Short Presentation: A Functional Scenario for Bytecode Verification of Space Bounds

Roberto Amadio, Solange Coupet-Grimal, Silvano Dal Zilio and Line Jakubiec

LIF, Laboratoire d'Informatique Fondamentale de Marseille
CNRS and Université de Provence [*]

Abstract

We define a simple stack machine for a first-order functional language and show how to perform type, size, and termination verifications at the level of the bytecode of the machine. In particular, we show that a combination of size verification based on quasi-interpretations and of termination verification based on lexicographic path orders leads to an explicit polynomial bound on the space required for the execution.

1 Introduction

Research on mobile code has been a hot topic since the late 90's with many proposals building, for instance, on the JAVA platforms. Security issues are one of the fundamental problems that still have to be solved before mobile code can become a well-established and well-accepted technology. Application scenarios may include, for instance, programmable switches, network games, and applications for smart cards.

Initial proposals have focused on the integrity of the execution environment, like the absence of memory faults. In this paper, we consider an additional property of interest to guarantee the safety of a mobile code, that is, ensuring bounds on the (computational) resources needed for the execution of the code. We most particularly study the *space* needed for the execution of a program.

We define a simple stack machine for a first-order functional language and show how to perform type, size, and termination verifications at the level of the bytecode of the machine. These verifications rely on certifiable annotations of the bytecode—we follow here the classical viewpoint that a program may originate from a malicious party and does not necessarily result from the compilation of a well-formed program. Finally, we show that we can extract from a combination of size and termination verifications a polynomials expression that, given the size of the initial parameters, bounds the space required for the execution of a program.

[*]This work was partially supported by the ACI CRISS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi t or commercial advantage and that copies bear this notice and the full citation on the fi rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi c permission and/or a fee.
SPACE'04, January 12, 2004, Venice, Italy.

The problem of bounding the usage made by programs of their resources has already attracted considerable attention. Automatic extraction of resource bounds has mainly focused on (first-order) functional languages starting from Cobham's characterization of polynomial time functions by bounded recursion on notation. Here we explore various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms.

We consider a rather standard first-order functional programming language with inductive types, pattern matching, and call-by value, that can be regarded as a fragment of various ML dialects. The language is also quite close to term rewriting systems (TRS) with constructor symbols. The language comes with three main varieties of static analyses: (i) a standard type analysis, (ii) an analysis of the size of the computed values based on the notion of *quasi-interpretation*, and (iii) an analysis that ensures termination; among the many available techniques we select here recursive path orderings.

The last two analyses, and in particular their combination, are instrumental to the prediction of the space and time required for the execution of a program as a function of the size of the input data. For instance, it is known [2] that a program admitting a polynomially bound quasi-interpretation and terminating by lexicographic path-ordering runs in polynomial space. This and other results [6, 7] can be regarded as generalizations and variations over Cobham's characterization of polynomial time by bounded recursion on notation [4].

The synthesis of termination orderings is a classical topic in term rewriting (see for instance [3]). The synthesis of quasi-interpretations—a concept introduced by Marion *et al.* [8]—is connected to the synthesis of polynomial interpretations for termination but it is generally easier because inequalities do not need to be strict and small degree polynomials are often enough [1]. We will not address synthesis issues in this paper. We suppose that the bytecode comes with annotations such as types and polynomial interpretations of function symbols and orders on function symbols.

Our main goal is to determine how these annotations have to be *formulated and verified* in order to entail size bounds and termination at *bytecode* level, *i.e.*, at the level of an assembler-like code produced by a compiler and executable on a simple stack machine. We carry on this program up to the point where it is possible to verify that a given bytecode will run in polynomial space thus providing a translation of the result mentioned above at byte code level. Beyond proving that the program "implements a PSPACE function," we extract a polynomials that bounds the size needed to run a program: assume f is a function (identifier) of arity n in a verified program, then we obtain a polynomial $q(x_1,\ldots,x_n)$ such that for all values

$v_1,\ldots,v_n$ of the appropriate types, the size needed for the evaluation of the call $f(v_1,\ldots,v_n)$ is bounded by $q(|v_1|,\ldots,|v_n|)$, where $|v|$ stands for the size of the value v.

A secondary goal is of a pedagogical nature: present a minimal but still relevant scenario in which problems connected to bytecode verification can be effectively discussed. For example, our functional virtual machine is based on a set of 6 instructions, a number that has to be compared with the almost 200 opcodes used in the JAVA virtual machine.

The interest of carrying on such analyses at bytecode level are now well understood [10, 11]. First, mobile code is shipped around in pre-compiled form (*i.e.*, bytecode) and needs to be analyzed as such. Second, compilation is an error prone process and therefore it seems safer to perform static analyses at the level of the bytecode rather than at source level. In particular it allows to reduce the size of the trusted code: we only have to trust the analyzer, not the whole compilation chain.

Most work in the literature on bytecode verification tends to guarantee the integrity of the execution environment. Related work is carried on in the MRG project [12]. The main technical differences appear to be as follows: (i) they rely on a general proof carrying code approach while we are closer to a typed assembly language approach and (ii) their analyses focus on the size of the heap while we also consider the size of the stack and the termination of the execution. Another related work is due to Marion and Moyen [9] who perform a resource analysis of counter machines by reduction to a certain type of termination in Petri Nets. Their virtual machine is much more restricted than the one we study here as natural numbers is the only data type and the stack can only contain return addresses.

The paper is organized as follows. Section 2 sketches a first-order functional language with simple types and call-by-value evaluation and recalls some basic facts about quasi-interpretations and termination. Section 3 describes a simple virtual machine comprising a minimal set of 6 instructions that suffice to compile the language described in the previous section. We also define a type verification analysis that guarantees that all values on the stack will be well typed. This verification assumes that constructors and function symbols in the bytecode are annotated with their type. In the following sections, we also assume that they are annotated with suitable functions to bound the size of the values on the stack (section 5) and with an order to guarantee termination (section 6). The size and termination verification depend on a path verification which is described in section 4. We provide an example of type, size and termination analyses in Section **??**.

We conclude with a combination of the size and termination verifications that guarantees that the bytecode runs in polynomial space. The presentation of each verification follows a common pattern: (i) definition of constraints on the bytecode and (ii) definition of a predicate which is invariant under machine reduction. The essential technical difficulty is in the structuring of the constraints and the invariants, the proofs are then routine inductive arguments which we delay to the appendix.

2 A Functional Language

We consider a very simple typed first-order functional language, with recursive abstract data-types and pattern-matching.

A *program*, *prog*, is composed of a list of mutually recursive type definitions followed by a list of mutually recursive first-order function definitions relying on pattern matching. Expressions and values are built from a finite number of constructors, ranged over by $\mathsf{c},\mathsf{c}_1,\ldots$. We use $f,f',\ldots$ to range over function identifiers and $x,x',\ldots$ for (first-order) variables and distinguish the following three syntactic categories:

$$\begin{array}{ll} v ::= \mathsf{c}(v,\ldots,v) & \text{(values)} \\ p ::= x \mid \mathsf{c}(p,\ldots,p) & \text{(patterns)} \\ e ::= x \mid \mathsf{c}(e,\ldots,e) \mid f(e,\ldots,e) & \text{(expressions).} \end{array}$$

A function is defined by a sequence of pattern-matching *rules* of the form $f(p_1,\ldots,p_n) = e$, where e is an expression. We follow the usual hypothesis that the patterns $p_1,\ldots,p_n$ are linear (a variable appears at most once) and do not superpose.

We use $t,t_1,\ldots$ to range over type identifiers. A type definition has the shape

$$t \;=\; \mathsf{c}_1 \textit{ of } t^1_1 * \cdots * t^1_{n_1} \mid \cdots \mid \mathsf{c}_k \textit{ of } t^k_1 * \cdots * t^k_{n_k}$$

For instance, we can define the type *nat* of natural numbers in unary format: $nat = \mathsf{z} \mid \mathsf{s}\ of\ nat$.

In the following, we consider that constructors are declared with their functional type $(t^i_1,\ldots,t^i_{n_i}) \to t$. Similar types can be either assigned or inferred for the function symbols. (The typing rules are standard and are omitted.)

The following table define the standard call-by-value evaluation relation, where σ is a substitution from variables to values.

Evaluation: $e \Downarrow v$

$$\frac{e_j \Downarrow v_j \quad \text{for all } j \in 1..n}{\mathsf{c}(e_1,\ldots,e_n) \Downarrow \mathsf{c}(v_1\ldots,v_n)}$$

$$\frac{\begin{array}{c} e_j \Downarrow v_j \quad \sigma p_j = v_j \quad \text{for all } j \in 1..n \\ f(p_1,\ldots,p_n) = e \in prog \quad \sigma(e) \Downarrow v \end{array}}{f(e_1,\ldots,e_n) \Downarrow v}$$

If e is an expression then $Var(e)$ is the set of variables occurring in it. The *size* of an expression $|e|$ is defined as 0 if e is a constant or a variable and $1 + \Sigma_{i\in 1..n}|e_i|$ if e is of the form $\mathsf{c}(e_1,\ldots,e_n)$ or $f(e_1,\ldots,e_n)$.

EXAMPLE 1. *The function add* : $(nat, nat) \to nat$*, defined by the following two rules, computes the sum of two values of type nat.*

$$\begin{array}{llcll} add & \mathsf{z} & y & = & y \\ add & \mathsf{s}(x) & y & = & add\ x\ \mathsf{s}(y) \end{array}$$

For instance, we have: add $\mathsf{s}(\mathsf{s}(\mathsf{z}))\ \mathsf{s}(\mathsf{z}) \Downarrow \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$.

2.1 Quasi-interpretations

To bound the size of the values computed by a function, the idea is to associate to every expression e with variables $x_1,\ldots,x_n$, a rational function, q_e, such that $q_e(|v_1|,\ldots,|v_n|)$ bounds the size of $e[v_1,\ldots,v_n/x_1,\ldots,x_n]$.

A quasi-interpretation assigns to constructors and function symbols functions over the non-negative rationals $\mathbf{Q}^+$ such that:

- if c is a constant then $q_\mathsf{c} = 0$,
- if c is a constructor with arity n then $q_\mathsf{c} = d + \Sigma_{i\in 1..n} x_i$, where $d \geqslant 1$,
- $q_f : (\mathbf{Q}^+)^n \to \mathbf{Q}^+$ is monotonic and for all $i \in 1..n$ we have $q_f(x_1,\ldots,x_n) \geqslant x_i$.

The assignment may be straightforwardly extended to all expressions as follows: $q_x = x$; $q_{\mathsf{c}(e_1,\ldots,e_n)} = q_\mathsf{c}(q_{e_1},\ldots,q_{e_n})$; and $q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1},\ldots,q_{e_n})$.

Moreover, it is required that for all the rules $f(p_1,\ldots,p_n) = e$ in the program, the inequality $q_{f(p_1,\ldots,p_n)} \geqslant q_e$ holds.

In general, a quasi-interpretation provides a bound on the size of the computed values as a function of the size of the input data. If $f(v_1,\ldots,v_n) \Downarrow v$ then $|v| \leqslant q_v \leqslant q_f(|v_1|,\ldots,|v_n|)$.

An interesting space for the synthesis of quasi-interpretations is the collection of max-plus polynomials [1], that is, functions equivalent to an expression of the form $\max_{i\in I}(\Sigma_{j\in 1..n} a_{i,j} x_j + a_i)$, with $a_{i,j} \in \mathbf{N}$ and $a_i \in \mathbf{Q}^+$.

EXAMPLE 2. *Assume we choose $q_{\mathsf{s}} = 1 + x$ for the quasi-interpretation of the constructor in nat (by definition, we have that $q_{\mathsf{z}} = 0$). The polynomial $q_{add}(x,y) = x + y$ is a valid quasi-interpretation for the function add defined in Example 1: we have $q_{add}(0,y) \geqslant y$ and $q_{add}(1+x,y) \geqslant q_{add}(x,1+y)$.*

2.2 Termination

Programs can be regarded as a set of term rewriting rules, just associate to every rule $f(p_1,\ldots,p_n) = e$ the *term rewriting rule* $f(p_1,\ldots,p_n) \to e$. Hence termination methods developed for term rewriting systems apply. In particular, under the hypothesis that the rules are orthogonal, termination of the TRS is equivalent to the termination of the call-by-value evaluation strategy.

3 Virtual Machine

We define a simple set of byte-code instructions, and a related stack machine, for the compilation and the evaluation of programs. For the sake of simplicity, we suppose that the code and type information of every function handled by the virtual machine is fixed and known in advance. Hence, we consider a fixed set of constructor and disjoint function names.

We use the notation $f : (t_1,\ldots,t_n) \to t$ to refer to the signature of the function associated to the identifier f and $ar(f)$ for the arity of f. We use similar notations for constructors. For the sake of simplicity, we equate a function identifier, f, with the sequence of instructions of its body code, We also use the notation $f[i]$ for the ith instruction in the (compiled) code of f and $|f|$ for the size of f (number of instructions). We adopt the usual notation on words: ε is the empty sequence, $h \cdot h'$ is the concatenation of two sequences h, h', and $|h|$ is the length of a sequence h.

The virtual machine is built around three components: (1) a *memory*, M, which is a stack of call frames; (2) an association list between function identifier and code; (3) a bytecode interpreter, modeled as a reduction relation $M \to M'$. The state of the interpreter, the *configuration* M, is either a value or a sequence of frames. A frame is a triple (f, pc, h) made of a function identifier, the value of the program counter (a natural number in $1..|f|$) and an evaluation *stack*. A stack is a list of values that serves both to store the parameters and the values computed during the execution of (a function call in) the code.

We work with a minimal set of instructions whose effect on the configuration is described below and write $M \to M'$ if the configuration M reduces to M' by applying exactly one of the transformations. We start with an informal explanation of the semantics of the virtual machine instructions.

The instruction `load` $(i : int)$ takes as parameter a valid indices in the current stack. Upon execution, the stack, is updated by pushing a copy of the ith value in h to the top. New values may also be created using the instruction `build` $(ident : string)$ $(n : int)$, which takes an identifier that corresponds to a constructor with arity n. When executed, the current stack, h, is updated as follows: the n values, $v_1,\ldots,v_n$, on top of h are discarded and replaced by the single value $ident(v_1,\ldots,v_n)$.

A function call is implemented by the instruction `call` $(ident : string)$ $(n : int)$, with first parameter an identifier corresponding to a function with arity n. Upon execution, a fresh call frame is created, which is initialized with a copy of the n values on top of the caller's stack. The lifetime of the current frame is controlled by two instructions, `return` and `stop`. The first instruction discard the current frame and returns the value at the head of the stack to the caller (i.e., the previous frame in the memory). The second instruction stops the virtual machine and returns with an error status.

The last instruction, `branch` $(ident : string)$ $(block : int)$, implements a conditional jump on the shape of the value, v, found on top of the current stack. If v is of the form $ident(v_1,\ldots,v_n)$, then the top of the current execution stack is discarded and replaced by the n sub-values $v_1,\ldots,v_n$. Otherwise, the stack is left unchanged and the execution jump to position $block$ in the code (with $block \in 1..|f|$).

Bytecode Interpreter

(Load)
$$\frac{f[pc] = \mathtt{load}\ i \qquad pc < |f| \qquad h[i] = v}{M \cdot (f, pc, h) \to M \cdot (f, pc+1, h \cdot v)}$$

(Build)
$$\frac{\begin{array}{c} f[pc] = \mathtt{build}\ c\ n \qquad pc < |f| \\ c : (t_1,\ldots,t_n) \to t_0 \qquad h = h' \cdot (v_1,\ldots,v_n) \end{array}}{M \cdot (f, pc, h) \to M \cdot (f, pc+1, h' \cdot c(v_1,\ldots,v_n))}$$

(Call)
$$\frac{\begin{array}{c} f[pc] = \mathtt{call}\ g\ n \qquad pc < |f| \\ g : (t_1,\ldots,t_n) \to t_0 \qquad h = h' \cdot (v_1,\ldots,v_n) \end{array}}{M \cdot (f, pc, h) \to M \cdot (f, pc, h) \cdot (g, 1, (v_1,\ldots,v_n))}$$

(Return)
$$\frac{\begin{array}{c} f[pc] = \mathtt{return} \qquad f : (t_1,\ldots,t_n) \to t_0 \\ top(h) = v_0 \qquad h' = h'' \cdot (v_1,\ldots,v_n) \end{array}}{M \cdot (g, pc', h') \cdot (f, pc, h) \to M \cdot (g, pc'+1, h'' \cdot v_0)}$$

(Stop)
$$\frac{f[pc] = \mathtt{stop}}{M \cdot (f, pc, h) \to error}$$

(BranchThen)
$$\frac{f[pc] = \mathtt{branch}\ c\ j \quad pc < |f| \quad h = h' \cdot c(v_1,\ldots,v_n)}{M \cdot (f, pc, h) \to M \cdot (f, pc+1, h' \cdot (v_1,\ldots,v_n))}$$

(BranchElse)
$$\frac{f[pc] = \mathtt{branch}\ c\ j \quad 1 \leqslant j \leqslant |f| \quad h = h' \cdot c'(\ldots) \quad c \neq c'}{M \cdot (f, pc, h) \to M \cdot (f, j, h)}$$

The reduction $M \to M'$ is deterministic. There is a special state of the memory, denoted *error*, corresponding to an empty configuration of frames. This state cannot be accessed during a computation that do not raise an error (execute the instruction `stop`). A "good" execution starts with a memory containing one frame, $(f, 1, (v_1,\ldots,v_n))$, corresponding to the evaluation of the expression $f(v_1,\ldots,v_n)$, and ends with a memory of the form $(f, pc, h \cdot v_0)$ where $1 \leqslant pc \leqslant |f|$ and $f[pc] = \mathtt{return}$. In this case the result of the evaluation is v_0.

3.1 Compilation

The language described in section 2 admits a direct compilation in the bytecode described above. Every function is compiled into a segment of instructions and linear pattern matching is compiled into a nesting of `branch` instructions. Finally, variables are replaced by offsets from the base of the stack frame.

Clearly, a realistic implementation of the virtual machine should at least include: (i) a mechanism to execute efficiently tail recursive calls (when a `call` instruction is immediately followed by `return`), and (ii) a mechanism to share common sub-values in a configuration. For instance, one could keep a stack of pointers to values which are allocated on a heap. Various policies could then be considered to garbage collect the heap.

EXAMPLE 3. *We give the result of the compilation of the function add, see Definition 1.*

```
1 : load 1
2 : branch s 7
3 : load 2
4 : build s
5 : call add 2
6 : return
7 : load 2
8 : return
```

3.2 Bytecode Verification

We define a simple analysis used to ensure the well-formedness and well-typedness of the machine configuration during execution. This analysis is the equivalent of *bytecode verification* in the JAVA machine for our "functional" virtual machine, and may be directly used as the basis of an algorithm for validating the bytecode before its execution by the interpreter.

The idea is to associate to every instruction (every step in the evaluation of a function code) an abstraction of the stack before it is executed. In our case, an abstract state is a list of types, or *type stack*, $T = (t_1, \ldots, t_n)$, that should exactly match the types of the values present in the stack at the time of the execution. Accordingly, we define an *abstract execution* for a function f, denoted F, as a list of length $|f|$ of type stacks.

We are interested by abstract states that are coherent with respect to the evaluation of the instructions in f. To express that an abstract execution F is coherent with f, we define the notion of valid *data-flow analysis* based on the auxiliary relation, $dfa_i(f,F)$. Informally, we have $dfa_i(f,F)$ if $F[i] = [t'_1, \ldots, t'_p]$ and during every valid evaluation of f, the stack at the time of the evaluation of $f[i]$ is of the form $[v_1, \ldots, v_p]$ where v_i is a value of type t'_i for every $i \in 1..p$.

As a side result of the bytecode verification we obtain, for every instruction, the size of the stack at the time of its execution. Coupled with a bound on the size of every value appearing in a stack (from the value size verification) and a bound on the maximal number of call frame (from the termination verification) this result is instrumental in the computation of a bound on the total space needed by an execution of the machine.

The definition of the relation $dfa_i(f,F)$ is by case analysis on the instruction $f[i]$. We have $dfa_i(f,F)$ if and only if we have $i \leqslant |f|$ and $i \leqslant |F|$ and:

$f[i] = \texttt{load}\ j$ implies $i < |F|$, $F[i] = T$, $T[j] = t_j$ and $F[i+1] = T \cdot t_j$,

$f[i] = \texttt{build}\ c\ n$ implies $c : (t_1, \ldots, t_n) \to t_0$, $i < |F|$, $F[i] = T \cdot (t_1, \ldots, t_n)$ and $F[i+1] = T \cdot t_0$,

$f[i] = \texttt{call}\ g\ n$ implies $g : (t_1, \ldots, t_n) \to t_0$, $i < |F|$, $F[i] = T \cdot (t_1, \ldots, t_n)$ and $F[i+1] = T \cdot t_0$,

$f[i] = \texttt{return}$ implies $f : (t_1, \ldots, t_n) \to t_0$ and $F[i] = T \cdot t_0$,

$f[i] = \texttt{stop}$ is always true,

$f[i] = \texttt{branch}\ c\ j$ implies $c : (t_1, \ldots, t_n) \to t_0$, $i < |F|$, $j \in 1..|F|$, $F[i] = T \cdot t_0$, $F[i+1] = T \cdot (t_1, \ldots, t_n)$ and $F[j] = T \cdot t_0$.

It is now possible to define when an abstract execution is a valid data-flow analysis for a function. We say that a sequence F is a *valid data-flow analysis* for the function $f : (t_1, \ldots, t_n) \to t_0$, denoted *dfa*$(f,F)$, if and only if $F[1] = (t_1, \ldots, t_n)$ and for every $i \in 1..|f|$ we have $dfa_i(f,F)$.

It is easy to define an algorithm computing the result of a valid data-flow analysis, if it exists. To verify a whole program, we simply need to verify every function one at a time.

Define a *directed access graph* $(\{1, \ldots, n\}, E)$ as the least one such that (i,j) and $(i,i+1)$ is in E if $f[i]$ is the instruction `branch c` j and such that $(i,i+1)$ is in E otherwise. (It is the flow graph of the function f.) If every node of the access graph is reachable from the initial node, 1, then the set of constraints has at most one solution: we can assign to every instruction the size of the stack, and for each element in this stack a unique type. In the following we assume that every node in the access graph is accessible.

Next, we define an invariant on programs that passes the bytecode verification (for types) that ensures the well-formedness of the machine during the execution and thus the absence of runtime errors. we define, $arg(M,j)$ the vector of arguments with which (under suitable hypothesis) the j^{th} frame in M has been called. By convention, $arg(M,1)$ is the sequence of values used to initialize the execution of the machine. In the other cases, if $M \equiv (f_1, i_1, h_1) \cdot \cdots \cdot (f_m, i_m, h_m)$, we have

$$arg(M,j) = \begin{cases} (v_1, \ldots, v_k) \text{ if } j > 1, ar(f_j) = k, h_{j-1} = h \cdot (v_1 \cdots v_k) \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

We say that a function is well-formed if every sequence of code terminates either with the `stop` or with the `return` instruction. Moreover, for every indices $i \in 1..|f|$ we ask that: if $f[i] = \texttt{load}\ k$ then $k \geqslant 1$ and if $f[i] = \texttt{branch c}\ j$ then $1 \leqslant j \leqslant |f|$.

A configuration $M \equiv (f_1, i_1, h_1) \cdot \ldots \cdot (f_m, i_m, h_m)$ is *well-formed* if for all $j \in 1..m$ we have (1) $1 \leqslant i_j \leqslant |f_j|$ and (2) $f_j[i_j] = \texttt{call}\ f_{j+1}$ and $arg(M,j)$ is defined for all $j \in 1..m-1$.

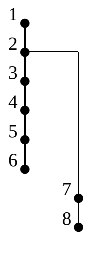
PROPOSITION 4. *Assume we have a valid data-flow analysis for every function and that M_0 is an initial well-formed frame $(f, 1, (v_1, \ldots, v_n))$. If $M_0 \to^* M$ the configuration M is also well-formed.*

EXAMPLE 5. *We continue with our running example and display the type of each instructions in the code of add. We also show the directed access graph associated to the function, which is a tree corresponding to the two possible "execution paths" in the code of add.*

add : $(nat, nat) \to nat$

1	:	(nat, nat)	:	`load 1`
2	:	(nat, nat, nat)	:	`branch s 7`
3	:	(nat, nat, nat)	:	`load 2`
4	:	(nat, nat, nat, nat)	:	`build s`
5	:	(nat, nat, nat, nat)	:	`call` *add* 2
6	:	(nat, nat, nat)	:	`return`
7	:	(nat, nat, nat)	:	`load 2`
8	:	(nat, nat, nat, nat)	:	`return`

E_{add}



4 Path Verification

This section define the first "non-standard" analysis on the bytecode. Instead of simply computing the type of the values in the stack, we prove that we can also (sometimes) obtain informations on the shape of the values, like for instance the identity of the topmost constructor. We give an example of analysis in Appendix **??**.

The path verification associates to every reachable instruction a substitution and to every position of the related stack a term over the signature of constructors and function symbols. This analysis is used in the following size and termination verifications (Sections 5 and 6).

We suppose that the code passes the bytecode verification and use (overload) the symbol h_i for the height of the stack for instruction i. As stated previously, this information is known statically. For every instruction index i and position k such that $1 \leqslant k \leqslant h_i$ we assume a fresh variable $x_{i,k}$ ranging over terms and we uniquely determine a substitution σ_i and an expression $e_{i,k}$ according to the rules in the following table. This computation will never fail if (i) the directed access graph defined in Section 3.2 is a tree rooted at instruction 1, (ii) on every path from the root along the tree the last instruction is either a `return` or a `stop`, and (iii) every `branch` instruction is preceded only by `load` or `branch` instructions[1]. These conditions are not too restrictive and are satisfied by bytecode obtained from the compilation of functional programs (in the absence of optimizations).

$f[i]$	Substitutions and expressions
	$\sigma_1 =_{\text{def}} id$, $e_{1,l} =_{\text{def}} x_{1,l}$ for $l \in 1..ar(f)$
`load` k	$\sigma_{i+1} =_{\text{def}} \sigma_i$, $e_{i+1,l} =_{\text{def}}$ if $l = h_i + 1$ then $e_{i,k}$ else $e_{i,l}$
`branch c` j	If e_{i,h_i} is a variable x, let $\sigma' = [\mathsf{c}(x_{i+1,h_i}, \ldots, x_{i+1,h_i+ar(\mathsf{c})-1})/x]$ in: $\sigma_{i+1} =_{\text{def}} \sigma' \circ \sigma_i$ $e_{i+1,l} =_{\text{def}}$ if $l \leqslant h_i - 1$ then $\sigma'(e_{i,l})$ else $x_{i+1,l}$ If $e_{i,h_i} = \mathsf{c}(e_{h_i}, \ldots, e_{h_i+ar(\mathsf{c})-1})$: $\sigma_{i+1} =_{\text{def}} \sigma_i$ $e_{i+1,l} =_{\text{def}}$ if $l \leqslant h_i - 1$ then $e_{i,l}$ else e_l Otherwise: $\sigma_j =_{\text{def}} \sigma_i$ $e_{j,l} =_{\text{def}} e_{i,l}$ for $l \in 1..h_i$
`build c`	$\sigma_{i+1} =_{\text{def}} \sigma_i$ $e_{i+1,l} =_{\text{def}}$ if $l = h_i - ar(\mathsf{c}) + 1$ then $\mathsf{c}(e_{i,h_i-ar(\mathsf{c})+1}, \ldots, e_{i,h_i})$ else $e_{i,l}$
`call` g	$\sigma_{i+1} =_{\text{def}} \sigma_i$ $e_{i+1,l} =_{\text{def}}$ if $l = h_i - ar(f) + 1$ then $g(e_{i,h_i-ar(f)+1}, \ldots, e_{i,h_i})$ else $e_{i,l}$

Note that applying a `branch c` j instruction to a stack whose head value is of the shape $\mathsf{d}(\ldots)$ with $\mathsf{d} \neq \mathsf{c}$ produces no effect which is fine since then the following instruction is not reachable (since the access graph is a tree, we have $j \neq i+1$).

PROPOSITION 6. *If σ_i is defined then: (1) the vector $(\sigma_i x_{1,1}, \ldots, \sigma_i x_{1,ar(f)})$ is a linear pattern; and (2) if $x \in Var(e_{i,j})$ then x occurs in the linear pattern in (1).*

EXAMPLE 7. *The shape constraints computed for the function add are as follows:*

Expression		*Instruction*		*Substitution*
$1 : (x_{1,1}\ x_{1,2})$	:	`load 1`	:	id
$2 : (x_{1,1}\ x_{1,2}\ x_{1,1})$	:	`branch s 7`	:	id
$3 : (\mathsf{s}(x_{3,3})\ x_{1,2}\ x_{3,3})$	:	`load 2`	:	$[\mathsf{s}(x_{3.3})/x_{1,1}]$
$4 : (\mathsf{s}(x_{3,3})\ x_{1,2}\ x_{3,3}\ x_{1,1})$	:	`build s`	:	$[\mathsf{s}(x_{3.3})/x_{1,1}]$
$5 : (\mathsf{s}(x_{3,3})\ x_{1,2}\ x_{3,3}\ \mathsf{s}(x_{1,1}))$	:	`call` *add* 2	:	$[\mathsf{s}(x_{3.3})/x_{1,1}]$
$6 : (\mathsf{s}(x_{3,3})\ x_{1,2}\ add(x_{1,1}, \mathsf{s}(x_{1,1})))$	:	`return`	:	$[\mathsf{s}(x_{3.3})/x_{1,1}]$
$7 : (x_{1,1}\ x_{1,2}\ x_{1,1})$	:	`load 2`	:	id
$8 : (x_{1,1}\ x_{1,2}\ x_{1,1}\ x_{1,2})$	:	`return`	:	id

[1] In every path from the root we cross a sequence of `branch` and `load` instructions, then a sequence of `load`, `build`, and `call` instructions, and finally either a `stop` or `return` instruction.

The soundness of path verification is obtained through the definition of a new predicate on configuration, *WP* (for well-path), that improves on the "well-type" predicate introduced in the previous section.

- If v is a value then $WP(v)$
- If there exists ρ such that $u_j = \rho(\sigma_i(x_{1,j}))$ for all $j \in 1..k$ and $v_j = \rho(e_{i,j})$ whenever $e_{i,j}$ is a pattern then $WP(f, (u_1, \ldots, u_k), i, v_1 \cdots v_{h_i})$
- Assume M is the configuration $(f_1, i_1, h_1) \cdot \cdots \cdot (f_m, i_m, h_m)$ and $h'_j = arg(M, j)$ for all $j \in 1..m$. If $WP(f_j, h'_j, i_j, h_j)$ for all $j \in 1..m$ then $WP(M)$.

PROPOSITION 8. *If $WP(M)$ and $M \rightarrow M'$ then $WP(M')$.*

5 Value Size Verification

We assume that we have synthesized suitable quasi-interpretations at the language level (before compilation) and that these informations are added to the bytecode. Hence, for every constructor c and function symbol f, the functions $q_{\mathsf{c}} : (\mathbf{Q}^+)^{ar(\mathsf{c})} \rightarrow \mathbf{Q}^+$ and $q_f : (\mathbf{Q}^+)^{ar(f)} \rightarrow \mathbf{Q}^+$ are given.

We prove that we may check the validity of the quasi-interpretations at the bytecode level (and then prevent malicious code containing deceitful size annotations) and that we may infer a bound on the size needed for the computation of the bytecode. The bound is a polynomials in the size of the argument.

We assume the byte code passes the path verification. Thus for every instruction index i in the segment of the function f the expressions $e_{i,l}$ for $1 \leqslant l \leqslant h_i$ and the substitution σ_i are determined.

We require that the following condition holds for $k = ar(f)$, $i = 1, \ldots, n$, $l = 1, \ldots, h_i$:

$$q_{f(\sigma_i x_{1,1}, \ldots, \sigma_i x_{1,k})} \geqslant q_{e_{i,l}} \tag{1}$$

Of course, the complexity of verifying condition (1) depends on the space of quasi-interpretations selected. We also notice that the condition is quite redundant. First, by the definition of quasi-interpretation the requirement is automatically verified for all instructions which are not preceded by a `build` or `call` instruction. Second, for the remaining instructions we could just perform one verification for every path that terminates with a `return` instruction provided that: (i) on a path terminating with a `stop` instruction, no `build` or `call` instructions occur and (ii) on a path terminating with a `return` instruction, the expressions built with `build` and `call` actually appear as subexpressions of the *returned* expression. These two conditions are needed because otherwise, a malicious bytecode could allocate on the frame large values which are not actually used.

Our result directly follows from the definition of an invariant on well-sized programs. We introduce a predicate *WS* (well-sized) on stacks M such that $WP(M)$ as follows:

- If v is a value then $WS(v)$
- If there exists ρ such that $u_j = \rho(\sigma_i(x_{1,j}))$ for all $j \in 1..k$ and $q_{\rho(e_{i,j})} \geqslant q_{v_j}$ for $j \in 1..h_i$ then $WS(f, (u_1, \ldots, u_k), i, v_1 \cdots v_{h_i})$
- Assume M is the configuration $(f_1, i_1, h_1) \cdot \cdots \cdot (f_m, i_m, h_m)$ and $h'_j = arg(M, j)$ for all $j \in 1..m$. If $WP(f_j, h'_j, i_j, h_j)$ and $q_{f_j(h'_j)} \geqslant q_{f_{j+1}(h'_{j+1})}$ for all $j \in 1..m$ then $WP(M)$.

PROPOSITION 9. *If $WS(M)$ and $M \rightarrow M'$ then $WS(M')$.*

As a corollary we show that if a program passes the size verification then the size of every value appearing in the computation of a function call, $f(v_1, \ldots, v_n)$, is bounded by the quasi-interpretation of $f(v_1, \ldots, v_n)$.

COROLLARY 10. *Assume we have a valid data-flow analysis for every function and that M_0 is an initial well-formed configuration $(f,1,(v_1,\ldots,v_n))$. If $M_0 \rightarrow^* M \cdot (g,i,h)$ and v is a value appearing in h then $|v| \leqslant q_{f(v_1,\ldots,v_k)}$.*

6 Termination Verification

We suppose that the code passes the stack height and path verification. We assume given a pre-order $\geqslant_\Sigma$ on the function symbols so that $f =_\Sigma g$ implies $ar(f) = ar(g)$. The pre-order is extended to the constructor symbols by assuming that a constructor is always smaller than a function symbol and that two distinct constructors are incomparable. We suppose the *status* of every function symbol is lexicographic (and compatible with $\geqslant_\Sigma$) and that the *status* of every constructor symbol is the product. We denote with $>_l$ the induced path order. Note that on values $v >_l v'$ iff v embeds homomorphically v' and $v >_l v'$ implies $|v| > |v'|$. We require that the following conditions hold for $i = 1,\ldots,n$ and $l = 1,\ldots,h_i$:

$$f(\sigma_i x_{1,1},\ldots,\sigma_i x_{1,ar(f)}) >_l e_{i,l} \tag{2}$$

This invariant resembles the one for value size. We introduce a predicate *TER* (terminating) on configurations M such that $WP(M)$ as follows:

- If v is a value then $TER(v)$
- If there exists ρ such that $u_j = \rho(\sigma_i(x_{1,j}))$ for all $j \in 1..k$ and $\rho(e_{i,j}) \geqslant_l v_j$ for $j \in 1..h_i$ then $TER(f,(u_1,\ldots,u_k),i,v_1 \cdots v_{h_i})$
- Assume M is the configuration $(f_1,i_1,h_1) \cdot \cdots \cdot (f_m,i_m,h_m)$ and $h'_j = arg(M,j)$ for all $j \in 1..m$. If $TER(f_j,h'_j,i_j,h_j)$ and $f_j(\vec{u}_j) >_l f_{j+1}(\vec{u}_{j+1})$ for all $j \in 1..m$ then $TER(M)$.

PROPOSITION 11. *If $TER(M)$ and $M \rightarrow M'$ then $TER(M')$.*

COROLLARY 12. *Every execution starting with a frame $(f,1,(v_1,\ldots,v_k))$, terminates.*

PROOF. We define a well-ordering on the configurations that is compatible with the evaluation of the machine. $\square$

As observed in [2], termination by lexicographic order combined with a polynomial bound on the size of the values leads to polynomial space. We derive a similar result with a similar proof at bytecode level.

COROLLARY 13. *Suppose that the quasi-interpretations are bound by polynomials and that the bytecode passes the value size and termination verifications. Then every execution starting from a frame $(f,1,v_1,\ldots,v_k)$ (terminates and) runs in space polynomial in the size of the arguments $|v_1|,\ldots,|v_k|$.*

PROOF. Note that if $f(\vec{v}) >_l g(\vec{u})$ then either $f >_\Sigma g$ or $f =_\Sigma g$ and $\vec{v} >_l \vec{u}$. In a sequence $f_1(\vec{v}_1) >_l \cdots f_m(\vec{v}_m)$, the first case can occur a constant number of times (the number of equivalence classes of function symbols with respect to $\geqslant_\Sigma$) thus it is enough to analyze the length of strictly decreasing sequences of tuples of values $(v_1,\ldots,v_k)$ lexicographically ordered where k is the largest arity of a function symbol. If b is a bound on the size of the values then since on values $v >_l v'$ implies $|v| > |v'|$ we derive that the sequence has length at most b^k. Since b is polynomial in the size of the arguments and the number of values on a frame is bound by a constant (via the stack height verification), a polynomial bound is easily derived. $\square$

7 Conclusion

The problem of bounding the size of the memory needed for executing a program has already attracted considerable attention. Nonetheless, automatic extraction of resource bounds has mainly focused on first-order functional languages and very few works address this problem at the level of the bytecode (or of the compiled program).

In this paper, we study the resource bounds problem in a simple stack machine and show how to perform type, size, and termination verifications at the level of the bytecode. In particular, we show that a combination of size verification based on quasi-interpretations and of termination verification based on lexicographic path orders leads to an explicit polynomial bound on the space required for the execution. More than simplifying our presentation, the choice of a simple set of bytecode instructions is of a pedagogical nature: we can present a minimal but still relevant scenario in which problems connected to bytecode verification can be effectively discussed.

We are in the process of formalizing our virtual machine and the related security properties in the COQ proof assistant. As a second step, we plan to experiment with the automatic derivation of proofs (certificates) for the object code of the virtual machine starting from the static analyses performed on the source code. Plans for future works also include extending our analysis to a richer language with, for instance, unconditional jumps (the "infamous" `Goto` statement) or subroutines.

8 References

[1] R. Amadio. Max-plus quasi-interpretations. In *Proc. Typed Lambda Calculi and Applications (TLCA '03)*, LNCS, to appear. Springer, 2003.

[2] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Andrei Ershov Fourth International Conference "Perspectives of System Informatics", LNCS*. Springer, 2001.

[3] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[4] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II, North Holland*, 1965.

[5] B. Gramlich. On proving termination by innermost termination. In *Proc. 7th Int. Conf. on Rewriting Techniques and Applications (RTA '96)*, volume 1103 of *LNCS*, pp. 93–107. Springer, 1996.

[6] M. Hofmann. The strength of non size-increasing computation. In *Proc. POPL*. ACM Press, 2002.

[7] N. Jones. *Computability and complexity, from a programming perspective*. MIT-Press, 1997.

[8] J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. PhD thesis, Université de Nancy, 2000. Habilitation à diriger des recherches.

[9] J.-Y. Marion, J.-Y. Moyen. *Termination and resource analysis of assembly programs by Petri Nets*. Technical Report, Université de Nancy, 2003.

[10] G. Morriset, D. Walker, K. Crary and N. Glew. From System F to Typed Assembly Language. In *ACM Transactions on Programming Languages and Systems*, 21(3):528-569, 1999.

[11] G. Necula. Proof carrying code. In *Proc. POPL*. ACM Press, 1997.

[12] D. Sannella. Mobile resource guarantee. Ist-global computing research proposal, U. Edinburgh, 2001. `http://www.dcs.ed.ac.uk/home/mrg/`.