss else
                 let (p, C)::w' = w in (f (propagate ss w' (step' p C ss[p]))).
```

   We prove that for any function *bot* on pairs *(ss, w)*

   $$\forall\texttt{(ss, w)}\ \exists\texttt{v}\ (\exists\texttt{k}_0\texttt{:nat})(\forall\texttt{k}>\texttt{k}_0)\ (\texttt{F}^{\texttt{k}}\ \texttt{bot (ss, w)) = v}$$

   For a given pair $(ss,\ w)$ the proof is performed by induction on the fact that this pair is accessible for relation $\prec_n$, which follows from the well-foundedness of the relation.

4. A proof term of such a statement is a pair *(f, h)* where:
   – f is a function which associates with each argument (ss, w) value v
   – h is a proof that $(\exists\texttt{k}_0\texttt{:nat})(\forall\texttt{k}>\texttt{k}_0)\ (\texttt{F}^{\texttt{k}}\ \texttt{bot (ss, w)) = v}$
   By deconstructing such a pair, it is possible to forget the logical comment h and to get the computational part of the term, that is program f.

5. This function f is in fact the function *iterate* that we intend to define. Indeed, we prove that f satisfies the fixpoint equation `(F f)=f`.

This is to be compared with the specification in Isabelle. In an early version of their work [Nip01], Klein and Nipkow used an opaque well-founded recursion whereas in a more elaborate version [KN03] they express the function in terms of the predefined *while*-combinator of type : $(\alpha \Rightarrow \mathit{bool}) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$ which satisfies the equation

$$\mathit{while}\ b\ c\ s = (\mathit{if}\ (b\ s)\ \mathit{then}\ (\mathit{while}\ b\ c\ (c\ s))\ \mathit{else}\ s)$$

This equation is a directly executable functional program. It makes it possible to define functions without proving any well-foundedness. However, to reason on such functions, establishing their termination is mandatory. As a matter of fact, proving that a certain property Q holds on a returned value *(while b c s)* is achieved through the following *while-rule*:

$P\ s \wedge (\forall s,\ P\ s \wedge b\ s \Rightarrow P\ (c\ s)) \wedge (\forall s,\ P\ s \wedge \neg\ b\ s \Rightarrow Q\ s) \wedge wf\ r \wedge (\forall s,\ P\ s \wedge b\ s \Rightarrow (c\ s,\ s) \in r)$
$\Rightarrow Q\ (\mathit{while}\ b\ c\ s)$

Moreover, this approach only applies to tail recursive functions.

3.4 Correctness of Kildall's algorithm

We mentioned that the greatest element $\top$ of σ stands for the error state. Therefore, all the bytecodes whose analysis generates a function state containing $\top$ will be rejected. In order to prove that only erroneous programs are rejected, we establish that Kildall's algorithm produces the least stable function state, greater than its argument (for relation $<_n$). This is done by using the monotonicity of function *step*. Now, how can this data flow analyzer be used for bytecode verification? And first of all what does it mean that the bytecode is correct with respect to a certain kind of analysis? This is expressed by a parameter *wi* to be later instantiated by a compatibility relationship between the instructions and a function state *ss*. For instance, in case of type verification, if instruction of index p is the *return* instruction, $(wi\ ss\ p\ \_)$ holds if and only if the element on the top of stack $ss[p]$ is less than or equal to the return type of the function. Assuming the following relationship between predicates *wi* and *stable*:

$(H5)$: `∀ss, ⊤ ∉ ss -> ((∀p: nat)(∀C: p<n), (wi ss p C) <-> (stable ss p C))` we can deduce the following two propositions:

$$\top \notin (\texttt{Kildall}\ \texttt{ss}) \ \rightarrow\ (\forall \texttt{p} : \texttt{nat})\ (\forall \texttt{C} : \texttt{p} < \texttt{n}),\ (\texttt{wi}\ (\texttt{Kildall}\ \texttt{ss})\ \texttt{p}\ \texttt{C}) \tag{1}$$

$$((\exists\, \texttt{ts} \geq_{\texttt{n}} \texttt{ss})\ (\forall \texttt{p} : \texttt{nat})\ (\forall \texttt{C} : \texttt{p} < \texttt{n}),\ (\texttt{wi}\ \texttt{ts}\ \texttt{p}\ \texttt{C}))\ \rightarrow \top \notin (\texttt{Kildall}\ \texttt{ss}) \tag{2}$$

We do not detail the proofs here. They are similar to that in [Nip01]. The differences are not in the proof schemes themselves, but rather in the specification style.

4 Application to two static analyses

Let us now apply this algorithm to perform type and shape analyses on the function bytecodes for the language introduced in section 2. We start by encoding in Coq the bytecode instructions and the virtual machine. This part of the development is parameterized by the set *name* of the names of types, functions, and constructors. The only axiom set on *name* is the decidability of equality over it.

4.1 The virtual machine

Instructions It is assumed that every function in the program passed the following preliminary verifications : for each instruction p, successors of p are valid indices in the function's bytecode. That condition falls into to parts :

- last instruction of a bytecode is not one of *load j*, *call g ar*, *build c ar* or *branch c j* (whose successors contain the instruction which immediately follows it in the bytecode)
- jump indices j in *branch c j* instructions are less than the length of the bytecode

We choose to represent the bytecode programs by using the dependent type of lists of fixed length n. This allows us to force the second condition directly in the instruction definition, by adding a third argument of type $j < n$ to instruction *branch*. Therefore, type *instruction* itself depends on n. It is defined as follows:

```
Inductive instruction (n:nat) : Set :=
  return : instruction n|
  stop : instruction n|
  load : nat -> instruction n|
  call : name -> nat ->instruction n|
  build : name -> nat -> instruction n|
  branch : name -> forall  (j:nat), j<n -> instruction n.
```

We can now introduce the following definition:

```
Definition bytecode:= (∀ n: nat), (sized_list n (instruction n)).
```

As expected, the successors set of an instruction of index p is $[p]$ for a *return* instruction, $[\,]$ for a *stop* instruction, $[p+1; j]$ for *(branch c j)*, and $[p+1]$ otherwise. The function that computes successors is encoded so as to produce lists of natural numbers less than n. It has type:

```
Succs: (∀ n: nat), (bytecode n) -> (∀ p: nat), p<n -> d_list n.
```

We can easily prove that function `succs:=(Succs n bc)` fulfills condition ($H2$) in section 3.3.

Functions Type `function` is that of records of the form $\tilde{f} = mkfun(f, sig_f, |f|, bc_f)$, where

- f, of type *name*, is the name of the function.
- sig_f, of type *name*(list name)*, is the signature of the function. It is a pair made of the return type, and the list of the arguments' types.

- $|f|$ is an integer that denotes the bytecode's length.
- bc_f is the bytecode of the function, of type ($bytecode\ |f|$).

For clarity, we describe record types in a simplified notation. For instance, if we denote by $\tilde{f}$ a term of type *function*, f is an abbreviation for $\tilde{\texttt{f}}$`.fun_name`. Similarly, sig_f stands for $\tilde{\texttt{f}}$`.fun_sig`, $|f|$ stands for $\tilde{\texttt{f}}$`.fun_size`, and bc_f stands for $\tilde{\texttt{f}}$`.fun_bytecode`.

Another parameter in this part of the development is `functions: (list function)` that represents the list of the functions the program is made of. Elements of this list are assumed to fulfill the following hypotheses :

$(H6)$: $(\forall \tilde{\texttt{f}} : \texttt{function})$, $\tilde{\texttt{f}} \in \texttt{functions} \rightarrow |\texttt{f}| > 0$
$(H7)$: $(\forall \tilde{\texttt{f}} : \texttt{function})$, $\tilde{\texttt{f}} \in \texttt{functions} \rightarrow \texttt{last\_return\_or\_stop}\ \texttt{bc}_\texttt{f}$

Here, *last_return_or_stop* is a predicate on lists of instructions that expresses that the last instruction of bc_f (i.e. element at index $|f| - 1$) is either a *return* or a *stop* instruction.
Function `Get_function: name → (Opt function)` will be used to find in list `functions` the first record with name field f. *Get_function*'s return type is optional to handle the case where no function named f appears in the program.

Constructors The type, `constructor` is that of records $\tilde{c} = mkcons(c, ret_c, args_c)$, where

- c, of type *name*, is the constructor's name.
- ret_c, of type *name*, is the name of the type built by $\tilde{c}$.
- $args_c$, of type *(list name)*, is the list of the types of its arguments.

As for functions, parameter `constructors: (list constructor)` contains all the constructors declared in the program. `Get_constr` plays for *constructors* a role similar to *Get_function* for *functions*.

Frames Their type `frame` is that of records $\bar{f} = mkfr(f, pc_f, stack_f, args_f)$, where

- f is the name of the function being evaluated in $\bar{f}$.
- pc_f is the index of the current instruction.
- $stack_f$, of type *(list value)*, is the value stack.
- $args_f$, of type *(list value)*, is the initial stack, i.e. the arguments on which function $\tilde{f}$ is evaluated.

Type `value` is that of trees the nodes of which are elements of type *name*. The fourth component in a frame does not appear in the description of the virtual machine as given in section 2. It must be considered as a dummy field without any computational relevance. It will only be used to achieve proofs. Let us point out that with these simplified notations, given a term $\bar{f}$ of type *frame*, f is an abbreviation for $\bar{\texttt{f}}$`.frm_name`. Similarly bc_f will stand for `x.fun_bytecode` where `(Get_function` $\bar{\texttt{f}}$`.frm_name) = (Some x)`. Therefore, invoking bc_f given a frame $\bar{f}$ implicitly induces the presence of a function named $\bar{\texttt{f}}$`.frm_name` in parameter *functions*.

Configurations Type `configuration` aliases *(list frame)* and it represents the machine states at runtime. The top frame (i.e. the most recent one) is the head of this list. The empty configuration [ ] represents an erroneous configuration. The instructions' semantics (see section 2) is encoded by a predicate `reduction` on the configurations. Let us take instruction *call* as illustrative example. Its formal semantics is expressed by the rule:

$$\frac{pc_f < |f| \qquad bc_f[pc_f] = \texttt{call}\ g\ ar \qquad g \in functions \qquad stack_f = [v_{ar}; \ldots; v_1] :: l}{(f,\ pc_f,\ stack_f,\ args_f) :: M \to (g,\ 0,\ [v_{ar}; \ldots; v_1],\ [v_{ar}; \ldots; v_1]) :: (f,\ pc_f,\ l,\ args_f) :: M} \quad (3)$$

Predicate `reduction` is defined inductively in Coq. Here is the constructor for instruction *call* :

```
red_call :  (∀ M : configuration) (∀ f̄ : frame) (∀ x,y : function)
            (∀ g : name) (∀ ar:nat) (∀ args, l : list value),
            Get_function f = Some  x  ->
            x.fun_bytecode[pc_f] = Some (call x.fun_size g ar) ->
            Get_function g = Some y ->
            split_k_elements ar stack_f = Some (args, l) ->
            (reduction  f̄:: M  (mkfr(g,0,args,args)::mkfr(f,pc_f,l,args_f)::M).
```

Here, *(split_k_elements k l)* returns an optional pair `Some` $(l_k,\ l_r)$, with $l = l_k@l_r$ and $|l_k| = k$ if $|l| \geq k$, *None* otherwise. Condition $pc_f < |f|$ is implicitly expressed by the fact that the pc_f^{th} element of the bytecode of f is of the form (*Some* ...). We can now express that a predicate is *invariant* by reduction :

```
(invariant P):= (∀ M M': configuration), (P M) → (reduction M M') → (P M')
```

Well-formed configurations Predicate `wellformed_configuration` holds on all configurations M that satisfy both conditions **wf1** and **wf2**:
(**wf1**) for each frame $\bar{f}$ in M, *(Get_function* $\bar{f} \neq$ *None)* and $pc_f < |f|$
(**wf2**) for each pair of consecutive frames $(\bar{f}, \bar{h})$ in M, frame $\bar{f}$ has been created by the last evaluated instruction in frame $\bar{h}$, i.e. $bc_h[pc_h]\ = Some\ (call\ |h|\ f\ |args_f|)$

This predicate enjoys the following property, proved by case analysis on the reduction rule applied:

Lemma 1 *Predicate* `wellformed_configuration` *is invariant by reduction.*

Executions are defined by an inductive predicate

```
        execution:  name -> (list value) -> (list configuration) -> Prop
```

such that `(execution f args L)` holds if and only if L is an initial segment of the history of the configurations met when running the program that computes function f on arguments *args*. It is introduced by two constructors:

ex1 : `(∀f: name)(∀args: (list value)), (execution f args [mkfr(f,0,args,args)])`
ex2 : `(∀f: name)(∀args: (list value))(∀ L: (list configuration))`
`(∀ M, M': configuration), (execution f args (M:: L)) -> (reduction M M') ->`
`(execution f args (M':: M:: L))`

It is shown that all properties P preserved through reduction are satisfied by all the configurations of an execution, provided P is true on the initial configuration :

Lemma 2 `(∀P: configuration -> Prop) (∀f: name)(∀args: (list value)),`

```
(P [mkfr(f,0,args,args)]) -> (invariant P) ->
(∀L: (list configuration)), (execution f args L)->
(∀M: configuration), M ∈ L -> (P M)
```

This statement is proved by induction on term `(execution f args L)`.

Results and errors By definition, `(config_result M v)` is satisfied if and only if
– configuration M contains an only frame $\bar{f}$,
– the instruction of rank pc_f in the bytecode of function named f is a *return* instruction,
– value v is the top element of value stack $stack_f$.
Similarly, an erroneous configuration is either the empty configuration, or a configuration on which no reduction can be performed and such that $\forall v : value,\ \neg(config\_result\ M\ v)$.

4.2 Type verification

Type instantiation We instantiate Kildall's generic algorithm in order to obtain a type verification algorithm called $\texttt{Kildall}_\texttt{T}$. This is done inside a Coq section parameterized by a function $\tilde{f}:\ function$. Therefore, in the terms introduced now, parameter $\tilde{f}$ will be either implicit (inside the Coq section) or explicit (when they are referred to outside the Coq section). As usual, bc_f denotes the bytecode of $\tilde{f}$ and $Rt = (fst\ sig_f)$ denotes its return type. Terms of type σ_T may be either abstract stacks encoded as lists of type names, or a special element $\top_T$ that stands for an erroneous state, or $\bot_T$ that reflects the absence of constraints.

```
Inductive σ_T : Set := ⊤_T  :  σ_T | ⊥_T  : σ_T | Types : (list name) -> σ_T.
```

Relation $>_{\sigma_T}$ is the flat order: $\top_T$ is the greatest element, $\bot_T$ is the least element, and all the other elements are incomparable. $\texttt{Kildall}_\texttt{T}$ will be run on the initial function state

$$\texttt{(init}_T\ \tilde{\texttt{f}}\texttt{) := (Types (reverse(snd } sig_f\texttt{)))::[}\bot_\texttt{T}; \ldots ; \bot_\texttt{T}\texttt{]}$$

since, when starting the analysis, the only constraint is to call the function with well typed arguments. The flow function *step* is instantiated by a function $\texttt{Step}_T$ defined by cases on *bc[p]*. For lack of space we only present the case of instruction *return*.

```
(Step_T p C s) := match bc_f[p] with
                  return => match s with
                            ⊥_T => [⊥_T] | ⊤_T => [⊤_T] |
                            (Types l) => match l with
                                         [ ] => [⊤_T]|
                                         Ret::t => if Ret=Rt then [s] else [⊤_T]
```

Predicate *wi* is instantiated by a predicate `Wti` that specifies whether instruction *p* in bytecode *bc* is well-typed with respect to function state *ss*. As for $\texttt{Step}_T$, we only give the case related to the *return* instruction.

```
(Wti ss p C) := match ss[p] with
                ⊤_T => False | ⊥_T => True |
                Types l => match bc_f[p] with
                           return => match l with
                                     Ret::t => if Ret = Rt then True else False |
                                     _ => False
```

It is now mandatory to prove that the hypotheses taken in 3 are fulfilled by these terms. More precisely, we establish that function $Step_T$ is monotone, that *($Step_T$ p C s)* does not depend on certificate *C*, that $(\sigma_T,\ >_{\sigma_T})$ has the expected properties, and lastly that *Wti* coincides with *stable* on all top-free function states. Though some of these proofs are long, none is difficult. They will not be shown here. We have now at our disposal a certified type verifier, program $Kildall_\mathrm{T}$, such that:

$$\top_\mathrm{T} \notin (\texttt{Kildall}_\texttt{T}\,\texttt{ss}) \ \rightarrow\ (\forall \texttt{p} : \texttt{nat})\ (\forall \texttt{C} : \texttt{p} < \texttt{n}),\ (\texttt{Wti}\ (\texttt{Kildall}_\texttt{T}\,\texttt{ss})\ \texttt{p}\ \texttt{C}) \tag{4}$$

$$((\exists\, \texttt{ts} \geq_\texttt{n} \texttt{ss})\ (\forall \texttt{p} : \texttt{nat})\ (\forall \texttt{C} : \texttt{p} < \texttt{n}),\ (\texttt{Wti ts p}))\ \rightarrow \top_\mathrm{T} \notin (\texttt{Kildall}_\texttt{T}\,\texttt{ss}) \tag{5}$$

We can now establish two kinds of results for programs that have passed successfully the type analysis: a well-typedness property on the executions and a progress property.

Well-typed frames Let us assume that every function in the program has passed the type verification, that is:

$(H8)\ \forall \tilde{\texttt{f}} :\ \texttt{function},\ \tilde{\texttt{f}} \in \texttt{functions} \rightarrow\ \top_\mathrm{T} \notin (\texttt{Kildall}_\texttt{T}\ \tilde{\texttt{f}}\ (\texttt{init}_\texttt{T}\ \tilde{\texttt{f}}))$

Then we introduce the notion of well-typed frames. Let $\bar{f}$ be a frame and $\tilde{f}$ the function retrieved in parameter *functions* from the function name appearing in $\bar{f}$. Frame $\bar{f}$ is well-typed if and only if the types of the elements in its value stack actually are those in the abstract stack of index pc_f in the function state computed by $(Kildall_T\ \tilde{f}\ (init_T\ \tilde{f}))$:

welltyped_frame $\bar{f}$:= stack_typing stack$_f$ (Kildall$_T$ $\tilde{\text{f}}$ (init$_\text{T}$ $\tilde{\text{f}}$))[pc$_f$]

Well-typed configurations The notion of well-typedness is extended to configurations. A configuration satisfies predicate welltyped_configuration if and only if:
wt1 : M is a well-formed configuration,
wt2 : the top frame of M is well-typed,
wt3 : for each pair of consecutive frames $(\bar{f},\ \bar{h})$ in M, frame $mkfr(h, pc_h, args_f@stack_h, args_h)$ is well-typed.

As a matter of fact, no frame in a valid configuration, except the top one, is well-typed. They are in an intermediate state, in which the current instruction is a *function call*, but the arguments of the function have been popped out from the stack. The main results are in the following three lemmas. For lack of space we do not detail the proofs.

Lemma 3 *Predicate* welltyped_configuration *is invariant by reduction.*

Lemma 4 *All configurations in all executions are well-typed provided the initial function call occurs on well-typed values.*

Lemma 5 (Progress) (∀M : configuration),(welltyped_configuration M)->
(M = [ ])∨(∃v: value), (config_result M v)∨(∃M': configuration), (reduction M M')

4.3 Shape verification

4.3.1 Shape instantiation

Since shape verification is done by performing a symbolic execution, it handles algebraic *expressions* built from variables, function names and constructor names. We shall consider distinguished expressions called *patterns*, in which no function symbol occurs. Fresh variables created by an instruction (*branch c _ _*) at rank p, with a m-ary constructor c, and a stack of height h, will be $x_{p,h}, \ldots, x_{p,h+m-1}$. Similarly, initial arguments in the symbolic execution of a m-ary function are $x_{0,0}, \ldots, x_{0,m-1}$. Set *name* of constant symbols is extended to a set Name that contains both constants symbols and variables.

```
Inductive Name: Set := x : nat→ nat→ Name | symbol: name → Name
```

Type Expression will be that of trees whose nodes and leaves are marked with elements of *Name*. Elementary substitutions (of type subst_elem) are records made of two variable indices and an expression. We will note $\{x_{i,j} \leftarrow expr\}$ such an elementary substitution. As substitutions are compositions of elementary substitutions, type Substitution is that of lists of elementary substitutions. Various functions are defined to handle expressions and substitutions :

- apply_elem_tree and apply respectively apply an elementary substitution and a substitution to an expression.
- (fresh p h m) returns a forest of single-node trees $x_{p,h}, \ldots, x_{p,h+m-1}$.
- (init_vars $\tilde{\text{f}}$) is the list of expressions $[x_{0,m-1}; \ldots; x_{0,0}]$ with $m = |snd\ sig_f|$
- init_subst:nat->nat->(list value)->(Opt Substitution). Term (init_subst h m args) is the substitution that matches variables $x_{0,h},\ \ldots,\ x_{0,h+m-1}$ with *args*, that is the list $[\{x_{0,h+m-1} \leftarrow args[0]\}; \ldots; \{x_{0,h} \leftarrow args[m-1]\}]$. It equals *None* if $|args| \neq m$.

- `tree_is_pattern: Expression -> Prop` indicates whether its argument is a pattern or not.
- `make_substitution: (forest Name)->(forest name)->(Opt Substitution)` matches a forest of expressions with a forest of values.

As in section 4.2, all terms introduced in the sequel of this section are implicitly parameterized by a given function $\tilde{\mathtt{f}}$. Kildall's algorithm is particularized, resulting in algorithm $\mathtt{Kildall}_S$. The new state type σ_S is defined by:

Inductive σ_S:Set:= $\top_S : \sigma_S$ | $\bot_S : \sigma_S$ | Shapes: Substitution->(list Expression)->σ_S.

As for types, relation $>_{\sigma_S}$ is the flat order over σ_S. $\mathtt{Kildall}_S$ will be run on initial function state

$$\texttt{(init\_S } \tilde{\texttt{f}}\texttt{) := (Shapes [ ] (init\_vars } \tilde{\texttt{f}}\texttt{))::}[\bot_S; \ldots; \bot_S]$$

Let us now describe function $\mathtt{Step}_S$ when instruction of index `p` is a *branch* instruction, which is the most interesting case. The formal semantics of instruction *branch* is defined by two rules:

$$\frac{pc_f < |f| \qquad bc_f[pc_f] = (\texttt{branch}\ c\ \_\ \_) \quad stack_f = c(a_1, \ldots, a_m) :: l}{(f,\ pc_f,\ stack_f,\ args_f) :: M \rightarrow (f,\ pc_f + 1,\ [a_m; \ldots; a_1]@l,\ args_f) :: M} \tag{6}$$

$$\frac{pc_f < |f| \qquad bc_f[pc_f] = (\texttt{branch}\ c\ j\ \_) \quad stack_f = d(\ldots) :: l \quad c \neq d}{(f,\ pc_f,\ stack_f,\ args_f) :: M \rightarrow (f,\ j,\ stack_f,\ args_f) :: M} \tag{7}$$

In the definition below, given in a simplified form, `c` and `d` are constructor names and x is a variable.

```
(Step_S p C s) =  match bc_f[p] with (branch c j _) =>
match s with  (Shapes S l) =>
  match l with
    d(e1, ...,em)::l' => if c = d then
                           [(Shapes S [em; ...;e1]@l'); ⊥_S]
                          else
                           [⊥_S; s] |
    x::l'             => if ‘‘c is a constructor name’’ then
                          let (m = arity c) in
                           let vars = (fresh p |l| m) in
                            let subst = {x ← c(vars)} in
                             let l'' = (map (apply_elem_subst subst) l') in
                               [(Shapes (subst::S) (reverse vars)@l'' ; s]
                          else [⊤_S;  ⊤_S] |
    _                 => [⊤_S;  ⊤_S]
```

Lastly, parameter `wi` is instantiated by `Wshi`. We describe below the part of its definition related to instruction *branch*. One can observe that the definitions of $Step_S$ and $Wshi$ are much alike.

```
(Wshi ss p _) = match ss[p]  with (Shapes S l) =>
match bc_f[p] with (branch c j _) =>
  match l with
    d(e1, ...,em)::l' => if c = d then ss[p+1] = (Shapes S [em;...;e1]@l')
                          else ss[j] = ss[p] |
    x::l'             => if ‘‘c is a constructor name’’ then
                           let (m = arity c) in
                            let vars = (fresh p |l| m) in
                             let subst = {x ← c(vars)} in
```

```
                              let l'' = (map (apply_elem_subst subst) l') in
                                ss[p+1]= (Shapes subst::S (reverse vars)@l'')
                                ∧ ss[j] =ss[p]
                           else False |
_                      => False
```

From section 3.4, we deduce the following two results:

$$\top_{\mathtt{S}} \notin (\mathtt{Kildall_S\ ss}) \ \rightarrow\ (\forall \mathtt{p} : \mathtt{nat})(\forall \mathtt{C} : \mathtt{p} < \mathtt{n}),\ (\mathtt{Wshi}\ (\mathtt{Kildall_S\ ss})\ \mathtt{p\ C}) \tag{8}$$

$$(\exists\, \mathtt{ts} \geq_{\mathtt{n}} \mathtt{ss}\ (\forall \mathtt{p} : \mathtt{nat})(\forall \mathtt{C} : \mathtt{p} < \mathtt{n}),\ (\mathtt{Wshi\ ts\ p\ C})) \ \rightarrow \top_{\mathtt{S}} \notin (\mathtt{Kildall_S\ ss}) \tag{9}$$

Well-shaped frames Let us assume that every function in the program has passed both the type and shape verifications, that is:

$(H9)$: $(\forall \tilde{\mathtt{f}} : \mathtt{function}), (\tilde{\mathtt{f}} \in \mathtt{functions}) \ \rightarrow\ \top_{\mathtt{S}} \notin (\mathtt{Kildall_S}\ \tilde{\mathtt{f}}\ (\mathtt{init_S}\ \tilde{\mathtt{f}}))$

We introduce the notion of well-shaped frames. Let $\bar{f}$ be a frame and $\tilde{f}$ be the function retrieved in parameter *functions* from the function name appearing in $\bar{f}$. Frame $\bar{f}$ is well-shaped if and only if the elements in its value stack actually match the expressions in the symbolic stack of index pc_f in the function state computed by $(\mathtt{Kildall}_S\ \tilde{\mathtt{f}}\ (\mathtt{init}_S\ \tilde{\mathtt{f}}))$. More precisely, assuming that $(\mathtt{Kildall}_S\ \tilde{\mathtt{f}}\ (\mathtt{init}_S\ \tilde{\mathtt{f}}))[\mathtt{pc_f}]$ = (Shapes S l), there exists a substitution ρ such that :

- ρ results from the matching of the actual arguments $args_f$ in the function call and *(map (apply S) (init_vars* $\tilde{f}$*))*.
- l and $stack_f$ are same length.
- For each pattern p_j of rank j in the symbolic stack l, if v_j denotes the value of same rank in the value stack $stack_f$ then $v_j = (apply\ \rho\ p_j)$.

Well-shaped configurations Similarly to what has been done for type verification, the notion of well-shapedness is extended to configurations. Configuration M satisfies predicate `wellshaped_configuration` if and only if:
wsh1 : M is a well-typed configuration,
wsh2 : the top frame of M is well-shaped,
wsh3 : for each pair of consecutive frames $\bar{f}\ \bar{h}$ in M, frame $(h, pc_h, args_f@stack_h, args_h)$ is well-shaped.

We establish the following two lemmas that are the analogous of lemmas 3 and 4 in section 4.2.

Lemma 6 *Predicate* `wellshaped_configuration` *is invariant by reduction.*

Lemma 7 *All configurations in all executions are well-shaped provided the initial function call occurs on well-typed values.*

Lemma 7 is a straightforward corollary of lemma 6. But proof of lemma 6 is quite long. It is performed by case analysis on the reduction rule that is applied. We detail the case of rule 6 in appendix A.

Let us mention that, as an instruction *branch* performs a pattern matching on the top of the current stack, the symbolic analysis only makes sense if the top of this stack is a pattern. This condition is not too restrictive and is fulfilled by all the bytecode programs produced by our compiler. In case of untrusted bytecode, it can be easily checked by scanning the function state computed by $(Kildall_S\ \tilde{f}\ (init_S\ \tilde{f}))$. Consequently, we will assume that:

$(H10)$ *For all function* $\tilde{f}$ *in the program that passed the shape analysis, the expression on the top of the symbolic stack on which a branch instruction is performed is a pattern.*

5 Conclusion

In other work, such as those cited in the introduction, the java bytecode verifier has been described as an instance of a generic data flow analyser. Here, we really make use of such a generic approach since we apply it to two distinct kinds of static analyses, and we intend to extend it to a third one. All of them are part of the same system designed to ensure bounds of the memory used when executing a program.
Let us point out that Kildall's data flow analyser is much more powerful than needed for our language, since it accomodates any state space which is a well-founded lattice (this is the case of the type lattice for languages with subtyping). In this paper, we only deal with flat lattices and the functions' flow graphs are supposed to be trees. We intend in the near future to allow bytecode optimisations, code sharing, and to enrich the language with object features. This is why we have treated carefully the generic part of this work. Moreover, such a functional specification of the broadly used Kildall's algorithm is of general interest. In this field, the work closest to ours is that of Klein and Nipkow's [Nip01, KN03], and a detailed comparison is given in section 3. Let us also mention the work of Barthe and al. [BDJMdS02] where an "abstract virtual machine" for the JavaCard language is specified in the Coq Proof Assistant.
The whole Coq development consists of 14000 lines among which 1900 are related to the generic Kildall's algorithm, 2500 to type analysis, 5500 to shape analysis, and 2600 to dependent lists. This shows in particular that the second verification is much more complex than the first one. Coq files will be available at the URL: *http:://www.cmi.univ-mrs.fr/∼solange.*

References

[ACGDZJ04] Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal-Zilio, and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, pages 265–279, 2004.

[BB02] Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.

[BDJ$^+$01] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard P. Serpette, and Simão Melo de Sousa. A formal executable semantics of the javacard platform. In *ESOP*, pages 302–319, 2001.

[BDJMdS02] Gilles Barthe, Guillaume Dufay, Line Jakubiec, and Simão Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 2294 of *Lecture Notes in Computer Science*. Springer, 2002.

[Gol98] Allen Goldberg. A specification of java loading and bytecode verification. In *ACM Conference on Computer and Communications Security*, pages 49–58, 1998.

[Kil73] Gary Arlen Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[Mar00] Jean-Yves Marion. *Complexité implicite des calculs de la théorie à la pratique.* PhD thesis, Habilitation à diriger des recherches, Université Nancy 2, December 2000.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In M. Miculan F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *LNCS*, pages 347–363. Springer-Verlag, 2001.

A Proof of lemma 6

Let $\tilde{f}$ be a function having passed the shape analysis and let us define $ss = (Kildall_S\ \tilde{f}\ (init_S\ \tilde{f}))$. From hypothesis $(H9)$ (section 4.3), we know that ss does not contain $\top_S$, and from proposition (8) (also in in section 4.3) we deduce that

$$\forall p < |f|, (Wshi\ ss\ p) \tag{10}$$

Let M and M' be two configurations such that *(reduction M M')* holds. Assuming that M is well-shaped, we have to prove that M' is well-shaped. We proceed by case analysis on the rule applied in the derivation of *(reduction M M')*. We will focus on the sole rule reduction (6). We know that:
$M = \bar{f} :: M_0$, $stack_f = c(a_1, \ldots, a_m) :: l$ and $M' = (f, pc_f + 1, [a_m;\ \ldots;\ a_1]@l,\ args_f) :: M_0$.
Moreover, $bc_f[pc_f] = (branch\ c\ \_\ \_)$. $\bar{f}$ being well-shaped, $ss[pc_f] = (Shapes\ S\ (e :: L))$ and there exists a substitution ρ that matches $args_f$ and $(map\ (apply\ S)(init\_vars\ \tilde{f}))$, that is :

$$(map\ (apply\ \rho)\ (map\ (apply\ S)\ (init\_vars\ \tilde{f}))) \ =\ \overline{args_f}\ ^1 \tag{11}$$

Moreover, we also have :

$$|e :: L| = |stack_f| \tag{12}$$

Lastly, for each pattern p_j of rank j in the symbolic stack $e :: L$, if v_j denotes the value of same rank in the value stack $stack_f$ then

$$v_j \ =\ (apply\ \rho\ p_j). \tag{13}$$

In particular, since e is a pattern from $(H10)$ (see section 4.3), we have $(apply\ \rho\ e)\ =\ c(a_1,\ \ldots\ a_m)$, and thus, we deduce that:

$$(e\ is\ a\ variable)\ \vee\ (e\ =\ c(e_1, \ldots, e_m)\ \wedge\ \forall i \in \{1, \ldots, m\}\ (apply\ \rho\ e_i)\ =\ a_i) \tag{14}$$

From lemma 3, M' satisfies **wsh1**. It can be shown without difficulty that **wsh3** is also satisfied by M'. Let us consider condition **wsh2**. We have thus to prove that frame $(f, pc_f + 1, [a_m;\ \ldots;\ a_1]@l,\ args_f)$ is well-shaped, that is:

(a) $ss[pc_f + 1] = (Shapes\ S'\ L')$

(b) $|L'|\ =\ |[a_m;\ \ldots;\ a_1]@l|$

(c) there exists a substitution ρ' such that
$$(map\ (apply\ \rho')\ (map\ (apply\ S')\ (init\_vars\ \tilde{f})))\ =\ \overline{args_f}.$$

(d) for each pattern p_j of rank j in the symbolic stack L', if v_j denotes the value of same rank in the value stack $[a_m;\ \ldots;\ a_1]@l$, then $v_j\ =\ (apply\ \rho'\ p_j)$

From (10) we know that $(Wshi\ ss\ pc_f)$. From (14) and from the definition of $Wshi$, we have:

– **either** $e\ =\ c(e_1, \ldots, e_m)$, then $ss[pc_f + 1]\ =\ (Shapes\ S\ [e_m; \ldots; e_1]@L)$. In this case, conditions **(a)** and **(b)** are immediately satisfied. Conditions **(c)** and **(d)** are fulfilled by choosing $\rho' = \rho$.

– **or** e **is a variable**. Let $e = x$. In this case,

(i) $ss[pc_f + 1] = (Shapes\ \ subst :: S\ \ (reverse\ vars)@SL)$ where:

(ii) $vars = (fresh\ pc_f\ |e :: L|\ m)$. This call to function $fresh$ generates m fresh variables in the way described in section 4.3. For readability, in this proof we denote them $x_1, \ldots, x_m$, and we assume that they are actually fresh. The freshness of the generated variables is formally proved in Coq in a non trivial lemma.

[1] Here, $\overline{args}$ is the conversion of a list of values into a list of expressions. This is just syntax, since values can be injected in expressions, and not relevant for the proof.

(iii) $\mathit{subst} = \{x \leftarrow c(x_1, \ldots, x_m)\}$

(iv) $SL = (\mathit{map}\ (\mathit{apply\_elem\_subst}\ \mathit{subst})\ L)$

Again, conditions **(a)** and **(b)** are trivially satisfied.
Let $\Sigma = [\{x_1 \leftarrow a_1\}; \ldots\ \{x_m \leftarrow a_m\}]$ and $\rho' = \rho@\Sigma$. Establishing condition **(c)** amounts to prove, from (11), that for all variables y in $(\mathit{init\_vars}\ \tilde{f})$:

$$(\mathit{apply}\ \rho@\Sigma@(\mathit{subst} :: S)\ y)\ =\ (\mathit{apply}\ \rho@S\ y)$$

Let us pose $(\mathit{apply}\ S\ y)\ =\ \mathit{expr}[x, y_1, \ldots, y_k]$ where expr is a context and x, $y_1, \ldots, y_k$ are the variables occurring in the expression. By definition,

$$(\mathit{apply}\ (\mathit{subst} :: S)\ y)\ =\ \mathit{expr}[c(x_1, \ldots, x_m), y_1, \ldots, y_k].$$

Since $x_1, \ldots, x_m$ are fresh, we can deduce the following two equalities, that are proved by using several Coq auxiliary lemmas:
$(\mathit{apply}\ \Sigma@(\mathit{subst} :: S)\ y)\ =\ \mathit{expr}[c(a_1, \ldots, a_m), y_1, \ldots, y_k]$
$(\mathit{apply}\ \rho@\Sigma@(\mathit{subst} :: S)\ y)\ =\ \mathit{expr}[c(a_1, \ldots, a_m), (\mathit{apply}\ \rho\ y_1), \ldots, (\mathit{apply}\ \rho\ y_k)]\ =$
$\quad \mathit{expr}[(\mathit{apply}\ \rho\ x), (\mathit{apply}\ \rho\ y_1), \ldots, (\mathit{apply}\ \rho\ y_k)] = (\mathit{apply}\ \rho\ \mathit{expr}[x, y_1, \ldots, y_k]\ =\ (\mathit{apply}\ \rho@S\ y)$

Let us now consider condition **(d)**. By hypothesis (13), substitution ρ matches all patterns in $x :: L$ with the value of same rank in $c(a_1, \ldots, a_m) :: l$. We have to prove the similar property for substitution $\rho@\Sigma$ and stacks $[x_m; \ldots; x_1]@L$ and $[a_m; \ldots; a_1]@l$. We can easily conclude from the fact that variables $x_1, \ldots, x_m$ are fresh.

The proof of the whole lemma in Coq takes approximatively 1,000 lines. It uses a lemma concerning function fresh whose proof takes 2,000 lines.