A Functional Scenario for Bytecode Verification of Resource Bounds*

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

Laboratoire d'Informatique Fondamentale de Marseille (LIF),
CNRS and Université de Provence.

Abstract. We consider a scenario where (functional) programs in pre-compiled form are exchanged among untrusted parties. Our contribution is a system of annotations for the code that can be verified at load time so as to ensure bounds on the time and space resources required for its execution, as well as to guarantee the usual integrity properties.
Specifically, 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 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 on the JAVA platform. Application scenarios include, for instance, programmable switches, network games, and applications for smart cards. A prevailing conclusion is that 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. Initial proposals have focused on the integrity properties of the execution environment such as 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.

The interest of carrying on such analyses at bytecode level are now well understood [15,16]. First, mobile code is shipped around in pre-compiled (or *bytecode*) form and needs to be analysed 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, we can reduce the size of the trusted code base and shift from the reliance on the correctness of the whole compilation chain to only the trust on the analyser.

Approach. The problem of bounding the usage made by programs of their resources has already attracted considerable attention. Automatic extraction of resource bounds

* This work was partly supported by ACI Sécurité Informatique, project CRISS.

has mainly focused on (first-order) functional languages starting from Cobham's characterisation [7] of polynomial time functions by bounded recursion on notation. Following work, see *e.g.*, [4,8,9,11], has developed 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 [5] that a program admitting a polynomially bound quasi-interpretation and terminating by lexicographic path-ordering runs in polynomial space. This and other results can be regarded as generalisations and variations over Cobham's characterisation.

Contribution. The synthesis of termination orderings is a classical topic in term rewriting (see for instance [6]). The synthesis of quasi-interpretations — a concept introduced by Marion *et al.* [13] — 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 [2]. 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.

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.

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 a program "is in PSPACE" we extract a polynomial that bounds the size needed to run a program: given a function (identifier) f of arity n in a verified program, 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 $|v|$ is the size of the value v.

A secondary goal of our work is of a pedagogical nature: present a minimal — the virtual machine includes only 6 instructions — but still relevant scenario in which problems connected to bytecode verification can be effectively discussed.

Our approach to resource bound certification follows distinctive design decisions. First, we allow the space needed for the execution of a program to vary depending on the size of its arguments. This is in contrast to most approaches that try to enforce a constant space bound. While this latter goal is reasonable for applications targeting embedded devices, we believe that it is not always relevant in the context of mobile code. Second, our method is applicable to a large class of algorithms and do not impose specific syntactical restrictions on programs. For example, we depart from works based on a linear usage of variables [8]. Given the specificities of our method, we may often ensure bounds on resources where other methods fail, but we may also give very rough estimate of the space needed, *e.g.* in cases where another method would have detected that memory operations may be achieved in-place. Hence, it may be interesting to couple our analysis with other methods for ensuring resource bounds.

Paper organisation. The paper is organised 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. In Section 4, we define a type verification 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 6) and with an order to guarantee termination (Section 7). The size and termination verifications depend on a shape verification which is described in Section 5.

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. Additional technical details and omitted proofs can be found in a long version of this extended abstract [3].

2 A Functional Language

We consider a simple, typed, first-order functional language, with inductive types and pattern-matching. A program 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 in the language 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 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 $f(p_1, \ldots, p_n) \Rightarrow e$, where e is an expression. We follow the usual hypothesis that the patterns $p_1, \ldots, p_n$ are linear and do not superpose. 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 $f(e_1, \ldots, e_n)$.

Types. We use $t, t_1, \ldots$ to range over type identifiers. A type definition associates with each identifier the sequence of the types of its constructors, of the form $\mathsf{c}\ of\ t_1 * \cdots * t_n$. For instance, we can define the type $bword$ of binary words and the type nat of natural numbers in unary format:

$$bword = \mathsf{nil} \ \mid \ \mathsf{0}\ of\ bword \ \mid \ \mathsf{1}\ of\ bword \qquad\qquad nat = \mathsf{z} \ \mid \ \mathsf{s}\ of\ nat$$

In the following, we consider that constructors are declared with their functional type $(t_1, \ldots, t_n) \to t$. Similar types can be either assigned or inferred for the function symbols. We use the notation $f : (t_1, \ldots, t_n) \to t$ to refer to the type of f and $ar(f)$ for the arity of f. We use similar notations for constructors. The typing rules for the language are standard and are omitted — all the results given in this paper could be easily extended to a system with parametric polymorphism.

Evaluation. The following two rules define the standard call-by-value evaluation relation, where σ is a substitution from variables to values. In order to define the rule selected in the evaluation of a function call, we rely on the function $match$ which returns the unique substitution (if any) defined on the variables in the patterns and matching the patterns against the vector of values. In particular, the condition $match((p_1, \ldots, p_n), (v_1, \ldots, v_n)) = \sigma$ imposes that $\sigma(p_i) = v_i$ for all $i \in 1..n$.

$$\frac{e_j \Downarrow v_j \qquad j \in 1..n}{\mathsf{c}(e_1, \ldots, e_n) \Downarrow \mathsf{c}(v_1, \ldots, v_n)} \qquad \frac{\begin{array}{c} f(p_1, \ldots, p_n) \Rightarrow e\ rule \quad e_j \Downarrow v_j \quad j \in 1..n \\ match((p_1, \ldots, p_n), (v_1, \ldots, v_n)) = \sigma \quad \sigma(e) \Downarrow v \end{array}}{f(e_1, \ldots, e_n) \Downarrow v}$$

Example 1. The function add of type $(nat,\ nat) \to nat$, defined by the following two rules, computes the sum of two natural numbers.

$$add(\mathsf{z}, y) \Rightarrow y \qquad\qquad add(\mathsf{s}(x), y) \Rightarrow add(x, \mathsf{s}(y))$$

Quasi-interpretations. Given a program, an *assignment* q associates with constructors $\mathsf{c}, \ldots$ and function symbols $f, \ldots$, functions $q_{\mathsf{c}}, q_f, \ldots$ over the non-negative reals $\mathbb{R}^+$ such that: (i) if c is a constant then q_{c} is the constant[1] 0, (ii) if c is a constructor with arity $n \geqslant 1$ then q_{c} is the function in $(\mathbb{R}^+)^n \to \mathbb{R}^+$ such that $q_{\mathsf{c}}(x_1, \ldots, x_n) = d + \Sigma_{i \in 1..n} x_i$, for some $d \geqslant 1$, and (iii) if f is a function (identifier) with arity n then $q_f : (\mathbb{R}^+)^n \to \mathbb{R}^+$ is monotonic and for all $i \in 1..n$ we have $q_f(x_1, \ldots, x_n) \geqslant x_i$. An assignment q is 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 $q_{f(e_1, \ldots, e_n)} = q_f(q_{e_1}, \ldots, q_{e_n})$.

Thus for every expression e we have a function expression q_e with variables in $Var(e)$. An assignment is a *quasi-interpretation* if for every rule $f(p_1, \ldots, p_n) \Rightarrow e$ in the program, the inequality $q_{f(p_1, \ldots, p_n)} \geqslant q_e$ holds over $\mathbb{R}^+$.

Example 2. With reference to Example 1, consider the assignment $q_{\mathsf{s}}(x) = 1 + x$ and $q_{add}(x, y) = x + y$. Since by definition $q_{\mathsf{z}} = 0$, we note that $q_v = |v|$ for all values v of type nat. Moreover, it is easy to check that q is a quasi-interpretation as the inequalities $q_{add}(0, y) \geqslant y$ and $q_{add}(1 + x, y) \geqslant q_{add}(x, 1 + y)$ hold. $\square$

[1] We can choose any positive real constant for q_{c}, but this choice simplifies some of our proofs.

Quasi-interpretations are designed so as to provide a bound on the size of the computed values as a function of the size of the input data. An interesting space for the synthesis of quasi-interpretations is the collection of max-plus polynomials [2], 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 $a_{i,j} \in \mathbb{N}$ and $a_i \in \mathbb{Q}^+$, where $\mathbb{N}$ are the natural numbers and $\mathbb{Q}^+$ are the non-negative rationals. In this case, checking whether an assignment is a quasi-interpretation can be reduced to checking the satisfiability of a Presburger formula, and is therefore a decidable problem.

3 The Virtual Machine

We define a simple stack machine and a related set of bytecode instructions for the compilation and the evaluation of programs. We adopt the usual notation on words: ϵ is the empty sequence, $x \cdot x'$ is the concatenation of two sequences x, x'. We may also omit the concatenation operation $\cdot$ by simply writing $x\,x'$. Moreover, if x is a sequence then $|x|$ is its length and $x[i]$ its i^{th} element counting from 1. We denote with $\boldsymbol{y}$ a vector $(y_1, \ldots, y_n)$ of elements. Then, $\boldsymbol{y}_i$ stands for the element y_i and $|\boldsymbol{y}|$ is the number n of elements in the vector. In the following, we will often manipulate *vectors of sequences* and use the notation $\boldsymbol{y}_i[k]$ to denote the k^{th} element in the i^{th} sequence of vector $\boldsymbol{y}$.

We suppose given a program with a set of *constructor names* and a disjoint set of *function names*. A function identifier f will also denote the sequence of instructions of the associated code. Then $f[i]$ stands for the i^{th} instruction in the (compiled) code of f and $|f|$ for the number of instructions.

The virtual machine is built around a few components: (1) an *association list* between function identifiers and function codes; (2) a *configuration* M, which is a sequence of *frames* representing the memory of the machine; (3) a *bytecode interpreter* modelled as a reduction relation on configurations. In turn, a *frame* is a triple (f, pc, ℓ) composed of a function identifier, the value of the program counter (a natural number in $1..|f|$), and a *stack*. A *stack* is a sequence of values that serves both to store the parameters and the values computed during the execution. We work with a minimal set of instructions whose effect on the configuration is described in Table 1 and write $M \to M'$ if M reduces to M' by applying exactly one of the transformations.

The reduction $M \to M'$ is deterministic. The empty sequence of frames ϵ is a special state which cannot be accessed during a computation not raising an error, *i.e.*, not executing the instruction `stop`. A “good” execution starts with a configuration of the form $(f, 1, v_1 \cdots v_n)$, containing only one frame that corresponds to the evaluation of the expression $f(v_1, \ldots, v_n)$. The execution ends with a configuration of the form $(f, pc, \ell \cdot v_0)$ where $1 \leqslant pc \leqslant |f|$ and $f[pc] = \texttt{return}\ n$ (the integer n is the arity of f). In this case the result of the evaluation is v_0. By extension, we say that the configuration M is a result v_0, denoted $M \downarrow v_0$, if there exists a sequence ℓ such that $M \equiv (f, pc, \ell \cdot v_0)$ with $1 \leqslant pc \leqslant |f|$ and $f[pc] = \texttt{return}\ n$. All the other cases of blocked configuration, such that $M \not\to$, are considered as runtime errors.

The language described in section 2 admits a direct compilation in our functional bytecode. 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.

$$\frac{f[pc] = \texttt{load}\, i \quad pc < |f| \quad \ell[i] = v}{M \cdot (f, pc, \ell) \to M \cdot (f, pc+1, \ell \cdot v)} \qquad \frac{f[pc] = \texttt{build}\, \mathsf{c}\, n \quad pc < |f| \quad \ell = \ell' \cdot v_1 \cdots v_n}{M \cdot (f, pc, \ell) \to M \cdot (f, pc+1, \ell' \cdot \mathsf{c}(v_1, \ldots, v_n))}$$

$$\frac{f[pc] = \texttt{branch}\, \mathsf{c}\, j \quad pc < |f| \quad \ell = \ell' \cdot \mathsf{c}(v_1, \ldots, v_n)}{M \cdot (f, pc, \ell) \to M \cdot (f, pc+1, \ell' \cdot v_1 \cdots v_n)} \qquad \frac{f[pc] = \texttt{branch}\, \mathsf{c}\, j \quad 1 \leqslant j \leqslant |f| \quad \ell = \ell' \cdot \mathsf{d}(\ldots) \quad \mathsf{c} \neq \mathsf{d}}{M \cdot (f, pc, \ell) \to M \cdot (f, j, \ell)}$$

$$\frac{f[pc] = \texttt{call}\, g\, n \quad pc < |f| \quad \ell = \ell' \cdot v_1 \cdots v_n}{M \cdot (f, pc, \ell) \to M \cdot (f, pc, \ell) \cdot (g, 1, v_1 \cdots v_n)} \qquad \frac{f[pc] = \texttt{stop}}{M \cdot (f, pc, \ell) \to \epsilon}$$

$$\frac{f[pc] = \texttt{return}\, n \quad \ell = \ell_0 \cdot v_0 \quad \ell' = \ell'' \cdot v_1 \cdots v_n}{M \cdot (g, pc', \ell') \cdot (f, pc, \ell) \to M \cdot (g, pc'+1, \ell'' \cdot v_0)}$$

Table 1. Bytecode Interpreter: $M \to M'$

Clearly, a realistic implementation should at least include a mechanism to execute efficiently tail recursive calls (when a `call` instruction is immediately followed by `return`) and a mechanism to share common sub-values in a configuration. For instance, using a stack of pointers to values allocated on a heap, it is possible to dispense with the copy performed by a `load` instructions. Our approach to size verification of the stack could be adapted to these possible enhancements of the virtual machine.

Preliminary Verifications. We define a minimal set of (syntactical) conditions on the shape of the code so as to avoid the simplest form of errors, *e.g.*, to guarantee that the program counter stays within the intended bounds.

A new frame may only originate from a `call` instruction, that is, for every pair of contiguous frames, $\cdots (f, pc, \ell)\ (g, pc', \ell') \cdots$, the instruction $f[pc]$ must be of the form `call` $g\ n$ and the stack ℓ must end with n values, say $v_1 \cdots v_n$, which are the parameters used in the call for g. We use the notation $arg(M, j)$ to refer to the vector of arguments with which the j^{th} frame in M has been called: if $1 < j \leqslant m$ and $M \equiv (f_1, i_1, \ell_1) \cdots (f_m, i_m, \ell_m)$, we have $arg(M, j) = (v_1, \ldots, v_k)$ where $ar(f_j) = k$ and $\ell_{j-1} = \ell \cdot v_1 \cdots v_k$. (An alternative presentation of the reduction rules could be to carry these parameters explicitly as extra annotations on each frame.) By convention, we use $arg(M, 1)$ for the sequence of values used to initialise the execution of the machine.

We say that a function f is *well-formed* if the sequence of code of f terminates either with the `stop` or with the `return` instruction. Moreover, for every index $i \in 1..|f|$, we ask that: (1) if $f[i] = \texttt{load}\, k$ then $k \geqslant 1$ and (2) if $f[i] = \texttt{branch}\, \mathsf{c}\, j$ then $1 \leqslant j \leqslant |f|$. We assume that every function in the code is well-formed; the result of the compilation of functional programs clearly meets these well-formedness conditions. We say that a configuration $M \equiv (f_1, i_1, \ell_1) \cdots (f_m, i_m, \ell_m)$ is *well-formed* if for all $j \in 1..m$ we have (1) the program counter i_j is in $1..|f_j|$; (2) the expression $arg(M, j)$ is defined; and (3) for all $j \in 1..m-1$ we have $f_j[i_j] = \texttt{call}\, f_{j+1}\, n_{j+1}$ — type verification will ensure, among other properties, that n_j is the arity of the function f_j. Well-formedness is preserved during execution (and the configuration ϵ is well-formed).

Proposition 1. *If M is a well-formed configuration and $M \to M'$ then the configuration M' is also well-formed.*

case $f[i]$ of

`load` k	: $i < \lvert\boldsymbol{T}\rvert$, $\boldsymbol{T}_i[k] = t$ and $\boldsymbol{T}_{i+1} = \boldsymbol{T}_i \cdot t$
`build c` n	: let `c` : $(t_1, \ldots, t_n) \to t_0$ in $\exists T.\, i < \lvert\boldsymbol{T}\rvert$, $\boldsymbol{T}_i = T \cdot t_1 \cdots t_n$ and $\boldsymbol{T}_{i+1} = T \cdot t_0$
`call` $g\ n$	: let $g : (t_1, \ldots, t_n) \to t_0$ in $\exists T.\, i < \lvert\boldsymbol{T}\rvert$, $\boldsymbol{T}_i = T \cdot t_1 \cdots t_n$ and $\boldsymbol{T}_{i+1} = T \cdot t_0$
`return` n	: let $f : (t_1, \ldots, t_n) \to t_0$ in $\exists T.\, \boldsymbol{T}_i = T \cdot t_0$
`stop`	: true
`branch c` j	: let `c` : $(t_1, \ldots, t_n) \to t_0$ in $\exists T.\, i < \lvert\boldsymbol{T}\rvert$, $\boldsymbol{T}_i = T \cdot t_0$, $\boldsymbol{T}_{i+1} = T \cdot t_1 \cdots t_n$ and $\boldsymbol{T}_j = \boldsymbol{T}_i$

Table 2. Well-Typed Instructions: $wt_i(f, \boldsymbol{T})$

4 Type Verification

In this section, we define a simple type verification to ensure the well-formedness and well-typedness of the machine configurations during execution. This verification is very similar to the so called *bytecode verification* in the JAVA platform, see *e.g.* [1], and can be directly used as the basis of an algorithm for validating the bytecode before its execution by the interpreter. (A major difference is that we do not have to consider subroutines, access modifiers or object initialisation in our language.)

Type verification associates with every instruction (every step in the evaluation of a function code) an abstraction of the stack. In our case, an abstract stack is a sequence of types, or *type stack*, $T = t_1 \cdots t_n$, that should exactly match the types of the values present in the stack at the time of the execution. Accordingly, an *abstract execution* for a function f is a sequence $\boldsymbol{T}$ of type stacks such that $|\boldsymbol{T}| = |f|$.

To express that an abstract execution $\boldsymbol{T}$ is coherent with the instructions in f, we define the notion of *well-typed instruction* based on the auxiliary relation $wt_i(f, \boldsymbol{T})$, given below. Informally, we show that if $wt_i(f, \boldsymbol{T})$ and $\boldsymbol{T}_i = t_1 \cdots t_k$ then for every valid evaluation of f, the stack of values at the time of the execution of $f[i]$ is $\ell = v_1 \cdots v_k$ where v_j is a value of type t_j for every $j \in 1..k$. The definition of the relation $wt_i(f, \boldsymbol{T})$, where $|f| = |\boldsymbol{T}|$, is by case analysis on the instruction $f[i]$.

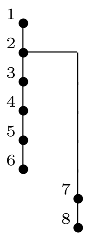
We define a well-typed function as a sequence of well-typed instructions. To verify a whole program, we simply need to verify every function separately.

Definition 1 (Well-Typed Function). *A sequence $\boldsymbol{T}$ is a* valid abstract execution *for the function f with signature $(t_1, \ldots, t_n) \to t_0$, denoted $wt(f, \boldsymbol{T})$, if and only if $\boldsymbol{T}_1 = t_1 \cdots t_n$ and $wt_i(f, \boldsymbol{T})$ for every $i \in 1..|f|$.*

We define the *flow graph* of function f as the directed graph $(\{1, \ldots, |f|\}, E_f)$ such that for all $i \in 1..|f| - 1$, the edge $(i, i+1)$ is in E_f if $f[i]$ is a `load`, `build`, or `call` instruction and the edges $(i, i+1)$ and (i, j) are in E_f if $f[i]$ is the instruction `branch c` j. If every node in the flow graph $(\{1, \ldots, |f|\}, E_f)$ is reachable from the node 1 then there is at most one abstract execution, $\boldsymbol{T}$, such that $wt(f, \boldsymbol{T})$. Moreover, $\boldsymbol{T}$ can be effectively computed as the fixpoint of a function iterating the conditions given in Table 2, for example using Kildall's algorithm [10].

Example 3. We continue with our running example and display the type of each instruction in the (compiled) code of add. We also show the flow graph associated with the function that exhibits the two possible "execution paths" in the code of add.

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 1
5 :	*nat nat nat nat*	: `call` *add* 2
6 :	*nat nat nat*	: `return` 2
7 :	*nat nat nat*	: `load` 2
8 :	*nat nat nat nat*	: `return` 2



In the following, we assume that every node in the flow graph is accessible. If $\boldsymbol{T}$ is "the" abstract execution of f, we say that f is a function (code) of type $\boldsymbol{T}$. Next, we prove that the execution of verified programs never fails. As expected, we start by proving that type information is preserved during evaluation. This relies on the notions of well-typed frames and configurations. For instance, we say that a stack has type T, denoted $\ell : T$, if $T = t_1 \cdots t_n$ and $\ell = v_1 \cdots v_n$, where v_i is of type t_i for all $i \in 1..n$.

$$\frac{\mathsf{c} : (t_1, \dots, t_n) \to t \qquad v_i : t_i \qquad i \in 1..n}{\mathsf{c}(v_1, \dots, v_n) : t} \qquad \frac{v_i : t_i \qquad i \in 1..n}{v_1 \cdots v_n : t_1 \cdots t_n}$$

$$\frac{wt(f, \boldsymbol{T}) \qquad \ell : \boldsymbol{T}_i}{wt(f, i, \ell)} \qquad \frac{\begin{array}{c} M \equiv (f_1, i_1, \ell_1) \ \dots \ (f_m, i_m, \ell_m) \text{ well-formed} \\ wt(f_j, i_j, \ell_j) \qquad j \in 1..m \end{array}}{wt(M)}$$

Proposition 2 (Type Invariant). *Let M be a configuration. If $wt(M)$ and $M \to M'$ then $wt(M')$.*

We note that as a side result of the type verification, we obtain, for every instruction, the size of the stack at the time of its execution. The soundness of the type verification follows from a progress property.

Proposition 3 (Progress). *Assume M is a well-typed configuration. Then either $M \equiv \epsilon$, or M is a result, $M \downarrow v_0$, or M reduces, $\exists M' \ (M \to M')$.*

5 Shape Analysis

We define a shape analysis on the bytecode which appears to be original. Instead of computing the type of the values in the stack, we prove that we can also obtain partial information on their shape such as the identity of their top-most constructor. This verification is used in the following size and termination verifications (Sections 6 and 7). We suppose that the code of every function f in the program passes the type verification of Section 4 and that $wt(f, \boldsymbol{T})$ holds. We denote with $\boldsymbol{h}$ a vector of numbers such that $\boldsymbol{h}_i$ is the height of the stack for instruction i, that is $\boldsymbol{h}_i = |\boldsymbol{T}_i|$ for all $i \in 1..|f|$. Furthermore, for every instruction index i and position $k \in 1..\boldsymbol{h}_i$ in the corresponding stack we assume a fresh variable $x_{i,k}$ ranging over expressions, that is terms built from variables, constructors and function symbols.

```
case $f[i]$ of
  load $k$     : $\boldsymbol{\sigma}_{i+1} = \boldsymbol{\sigma}_i$ and $\boldsymbol{E}_{i+1} = \boldsymbol{E}_i \cdot \boldsymbol{E}_i[k]$
  build c $n$  : $\boldsymbol{\sigma}_{i+1} = \boldsymbol{\sigma}_i$, $\boldsymbol{E}_i = E \cdot e_1 \cdots e_n$ and $\boldsymbol{E}_{i+1} = E \cdot \mathsf{c}(e_1, \ldots, e_n)$
  call $g$ $n$ : $\boldsymbol{\sigma}_{i+1} = \boldsymbol{\sigma}_i$, $\boldsymbol{E}_i = E \cdot e_1 \cdots e_n$ and $\boldsymbol{E}_{i+1} = E \cdot g(e_1, \ldots, e_n)$
  branch c $j$ : let $\boldsymbol{E}_i = E \cdot p$ in
    if $p$ is a variable $x$ then
      let $\sigma' = [\mathsf{c}(x_{i+1,\boldsymbol{h}_i}, \ldots, x_{i+1,\boldsymbol{h}_{i+1}})/x]$ in
      $\boldsymbol{\sigma}_j = \boldsymbol{\sigma}_i$, $\boldsymbol{E}_j = \boldsymbol{E}_i$, $\boldsymbol{\sigma}_{i+1} = \sigma' \circ \boldsymbol{\sigma}_i$ and $\boldsymbol{E}_{i+1} = \sigma'(E) \cdot x_{i+1,\boldsymbol{h}_i} \cdots x_{i+1,\boldsymbol{h}_{i+1}}$
    else if $p = \mathsf{c}(e_1, \ldots, e_n)$ then $\boldsymbol{\sigma}_{i+1} = \boldsymbol{\sigma}_i$ and $\boldsymbol{E}_{i+1} = E \cdot e_1 \cdots e_n$
    else if $p = \mathsf{d}(\ldots)$ with $\mathsf{d} \neq \mathsf{c}$ then $\boldsymbol{\sigma}_j = \boldsymbol{\sigma}_i$ and $\boldsymbol{E}_j = \boldsymbol{E}_i$
                                        (where $\boldsymbol{h}_{i+1} = \boldsymbol{h}_i + ar(\mathsf{c}) - 1$ and $\boldsymbol{h}_j = \boldsymbol{h}_i$)
```

Table 3. Shape Constraints at Instruction i: $wsh_i(f, \boldsymbol{\sigma}, \boldsymbol{E})$

We show that under some restrictions on the form of the code, we can solve certain shape constraints and associate with every reachable instruction a substitution, $\boldsymbol{\sigma}_i$, and to every position of the related stack an expression, $e_{i,j}$ (if f is well-typed and every node in its flow graph is reachable then the solution is unique). We can compare the shape analysis with the type verification of Section 4: we compute for each instruction a sequence of expressions, $E = e_1 \cdots e_n$, instead of a sequence of types $T = t_1 \cdots t_n$. The restrictions on the code are the following:

(1) the flow graph of the function is a tree rooted at instruction 1 whose leaves correspond to the instructions `return` or `stop`;

(2) every `branch` instruction is preceded only by `load` or `branch` instructions.

These conditions are satisfied by the bytecode obtained from the compilation of functional programs and entail that 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 shape constraints are displayed below. We 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 flow graph is a tree, we have $j \neq i + 1$). Hence the shape analysis may also be used to locate dead code. The definition of the relation $wsh_i(f, \boldsymbol{\sigma}, \boldsymbol{E})$, where $|f| = |\boldsymbol{\sigma}| = |\boldsymbol{E}|$, is by case analysis on the instruction $f[i]$. There are no constraints on $\boldsymbol{\sigma}$ and $\boldsymbol{E}$ if $f[i]$ is a `return` or `stop` instruction.

The soundness of shape verification is obtained through the definition of a new predicate on configurations, wsh, which improves on the "well-typed" predicate introduced in the previous section.

Definition 2 (Well-Shaped Function). *A pair $(\boldsymbol{\sigma}, \boldsymbol{E})$ is a valid shape for the function f of type $(t_1, \ldots, t_n) \rightarrow t_0$, denoted $wsh(f, \boldsymbol{\sigma}, \boldsymbol{E})$, if $\boldsymbol{\sigma}_1$ is the identity substitution, id, $\boldsymbol{E}_1 = x_{1,1} \cdots x_{1,ar(f)}$, and $wsh_i(f, \boldsymbol{\sigma}, \boldsymbol{E})$ for all $i \in 1..|f|$.*

Assume we have a well-formed configuration M containing the frame (f, i, ℓ) in j^{th} position and that $\arg(M, j) = (u_1, \ldots, u_k)$ are the parameters used to initialise this frame. The substitution $\boldsymbol{\sigma}_i$ relates the values $u_1, \ldots, u_k$ to the values occurring in ℓ. More precisely, $\boldsymbol{\sigma}_i(x_{1,l})$ is a pattern with variables in $(x_{i,j})_{j \in 1..\boldsymbol{h}_i}$ and there is at most one matching substitution ρ such that $\rho \circ \boldsymbol{\sigma}_i(x_{1,l}) = u_l$ for all $l \in 1..k$. On the other hand, the expressions $e_{i,j}$ describe the values occurring in ℓ. If $e_{i,j}$ is a pattern, that is, if it does not contain a function symbol (which is always the case if the instruction $f[i]$ occurs before the first function call in the execution path), then $\ell[j] = \rho(e_{i,j})$.

For example, if we consider the shape constraints computed for the function add below, we have that for every frame (add, i, ℓ) originating from the parameters $(u_1\ u_2)$, if $i = 5$ (at the point of the recursive call) then u_1 is of the form $\mathsf{s}(u_3)$ and ℓ is the stack $(\mathsf{s}(u_3)\ u_2\ u_3\ \mathsf{s}(u_2))$.

$$\begin{array}{lll}
\boldsymbol{E}_1 = x_{1,1}\ x_{1,2} & : \texttt{load}\ 1 & : \sigma_1 = id \\
\boldsymbol{E}_2 = x_{1,1}\ x_{1,2}\ x_{1,1} & : \texttt{branch}\ \mathsf{s}\ 7 & : \sigma_2 = id \\
\boldsymbol{E}_3 = \mathsf{s}(x_{3,3})\ x_{1,2}\ x_{3,3} & : \texttt{load}\ 2 & : \sigma_3 = [\mathsf{s}(x_{3.3})/x_{1,1}] \\
\boldsymbol{E}_4 = \mathsf{s}(x_{3,3})\ x_{1,2}\ x_{3,3}\ x_{1,2} & : \texttt{build}\ \mathsf{s}\ 1 & : \sigma_4 = [\mathsf{s}(x_{3.3})/x_{1,1}] \\
\boldsymbol{E}_5 = \mathsf{s}(x_{3,3})\ x_{1,2}\ x_{3,3}\ \mathsf{s}(x_{1,2}) & : \texttt{call}\ add\ 2 & : \sigma_5 = [\mathsf{s}(x_{3.3})/x_{1,1}] \\
\boldsymbol{E}_6 = \mathsf{s}(x_{3,3})\ x_{1,2}\ add(x_{3,3}, \mathsf{s}(x_{1,2})) & : \texttt{return}\ 2 & : \sigma_6 = [\mathsf{s}(x_{3.3})/x_{1,1}] \\
\boldsymbol{E}_7 = x_{1,1}\ x_{1,2}\ x_{1,1} & : \texttt{load}\ 2 & : \sigma_7 = id \\
\boldsymbol{E}_8 = x_{1,1}\ x_{1,2}\ x_{1,1}\ x_{1,2} & : \texttt{return}\ 2 & : \sigma_8 = id
\end{array}$$

A configuration M is well-shaped if all the frames (f, i, ℓ) in M are well-shaped. This condition relies on the parameters used to initialise the frame.

$$\frac{\begin{array}{c} wsh(f, \boldsymbol{\sigma}, \boldsymbol{E}) \qquad match\big((\boldsymbol{\sigma}_i(x_{1,1}), \ldots, \boldsymbol{\sigma}_i(x_{1,ar(f)})), \boldsymbol{u}\big) = \rho \\ \text{if } \boldsymbol{E}_i[j] \text{ is a pattern then } \ell[j] = \rho(\boldsymbol{E}_i[j]) \end{array}}{wsh(f, \boldsymbol{u}, i, \ell)}$$

$$\frac{\begin{array}{c} M \equiv (f_1, i_1, \ell_1) \cdots (f_m, i_m, \ell_m) \qquad wt(M) \\ \boldsymbol{u}_j = arg(M, j) \qquad wsh(f_j, \boldsymbol{u}_j, i_j, \ell_j) \qquad j \in 1..m \end{array}}{wsh(M)}$$

Assume the bytecode of the function f has passed the type and shape verifications. As for type verification, we prove that the shape predicate is invariant under reduction.

Proposition 4. *If $wsh(M)$ and $M \to M'$ then $wsh(M')$.*

The shape verification is particularly well-suited to the analysis of code obtained from the compilation of functional programs, but it may not scale well to optimised code, like the one obtained by the elimination of tail recursive calls. Nonetheless, we can easily define the size verification (see Section 6) without relying on the shape analysis and perform this verification on programs that do not meet the conditions given previously. Hence, we should not see the shape analysis as a required step of our method but rather as an elegant way to define simultaneously the core of our size and termination analyses.

6 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 $q_{\mathsf{c}} : (\mathbb{R}^+)^{ar(\mathsf{c})} \to \mathbb{R}^+$ and $q_f : (\mathbb{R}^+)^{ar(f)} \to \mathbb{R}^+$ are given.

We prove that we can 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 of the frames on the stack.

We assume the bytecode passes the shape verification. Thus for every instruction index i in the segment of the function f, the sequence of expressions $\boldsymbol{E}_i$ and the substitution $\boldsymbol{\sigma}_i$ are determined. We also know $\boldsymbol{h}_i$, the height of the stack at instruction i, as computed during the type verification.

Definition 3. *We say that the size annotations for the function f are correct if the following condition holds for all $i \in 1..|f|$. Assume $\boldsymbol{E}_i = e_1 \cdots e_{\boldsymbol{h}_i}$, then:*

$$\forall j \in 1..\boldsymbol{h}_i \qquad q_f(q_{\boldsymbol{\sigma}_i(x_{1,1})}, \ldots, q_{\boldsymbol{\sigma}_i(x_{1,ar(f)})}) \geqslant q_{e_j} \quad \textit{over } \mathbb{R}^+ \tag{1}$$

In the case of the (compiled) function add, for example, the correctness of the size annotations results from the validity of the inequality: $q_{add}(1 + x_{3,3}, x_{1,2}) \geqslant q_{add}(x_{3,3}, 1 + x_{1,2})$ (from (1) on the expressions obtained for instruction 6).

The complexity of verifying condition (1) depends on the choice of the quasi-interpretations space. This problem has the same complexity as verifying the correction of the quasi-interpretation at the level of the functional language, see Section 2. We also notice that the condition is quite redundant and can be optimised. Next, we show (Corollary 1) that the size of all the values occurring in a configuration during the evaluation of an expression $f(v_1, \ldots, v_n)$ are bounded by the quasi-interpretation of $f(v_1, \ldots, v_n)$. This follows from the definition of a new predicate $wsz(M)$ and a related invariant.

$$\frac{\begin{array}{c} wsh(f, \boldsymbol{\sigma}, \boldsymbol{E}) \qquad match((\boldsymbol{\sigma}_i(x_{1,1}), \ldots, \boldsymbol{\sigma}_i(x_{1,ar(f)})), \boldsymbol{u}) = \rho \\ \boldsymbol{E}_i = e_1 \cdots e_{\boldsymbol{h}_i} \qquad \ell = v_1 \cdots v_{\boldsymbol{h}_i} \qquad q_{\rho(e_j)} \geqslant q_{v_j} \qquad j \in 1..\boldsymbol{h}_i \end{array}}{wsz(f, \boldsymbol{u}, i, \ell)}$$

$$\frac{\begin{array}{c} M \equiv (f_1, i_1, \ell_1) \ldots (f_m, i_m, \ell_m) \qquad wsh(M) \qquad \boldsymbol{u}_j = arg(M, j) \\ wsz(f_j, \boldsymbol{u}_j, i_j, \ell_j) \qquad q_{f_k(\boldsymbol{u}_k)} \geqslant q_{f_{k+1}(\boldsymbol{u}_{k+1})} \qquad j \in 1..m \qquad k \in 1..m-1 \end{array}}{wsz(M)}$$

Assume the bytecode of the function f has passed the type and shape verifications. As for the type and shape verifications, we prove that the size predicate is invariant under reduction.

Proposition 5. *If $wsz(M)$ and $M \to M'$ then $wsz(M')$.*

Corollary 1. *Assume that all the functions in the program are well-sized. If the expression $f(v_1, \ldots, v_n)$ is well-typed and $(f, 1, v_1 \cdots v_n) \xrightarrow{*} M \cdot (g, i, \ell)$ then $|v| \leqslant q_{f(v_1, \ldots, v_n)}$ for all the values v occurring in ℓ.*

Proof. By definition, $wsz(f, 1, v_1 \cdots v_n)$. By proposition 5, it follows that $wsz(M \cdot (g, i, \ell))$. Let $\boldsymbol{u} = (u_1, \ldots, u_k)$ be the parameters used in the initialization of the top frame: $\boldsymbol{u} = arg(M \cdot (g, i, \ell), |M| + 1)$. Since the configuration is well-sized, we have $wsz(g, \boldsymbol{u}, i, \ell)$ and there is a substitution ρ such that: (c1) $q_{f(v_1, \ldots, v_n)} \geqslant q_{g(u_1, \ldots, u_k)}$, (c2) $\rho \circ \boldsymbol{\sigma}_i(x_{1,j}) = u_j$ for all $j \in 1..k$, and (c3) $wsh(g, \boldsymbol{\sigma}, \boldsymbol{E})$ and $\boldsymbol{E}_i = e_1 \cdots e_n$ and $q_{\rho(e_j)} \geqslant q_{\ell[j]}$ for $j \in 1..n$.

By definition, the size annotations in the bytecode are correct, which means that by the verification condition (1) we have: $q_{g(\boldsymbol{\sigma}_i(x_{1,1}),\ldots,\boldsymbol{\sigma}_i(x_{1,k}))} \geqslant q_{e_j}$ for all $j \in 1..n$. We conclude:

$$\begin{aligned} q_{f(v_1,\ldots,v_n)} &\geqslant q_{g(u_1,\ldots,u_k)} && \text{by (c1)} \\ &= q_{g(\rho\circ\boldsymbol{\sigma}_i(x_{1,1}),\ldots,\rho\circ\boldsymbol{\sigma}_i(x_{1,k}))} && \text{by (c2)} \\ &\geqslant q_{\rho(e_j)} && \text{by (1) and monotonicity} \\ &\geqslant q_v && \text{by (c3)} \\ &\geqslant |v| && q_v \text{ is a quasi-interpretation} \end{aligned}$$ □

We can use corollary 1 to give a rough estimate of the size of the frames occurring in a configuration. Assume that all the functions in the program are well-sized and consider a frame (g, i, ℓ) occurring in a configuration reached from the evaluation of the expression $f(v_1, \ldots, v_n)$.

From the type verification, we obtain a bound h_g on the length of the stack ℓ (for the function add in our examples we have $h_{add} = 4$). We may define the size of a frame as the sum of the size of the values in ℓ added to h_g — the quantity h_g makes allowance for the presence of constants stored in the stack and we neglect the space needed for storing the function identifier and the program counter. Hence the size of the frame (g, i, ℓ) is less than $h_g \cdot (q_{f(v_1,\ldots,v_n)} + 1)$. Likewise, if we define the size of a configuration M as the sum of the frames occurring in M, then we can bound the size of M by the expression $h_m \cdot (q_{f(v_1,\ldots,v_n)} + 1) \cdot l$, where l is the number of frames in M and h_m is the maximum of the h_g for all the functions g in the program.

In the next section, we use information obtained from a termination analysis to bound the number of frames that may appear in a reachable configuration. As a result, we obtain a bound on the maximal space needed for the evaluation of the expression $f(v_1, \ldots, v_n)$ (see Corollary 3). Moreover, if we can prove termination by lexicographic order, then this bound can be expressed as a polynomial expression on the values $|v_1|, \ldots, |v_n|$.

7 Termination Verification

In this section, we adapt recursive path orderings, a popular technique for checking termination (see, *e.g.*, [6]), to prove termination of the evaluation of the virtual machine. We suppose that the shape verification of the code succeeds. We assume given a pre-order $\geqslant_\Sigma$ on the function symbols so that $f =_\Sigma g$ implies $ar(f) = ar(g)$. Recursive path ordering conditions force $f \geqslant_\Sigma g$ whenever f may call g, and $f =_\Sigma g$ whenever f and g are mutually recursive. The pre-order $\geqslant_\Sigma$ 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 recall that in the recursive path ordering one associates a *status* with each symbol specifying how its arguments have to be compared. It is required that if $f =_\Sigma g$ then f and g have the same status. Here we suppose that the status of every function symbol is lexicographic and that the status of every constructor symbol is the product. We denote with $>_l$ the induced path order. Note that on values $v >_l v'$ if and only if v embeds homomorphically v'. Hence, $v >_l v'$ implies $|v| > |v'|$.

The technical development resembles the one for the value size verification. First, we have to define when the termination annotations given with the bytecode are correct.

Definition 4. *We say that the termination annotations for the function f are correct if the following condition holds for all $i \in 1..|f|$. Assume $\boldsymbol{E}_i = e_1 \cdots e_{\boldsymbol{h}_i}$, then:*

$$\forall j \in 1..\boldsymbol{h}_i \qquad f\big(\boldsymbol{\sigma}_i(x_{1,1}), \ldots, \boldsymbol{\sigma}_i(x_{1,ar(f)})\big) >_l e_j \tag{2}$$

For the function add, the correctness of the termination annotations results primarily from the validity of the relation $add(\mathsf{s}(x_{3,3}), x_{1,2}) >_l add(x_{3,3}, \mathsf{s}(x_{1,2}))$ (for the lexicographic path ordering). Next, we introduce a predicate ter (for terminating) on well-shaped configurations M. As expected, the termination predicate is an invariant.

$$\frac{\begin{array}{c} wsh(f, \boldsymbol{\sigma}, \boldsymbol{E}) \qquad match\big((\boldsymbol{\sigma}_i(x_{1,1}), \ldots, \boldsymbol{\sigma}_i(x_{1,ar(f)})), \boldsymbol{u}\big) = \rho \\ \boldsymbol{E}_i = e_1 \cdots e_{\boldsymbol{h}_i} \qquad \ell = v_1 \cdots v_{\boldsymbol{h}_i} \qquad \rho(e_j) \geqslant_l v_j \qquad j \in 1..\boldsymbol{h}_i \end{array}}{ter(f, \boldsymbol{u}, i, \ell)}$$

$$\frac{\begin{array}{c} M \equiv (f_1, i_1, \ell_1) \ldots (f_m, i_m, \ell_m) \qquad wsh(M) \qquad \boldsymbol{u}_j = arg(M, j) \\ ter(f_j, \boldsymbol{u}_j, i_j, \ell_j) f_k(\boldsymbol{u}_k) >_l f_{k+1}(\boldsymbol{u}_{k+1}) \qquad j \in 1..m \qquad k \in 1..m-1 \end{array}}{ter(M)}$$

Proposition 6. *If $ter(M)$ and $M \to M'$ then $ter(M')$.*

Corollary 2. *Assume that all the functions in the program have correct termination information (see Definition 4). Then the execution of a well-typed frame $(f, 1, v_1 \cdots v_n)$ terminates.*

Proof. We define a well-founded order on well-formed configurations that is compatible with the evaluation of the machine. If i is the index of an instruction in the code of f, let $acc(i)$ denotes the number of instructions reachable from i in the flow graph E_f. Since the flow graph is a tree, whenever we increment the counter or jump to another instruction this value decreases. Let $T = (T_\Sigma, >_l)$ be the collection of values with the lexicographic path order. It is well known that this is a well-founded order. Then consider $T \times \mathbb{N}$ with the lexicographic order from left to right. Again this is a well-founded order. Finally, consider $\mathcal{M}(T \times \mathbb{N})$ the finite multisets over $T \times \mathbb{N}$ with the induced well-founded order. We associate with a configuration $M \equiv (f_1, i_1, \ell_1) \cdots (f_m, i_m, \ell_m)$ the measure $\mu(M) = \{\!|(f_1\, arg(M,1), acc(i_1)-1), \ldots, (f_{m-1}\, arg(M, m-1), acc(i_{m-1})-1), (f_m\, arg(M,m), acc(i_m))|\!\}$.

Then, by case analysis, we check that all the reduction rules decrease this measure. This proof is by case analysis on the instruction $f_m[i_m]$. Assume $f_m[i_m] = \texttt{call}\ g\ n$. An element $(f(\boldsymbol{v}), i)$ of the multiset is replaced by the two elements $(f(\boldsymbol{v}), i-1)$ and $(g(\boldsymbol{u}), acc(1))$, where $f(\boldsymbol{v}) >_l g(\boldsymbol{u})$ (by the invariant ter) so that, with respect to the lexicographic order: $(f(\boldsymbol{v}), i) > (f(\boldsymbol{v}), i-1)$ and $(f(\boldsymbol{v}), i) > (g(\boldsymbol{u}), acc(1))$. In the other cases, an element $(f(\boldsymbol{v}), i)$ is either removed or replaced by $(f(\boldsymbol{v}), j)$ with $i > j$, as needed. □

As observed in [5], 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 3. *Suppose that the quasi-interpretations are bound by polynomials and that the value size and termination verifications of the bytecode succeeds. Then there exists a polynomial* q *such that every execution starting from a frame* $(f, 1, v_1 \cdots v_n)$ *(terminates and) runs in space bound by* $q(|v_1|, \ldots, |v_n|)$.

Proof. Note that if $f(\boldsymbol{v}) >_l g(\boldsymbol{u})$ then either $f >_\Sigma g$ or $f =_\Sigma g$ and $\boldsymbol{v} >_l \boldsymbol{u}$. In a sequence $f_1(\boldsymbol{v}_1) >_l \cdots >_l f_m(\boldsymbol{v}_m)$, 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 analyse 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. □

From the type verification we obtain a bound h_m on the length of the stacks (for the function add in our examples we have $h_m = 4$). From the size verification we obtain a bound $q_m = q_{f(v_1,\ldots,v_n)}$ on the size of every value occurring in a stack (in our example $q_m = |v_1| + |v_2|$). Finally, the termination analysis provides a bound on the maximal number of frames. A crude analysis gives at most q_m^k frames, where k is the greatest arity among the functions occurring during the execution. Hence, the size needed for the execution of a correct program on the initial configuration $(f, 1, v_1 \cdots v_n)$ is bounded by the product $h_m \cdot (q_m + 1) \cdot q_m^k$. We may improve this bound using a finer analysis of the (proof of correctness of the) termination annotations. In the case of add, for example, we remark that the size of the first parameter decreases at every call — there could be at most $|v_1|$ frames in a reachable configuration — and therefore we may derive the stricter bound $4 \cdot (|v_1| + |v_2| + 1) \cdot |v_1|$ instead of $4 \cdot (|v_1| + |v_2| + 1) \cdot (|v_1| + |v_2|)^2$.

8 Conclusion and Related work

The problem of bounding the size of the memory needed for executing a program has already attracted considerable attention but few works have addressed this problem at the level of the bytecode.

Most work in the literature on bytecode verification tends to guarantee the integrity of the execution environment. Work on resource bounds is carried on in the MRG project [17]. 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 [14] 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.

We have shown how to perform type, size, and termination verifications at the level of the bytecode running on a simple stack machine. We believe that the choice of a simple set of bytecode instructions has a pedagogical value: 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 formalising our virtual machine and the related invariants in the COQ proof assistant. We are also experimenting with the automatic synthesis of annotations at the source code level and with their verification at byte code level. Moreover, we plan to refine the predictions on the space needed for the execution of a program by referring to an optimised implementation of the virtual machine.

References

1. M. Abadi and R. Stata. A type system for Java bytecode subroutines. In *Proc. POPL*, 1998.
2. R. Amadio. Max-plus quasi-interpretations. In *Proc. TLCA*, Springer LNCS 2701, 2003.
3. R. Amadio, S. Coupet-Grimal, S. Dal Zilio and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. Research Report LIF 17-2004.
4. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.
5. G. Bonfante, J.-Y. Marion and J.-Y. Moyen. On termination methods with space bound certifications. In *Proc. Perspectives of System Informatics*, Springer LNCS 2244, 2001.
6. F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
7. A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II*, North Holland, 1965.
8. M. Hofmann. The strength of non size-increasing computation. In *Proc. POPL*, 2002.
9. N. Jones. *Computability and complexity, from a programming perspective*. MIT-Press, 1997.
10. G. Kildall, A unified approach to global program optimization. In *Proc. POPL*, 1973.
11. D. Leivant. Predicative recurrence and computational complexity i: word recurrence and poly-time. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser, 1994.
12. T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison-Wesley, 1999.
13. J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. Habilitation à diriger des recherches, Université de Nancy, 2000.
14. J.-Y. Marion, J.-Y. Moyen. *Termination and resource analysis of assembly programs by Petri Nets*. Technical Report, Université de Nancy, 2003.
15. 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.
16. G. Necula. Proof carrying code. In *Proc. POPL*, 1997.
17. D. Sannella. Mobile resource guarantee. IST-Global Computing research proposal, U. Edinburgh, 2001. `http://www.dcs.ed.ac.uk/home/mrg/`.