Resource Bound Certification for a Tail-Recursive Virtual Machine*

Silvano Dal Zilio[1] and Régis Gascon[2]

[1] LIF, CNRS and Université de Provence, France
[2] LSV, CNRS and ENS Cachan, France

Abstract. We define a method to statically bound the size of values computed during the execution of a program as a function of the size of its parameters. More precisely, we consider bytecode programs that should be executed on a simple stack machine with support for algebraic data types, pattern-matching and tail-recursion. Our size verification method is expressed as a static analysis, performed at the level of the bytecode, that relies on machine-checkable certificates. We follow here the usual assumption that code and certificates may be forged and should be checked before execution.
Our approach extends a system of static analyses based on the notion of quasi-interpretations that has already been used to enforce resource bounds on first-order functional programs. This paper makes two additional contributions. First, we are able to check optimized programs, containing instructions for unconditional jumps and tail-recursive calls, and remove restrictions on the structure of the bytecode that was imposed in previous works. Second, we propose a direct algorithm that depends only on solving a set of arithmetical constraints.

1 Introduction

Bytecode programs are a form of intermediate code commonly used by language implementors when programs should be distributed and run on multiple platforms. Because of its advantages on performance and portability, many programming languages are actually compiled into bytecode. Java and Microsoft C# are representative examples, but bytecode compilers can also be found for less conventional languages, such as O'Caml, Perl or PHP. On the downside, bytecode typically stands at an abstraction level in between (high-level) source code and machine code: it is usually more compact and closer to the computer architecture than program code that is intended for "human consumption". Therefore, it is necessary to devise specific verification methods to guarantee properties at the bytecode level. For instance, to ensure the safety of executing newly loaded code, virtual machines generally rely on machine-checkable certificates that the program will comply with user-specific requirements. The interest of verification of such properties at the bytecode level is now well understood, see for example [14,17].

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

As networked and mobile applications become more and more pervasive, and with the lack of third parties in control of trust management (like e.g. frameworks based on code signing), security appears as a major issue. Initial proposals for securing bytecode applications have focused on the integrity of the execution environment, such as the absence of memory faults and access violations. In this paper, we consider another important property, namely certifying bounds on the resources needed for the execution of the code. This problem naturally occurs when dealing with mobile code, for example to prevent denial of service attacks, in which the virtual machine is starved of memory by the execution of a malicious program. More precisely, we define a method to statically bound the size of values computed during the execution of a program. The size-bound obtained by this method is expressed as a function of the size of the parameters of the program (actually as a polynomial expression) and has several uses. For instance, together with an analysis that bounds the maximal number of stacks in the evaluation of a program, it gives an overall bound on the memory space needed by the virtual machine. This size-bound can also be used with automatic memory management techniques, e.g. to bound the physical size of regions in region-based systems [18].

We consider bytecode programs that should be executed on a simple stack machine with support for algebraic data types, pattern-matching and tail-recursion. The bytecode language can be the target of the compilation of a simply-typed, first-order functional language. We hint at this functional source language in several places but, since all our results are stated on the bytecode, we do not need to define it formally here (see [1] for a definition). Our *size verification* algorithm is expressed as a static analysis relying on certificates that can be verified at load time. We follow here the usual assumption that code and certificates may be forged by a malicious party. In particular, they do not have to result from the compilation of legit programs. Standard bytecode verification algorithms build for each instruction an abstract representation of the stack. This information typically consists of the types of the values on the stack when the instruction is executed. In a nutshell, the size verification algorithm builds for each instruction an abstract bound on the size of the values in the stack. In our case the bound is a polynomial expression. It also builds proof obligations that the bounds decrease throughout program execution.

Our method generalizes (and lift some of the restrictions) an approach designed for first-order functional languages [1] that relies on a combination of standard techniques for term rewriting systems with a static analysis based on the notion of quasi-interpretation. Similar analyses were also used to deal with systems of concurrent, interactive threads communicating via a shared memory [2]. This paper makes two additional contributions. First, we are able to check programs containing instructions for unconditional jumps and tail-recursive calls, and remove restrictions on the structure of the bytecode that was imposed in these two initial works. These two instructions are essential in the optimization of codes obtained from the compilation of functional programs. They are also required if we need to compile procedural languages. Second, we propose a direct algorithm that depends only on solving a set of arithmetical constraints. Indeed, the size verifications defined in [1,2] are based on a preliminary *shape analysis* which builds, for each bytecode instruction, a sequence of first-order expressions representing the shape of the values in the stack (e.g. it may give the top-most constructors). While the shape verification is well-suited to the analysis of “functional code”, it does not scale to programs containing tail recursive calls.

Another result of this work is educational: we present a minimal but still relevant scenario in which problems connected to bytecode verification can be effectively studied. For instance, our virtual machine is based on a set of 8 instructions, a number that has to be compared with the almost 200 opcodes used in the Java Virtual Machine (JVM). We believe that the simplicity of the virtual machine and the bytecode verifiers defined in this paper make them suitable for teaching purposes. (Actually, we have already used them for projects in compiler design classes.)

The paper is organized as follows. Section 2 defines a simple virtual machine and a bytecode language built from a minimal set of instructions. In Section 3, we introduce the notion of quasi-interpretations and define our size verification method. This verification assumes that constructors and function symbols in the bytecode are annotated with suitable functions to bound the size of the values on the stack. Before concluding, we study the complexity of checking the constraints generated during the size analysis. In particular, we show that their satisfiability can be reduced to checking the sign of a polynomial expression.

2 Virtual Machine

We define a simple set of bytecode instructions and a related stack machine. A program is composed of a list of mutually recursive type definitions followed by a list of function definitions. In our setting, a function is a sequence of bytecode instructions. Unlike traditional virtual machines that operate on literal values, such as bytes or floating point numbers, we consider values taken from an arbitrary set of inductive types.

A value v is a term built from a finite set of constructors, ranged over by $\mathsf{c}, \mathsf{d}, \ldots$ The *size* of v, denoted $|v|$, is 0 if v is a constant (a constructor of arity 0) and $1 + \Sigma_{i \in 1..n} |v_i|$ if v is of the form $\mathsf{c}(v_1, \ldots, v_n)$.

We consider a fixed set of type identifiers $t, t', \ldots$ where each identifier is associated to a unique type definition of the form $t = \ldots \mid \mathsf{c}\ \mathit{of}\ t_1 * \cdots * t_n \mid \ldots$ For instance, we can define the type *nat* of natural numbers in unary format and the type *bw* of binary words:

$$\mathit{nat} = \mathsf{z} \mid \mathsf{s}\ \mathit{of}\ \mathit{nat}\ , \qquad\qquad \mathit{bw} = \mathsf{E} \mid \mathsf{O}\ \mathit{of}\ \mathit{bw} \mid \mathsf{I}\ \mathit{of}\ \mathit{bw}\ .$$

For instance, the values $\mathsf{s}(\mathsf{s}(\mathsf{z}))$ of type *nat* and $\mathsf{O}(\mathsf{I}(\mathsf{O}(\mathsf{E})))$ of type *bw* stand for the number 2. We will often use the type *nat* in our examples since functions manipulating natural numbers can be interpreted as an abstraction of functions manipulating finite lists (e.g. addition is related to list catenation).

For the sake of simplicity, we suppose that the code and type of functions is fixed and known in advance. Hence we consider a fixed set of constructor and function names. We suppose that every constructor is declared with its functional type $(t_1, \ldots, t_n) \to t$ and we denote $ar(\mathsf{c})$ the arity of the constructor c. Similar types can be either assigned or inferred for functions. We adopt the notation ε for the empty sequence and $\ell \cdot \ell'$ for the catenation of two sequences. The expression $|\ell|$ denotes the length of ℓ and $\ell[i]$ denotes the i^{th} element in ℓ. When the length is given by the context, we will sometimes use the vectorial notation $\vec{v}$ to represent the sequence $(v_1, \ldots, v_n)$. In the following, we equate a function identifier f with the sequence of instructions of its body code and thus write $f[i]$ for the i^{th} instruction in f.

The virtual machine is built around three components: (1) a *configuration* M that 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 a sequence of frames and we write $M \to M'$ if M reduces to M' using one of the transformation rules described by the table below.

The most important operation of the virtual machine corresponds to function calls. The execution of a function call is represented by a *frame*, that is a triple $(f, pc, \ell)_\rho$ made of a function identifier f, the value of the program counter pc (a natural number in $1..|f|$) and an *evaluation stack* ℓ. A stack is a sequence of values that is used to store both the parameters of the call as well as the "local values" computed during the life span of the frame. Hence the stack partially plays the role devoted to registers in traditional architectures. The annotation ρ is used to keep trace of the call that initiated the frame and has no operational meaning (it is only used to validate our size verification method). We refine the system of annotations in Section 2.1.

Bytecode Interpreter: $M \to M'$

(Load)

$$\frac{f[pc] = \mathtt{load}\ i \quad pc < |f| \quad \ell[i] = v}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc+1, \ell \cdot v)_\rho}$$

(Build)

$$\frac{f[pc] = \mathtt{build}\ \mathsf{c}\ n \quad pc < |f| \quad ar(\mathsf{c}) = n \qquad \ell = \ell' \cdot (v_1, \ldots, v_n) \quad v_o = \mathsf{c}(v_1, \ldots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc+1, \ell' \cdot v_o)_\rho}$$

(BranchThen)

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

(BranchElse)

$$\frac{f[pc] = \mathtt{branch}\ \mathsf{c}\ j \quad 1 \leqslant j \leqslant |f| \qquad \ell = \ell' \cdot \mathsf{d}(\ldots) \quad \mathsf{c} \neq \mathsf{d}}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, j, \ell)_\rho}$$

(Call)

$$\frac{f[pc] = \mathtt{call}\ g\ n \quad pc < |f| \quad ar(g) = n \qquad \ell = \ell' \cdot \ell'' \quad \ell'' = (v_1, \ldots, v_n) \quad \rho' = g(v_1, \ldots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, pc, \ell')_\rho \cdot (g, 1, \ell'')_{\rho'}}$$

(Jump)

$$\frac{f[pc] = \mathtt{jump}\ j\ n \quad 1 \leqslant j \leqslant |f| \qquad \ell = \ell' \cdot \ell'' \quad \ell'' = (v_1, \ldots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (f, j, \ell'')_\rho}$$

(TCall)

$$\frac{f[pc] = \mathtt{tcall}\ g\ n \quad pc < |f| \quad ar(g) = n \qquad \ell = \ell' \cdot \ell'' \quad \ell'' = (v_1, \ldots, v_n)}{M \cdot (f, pc, \ell)_\rho \to M \cdot (g, 1, \ell'')_\rho}$$

(Stop)

$$\frac{f[pc] = \mathtt{stop}}{M \cdot (f, pc, \ell)_\rho \to \mathit{error}}$$

(Return)

$$\frac{f[pc] = \texttt{return} \quad \ell = \ell'' \cdot v_o}{M \cdot (g, pc', \ell')_{\rho'} \cdot (f, pc, \ell)_\rho \rightarrow M \cdot (g, pc' + 1, \ell' \cdot v_o)_{\rho'}}$$

We give an informal description of the bytecode language. Let ℓ be the stack of the current frame, i.e. the frame on the top of the current configuration. The instruction `load` i takes a copy of the i^{th} value of ℓ and puts it on the top of the stack (i.e. it is equivalent to a register load). New values may be created using the instruction `build c` n, where `c` is a constructor of arity n. When executed, the n values $v_1, \ldots, v_n$ on top of ℓ are replaced by $\mathtt{c}(v_1, \ldots, v_n)$. The instruction `branch c` j implements a conditional jump on the shape of the value v found on top of ℓ. If v is of the form $\mathtt{c}(v_1, \ldots, v_n)$ then the top of the stack is replaced by the n values $v_1, \ldots, v_n$ (rule BranchThen). Otherwise, the stack is left unchanged and the execution jumps to position j in the code, with $j \in 1..|f|$ (rule BranchElse).

Function calls are implemented by the instruction `call` f n, where n is the arity of f. 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` discards the current frame and returns the value on top of the caller's stack; `stop` finishes the execution and returns an error code. Finally, the instruction `jump` j n is an unconditional jump, similar to a goto statement, whereas `tcall` g n is similar to a call instruction, except that the current frame is used to evaluate the call to g. These two instructions are used to share common code between functions and to efficiently compile tail recursion (when call instructions are immediately followed by a return). This is essential because many programming idioms depend heavily on recursion. For example, the Scheme language reference explicitly requires tail recursion to be recognized and automatically optimized by a compiler.

The reduction relation $M \rightarrow M'$ is deterministic and uses a special state of the memory, *error*, that denotes the empty configuration ε. The empty state cannot be reached during an execution that does not raise an error (executes a `stop` instruction). A "correct" execution starts with a single frame $M_\iota = (f, 1, \ell)_{f(\ell)}$, where $\ell = (v_1, \ldots, v_n)$, and ends with a configuration of the form $M_o = (f, pc, \ell' \cdot v_0)_{f(\ell)}$, where $1 \leqslant pc \leqslant |f|$ and $f[pc] = \texttt{return}$. We name the configuration M_ι a *call to* $f(v_1, \ldots, v_n)$ and M_o the *result* of evaluating M_ι and we write $M_o \searrow v_0$. The others cases of blocked configurations are runtime errors.

2.1 Control Flow Graphs and Well-Formedness

Before giving examples of bytecode programs, we define the notions of *control flow graph* (CFG) and *checkpoints* of a function f. The CFG of f is the smallest directed graph $(\{1, \ldots, |f|\}, E)$ such that for all node $i \in 1..|f|$ the edge $(i, i+1)$ is in E if $f[i]$ is a `load`, `build`, `call` or `branch` instruction and (i, j) is in E if $f[i]$ is `branch c` j or `jump` j n. Nodes that are the target of a `jump` or `branch` instruction can have several immediate predecessors. We call such nodes the checkpoints of f. The first instruction of a function is also a checkpoint.

We associate to every node i of f the node $PC_i \in 1..|f|$ that is the only checkpoint dominating the node i in the CFG of f: it is the first checkpoint encountered from i when moving backward in the CFG. We say that PC_i is the checkpoint of i and we have $PC_i = i$ iff i is a checkpoint. By construction, every node of a CFG is associated to a unique checkpoint and there is a unique path between PC_i and i without cycles. We also define the predicate $Control_f(i)$ which is true iff i is a checkpoint of f.

We refine the semantics of the virtual machine to take into account checkpoints in frame annotation. We store in the annotations the state of the execution stack when we pass a new checkpoint (together with the state of the stack when the frame is initialized). This improvement is needed for our size analysis but the dynamic semantics of the machine does not need any change. The only difference is in the frame annotation ρ that is now of the form $(g(\ell_o), i, \ell_c)$ where $g(\ell_o)$ is the "call" used to initialize the frame, i is the last checkpoint encountered and ℓ_c is the state of the execution stack when we passed i. For each transition $M \cdot (f, pc, \ell)_\rho \rightarrow M' \cdot (f', pc', \ell')_{\rho'}$ of the new relation, we have $\rho' = (\ldots, pc', \ell')$ if pc' is a checkpoint of the function f' and $\rho' = \rho$ otherwise. Note that the evaluation of a `call` or `tcall` instruction (the only case in which $f \neq f'$) always leads to a configuration where the program counter of the last frame is equal to 1, i.e. is a checkpoint.

Annotated Semantics

(Regular)
$$\frac{M \cdot (f, pc, \ell)_{g(\ell_o)} \rightarrow M \cdot (f, pc', \ell')_{g(\ell_o)} \quad Control_f(pc') \text{ is false} \quad \rho = (g(\ell_o), i, \ell_c)}{M \cdot (f, pc, \ell)_\rho \rightarrow M \cdot (f, pc', \ell')_\rho}$$

(Checkpoint)
$$\frac{M \cdot (f, pc, \ell)_{g(\ell_o)} \rightarrow M' \cdot (f', pc', \ell')_{h(\ell_1)} \quad Control_f(pc') \text{ is true} \quad \rho = (g(\ell_o), i, \ell_c)}{M \cdot (f, pc, \ell)_\rho \rightarrow M' \cdot (f', pc', \ell')_{(h(\ell_1), pc', \ell')}}$$

Examples. Our first example is the function $dble : nat \rightarrow nat$, that doubles its parameter. A possible specification of this function using a functional syntax could be $dble(\mathsf{z}) = \mathsf{z}$ and $dble(\mathsf{s}(x)) = \mathsf{s}(\mathsf{s}(dble(x)))$ (actually, the code given below is the result of compiling this functional program as in [1]). In the following, we display the index of each instruction next to its code and underline the indices of checkpoints.

$dble =$	$\underline{1}$: `load` 1	5 : `branch s` 10	9 : `return`
	2 : `branch z` 5	6 : `call` *dble* 1	10 : `stop`
	3 : `build z` 0	7 : `build s` 1	
	4 : `return`	8 : `build s` 1	

The evaluation of the call to $dble(\mathsf{s}(v))$ gives the following reductions, where w stands for the result of the call to $dble(v)$ (we do not write annotations).

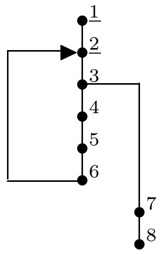
$$\begin{array}{l}(dble, 1, (\mathsf{s}(v))) \rightarrow (dble, 2, (\mathsf{s}(v), \mathsf{s}(v))) \rightarrow (dble, 5, (\mathsf{s}(v), \mathsf{s}(v))) \rightarrow (dble, 6, (\mathsf{s}(v), v)) \\ \rightarrow (dble, 6, (\mathsf{s}(v))) \cdot (dble, 1, (v)) \rightarrow \cdots \rightarrow (dble, 6, (\mathsf{s}(v))) \cdot (dble, 9, (w)) \\ \rightarrow (dble, 7, (\mathsf{s}(v), w)) \rightarrow (dble, 8, (\mathsf{s}(v), \mathsf{s}(w))) \rightarrow (dble, 9, (\mathsf{s}(v), \mathsf{s}(\mathsf{s}(w)))) \searrow \mathsf{s}(\mathsf{s}(w))\end{array}$$

¿ From this simple example we can already see that first-order functional programs admit a direct compilation into our bytecode: every function is compiled into a segment of instructions where pattern matching is represented by a nesting of `branch` instructions. In particular the CFG of a compiled program is a tree. Because our virtual machine does not allow to store code closures, we cannot directly support subroutines or higher-order functions. We plan to study these extensions in future works.

We can simplify our first example following two distinct directions. We obtain an equivalent function by noticing that a natural number that is not of the form `s`(...) is necessarily `z`. Hence we can discard a useless `branch` instruction. A better optimization is obtained with the function *tdble* : *nat* → *nat*: we duplicate the parameter (with a `load` instruction) and use it as an accumulator, giving the opportunity to use a `jump` instruction. Finally, the function *xdble* : *nat* → *nat* is an example of malicious code that loops and computes unbounded values.

inst. # :	*dble*	*xdble*	*tdble*
1 :	`load` 1	`load` 1	`load` 1
2 :	`branch s` 6	`build s` 1	`load` 1
3 :	`call` *dble* 1	`build s` 1	`branch s` 7
4 :	`build s` 1	`call` *xdble* 1	`load` 2
5 :	`build s` 1	`return`	`build s` 1
6 :	`return`		`jump` 2 2
7 :			`load` 2
8 :			`return`

CFG of *tdble*



Our last example is the function *sum* : *nat* → *nat* such that a call to $sum(x)$ computes the value of the expression $x + (x - 1) + \cdots + 1$. The function *sum* is interesting because it is a non trivial example mixing recursive calls and "superlinear" size computations: our size verification can be used to prove that the size of the result is bound by $\frac{1}{2}|x|(|x| + 1)$. The definition of *sum* makes use of the function *add* : $(nat, nat) \to nat$ that tallies up its two parameters.

sum =	$\underline{1}$: `load` 1	3 : `call` *sum* 1	5 : `return`
	2 : `branch s` 5	4 : `call` *add* 2	

add =	$\underline{1}$: `branch s` 5	4 : `load` 2	7 : `return`
	2 : `load` 1	5 : `jump` 1 2	
	3 : `build s` 1	6 : `load` 1	

Well-Typed Programs. We define a type verification that associates to every bytecode instruction an abstraction of the stack when it is executed. In our case, an abstract state T is a sequence of types $(t_1, \ldots, t_n)$ that matches the types of the values in the stack. We say that a stack ℓ has type T, and we note $\ell : T$, if $\ell = (v_1, \ldots, v_n)$ where v_i is a value of type t_i for all $i \in 1..n$. The type of a function f is a sequence of length $|f|$ of type stacks and a well-typed function is a sequence of *well-typed instructions*. The notion of well-typed instruction is formally defined by means of the relation $wt_i(f, \vec{T})$, defined below. For example, if $f[i] = \texttt{load}\ k$ and if the type of $f[i]$ is $T_i = (t_1, \ldots, t_n)$, with $n \geqslant k$, then the

Table 1. Type Analysis ($wt_i(f, \vec{T})$)

Assume $f : (t_1, \ldots, t_n) \to t_0$. Case $f[i]$ of:

(`load` k) then $wt_i(f, \vec{T})$ is true iff $i < |f|$, $T_i[k] = t_k$ and $T_{i+1} = T_i \cdot t_k$;

(`build` `c` m) let `c` $: (t'_1, \ldots, t'_m) \to t'_0$, then $wt_i(f, \vec{T})$ is true iff $i < |f|$, $T_i = T \cdot (t'_1, \ldots, t'_m)$ and $T_{i+1} = T \cdot t'_0$;

(`branch` `c` j) let `c` $: (t'_1, \ldots, t'_m) \to t'_0$, then $wt_i(f, \vec{T})$ is true iff $i < |f|$, $j \in 1..|f|$, $T_i = T \cdot t'_0$, $T_{i+1} = T \cdot (t'_1 \ldots t'_m)$ and $T_j = T_i$;

(`call` g m) let $g : (t'_1, \ldots, t'_m) \to t'_0$, then $wt_i(f, \vec{T})$ is true iff $i < |f|$, $T_i = T \cdot (t'_1 \ldots t'_m)$ and $T_{i+1} = T \cdot t'_0$;

(`tcall` g m) let $g : (t'_1, \ldots, t'_m) \to t'_0$, then $wt_i(f, \vec{T})$ is true iff $i < |f|$, $t_0 = t'_0$, $T_i = T \cdot (t'_1, \ldots, t'_m)$ and $T_{i+1} = T \cdot t_0$;

(`jump` j m) then $wt_i(f, \vec{T})$ is true iff $1 \leqslant j \in 1..|f|$, $T_i = T \cdot (t'_1 \ldots t'_m)$ and $T_j = (t'_1 \ldots t'_m)$;

(`return`) then $wt_i(f, \vec{T})$ is true iff $T_i = T \cdot t_0$;

(`stop`) Then $wt_i(f, \vec{T})$ is true.

type of $f[i+1]$ should be equal to $(t_1, \ldots, t_n, t_k)$. (The abstract state T_i gives the type of values in the stack "before" the execution of instruction i.) A program is well-typed if all its functions are well-typed: a sequence $\vec{T}$ is a *valid abstract execution* of the function $f : (t_1, \ldots, t_n) \to t_0$, denoted $wt(f, \vec{T})$, if and only if $T_1 = (t_1 \ldots t_n)$ and $wt_i(f, \vec{T})$ for all $i \in 1..|f|$. The definition of $wt_i(f, \vec{T})$ is by case analysis on the instruction $f[i]$, see Table 1.

We can define from the predicate wt an algorithm that computes a valid type for a function f if it exists, e.g. using Kildall's algorithm [16]. (We can view type verification as a kind of symbolic execution on stacks of types.) Moreover, we can prove that if the CFG is a connected graph then there is at most one valid type. Then we can assign to every instruction of f the size of its stack, and to every element of that stack a single type. As an example, we give the type inferred for the function $tdble : nat \to nat$.

$\underline{1}$: `load` 1	(nat)	5 : `build` `s` 1	(nat, nat, nat, nat)
$\underline{2}$: `load` 1	(nat, nat)	6 : `jump` 2 2	(nat, nat, nat, nat)
3 : `branch` `s` 7	(nat, nat, nat)	7 : `load` 2	(nat, nat, nat)
4 : `load` 2	(nat, nat, nat)	8 : `return`	(nat, nat, nat, nat)

We can prove that the execution of a well-typed program never fails. For example, We can prove a subject reduction property and follow an approach similar to the one used in Section 3 to prove the validity of our size analysis. Due to the limited amount of space available, we prefer to develop the background on size verification, which is the most innovative result of this paper. Most of the formal details on type verification may be found in [1].

The type verification provides a bound on the length of the stacks during an execution and we note $h_{f,i}$ the size of the stack ℓ in a frame $(f, i, \ell)_\rho$ of a well-typed configuration. In the next section we show how to obtain a bound on the size of the computed values from our size analysis. Together, these two information can already be used to reject programs that compute arbitrarily large values but, to obtain a bound on the size needed for the execution of a program, we also need to bound the maximal number of frames, which usually necessitates a termination analysis.

3 Size Verification

We define a size verification based on the notion of quasi-interpretations [12]. This paper makes two additional contributions to our previous works on resource certification [1,2]. First, we are able to check programs whose CFG contains cycles, improving what was done in previous work. Second, we propose a direct algorithm that depends only on solving a set of arithmetical constraints, without resorting to an auxiliary *shape analysis*.

Quasi-interpretation. Quasi-interpretations have been defined by Marion et al. [12] to reason about the implicit complexity of term rewriting systems. The idea is close to polynomial interpretation for termination proofs: we assign to every function and constructor of a program a numerical function bounding the size of the computed values. More formally, a quasi-interpretation assigns to every identifier id in a program a function q_{id} (with arity $ar(id)$) over the non-negative rational numbers $\mathbb{Q}^+$ such that: (1) if c is a constant then $q_{\mathsf{c}}(\,) = 0$; (2) if c is a constructor with arity n then $q_{\mathsf{c}}(x_1, \ldots, x_n) = d + \Sigma_{i \in 1..n} x_i$, where $d \geqslant 1$; (3) if f is a function with arity n then $q_f : (\mathbb{Q}^+)^n \to \mathbb{Q}^+$ is monotonic and for all $i \in 1..n$ we have $q_f(x_1, \ldots, x_n) \geqslant x_i$.

An assignment can be easily extended to functional 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 $q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1}, \ldots, q_{e_n})$. Then an assignment is a valid quasi-interpretation for a system of recursive function definitions if for all declarations $f(p_1, \ldots, p_n) = e$ in the program, the inequality $q_{f(p_1,\ldots,p_n)} \geqslant q_e$ holds. For instance, if we choose $q_{\mathsf{s}} = 1 + x$ for the quasi-interpretation of the constructor in *nat* (by definition, $q_{\mathsf{z}} = 0$) then $q_{dble}(x) = 2x$ is a valid quasi-interpretation for the function *dble* defined in our examples: we have $q_{dble}(q_{\mathsf{z}}(\,)) \geqslant q_{\mathsf{z}}(\,)$ and $q_{dble}(q_{\mathsf{s}}(x)) \geqslant q_{\mathsf{s}}(q_{\mathsf{s}}(q_{dble}(x)))$. 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 $f(v_1, \ldots, v_n) \searrow v$ then $|v| \leqslant q_v \leqslant q_f(|v_1|, \ldots, |v_n|)$.

The problem of synthesizing quasi-interpretations (from a set of functional declarations) 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. For instance, Amadio [3,4] has considered the problem of automatically inferring quasi-interpretations in the space of multi-variate max-plus polynomials.

In this paper, we define a similar notion of quasi-interpretation for bytecode programs. Assume a function f of the bytecode program. An assignment associates to every checkpoint i of f a polynomial expression $q_{f,i}$ with $h_{f,i}$ variables. We also use the notation q_f to denote the function $q_{f,1}$ assigned to the entry point of f. Like in the functional case, we require that each polynomial $q_{f,i}$ satisfies the hypotheses for quasi-interpretations (properties (1)-(3) listed above). The machine-checkable certificates used in our size verification are quasi-interpretations, that is assignment of numerical functions, in our case polynomial expressions, to instructions in the program. An advantage of this approach is that quasi-interpretation can be synthesized at the source-code level and verified at the bytecode level. We will not address synthesis issues in this paper and we suppose that the bytecode comes with all the necessary types and size annotations. For example the function *tdble* has two checkpoints, the nodes 1 and 2, with respective types (nat) and (nat, nat), which means that $h_{tdble,1} = 1$ and $h_{tdble,2} = 2$. In the following we assume that the assignment is $q_{tdble,1}(x_1) = 2x_1$ and $q_{tdble,2}(y_1, y_2) = y_1 + y_2$.

Size Analysis. We show how to check the validity of an assignment and to obtain a size bound from a quasi-interpretation. Like the type verification, our *size verification* associates to every bytecode instruction an abstraction of the stack at the time it is executed. In this case, the abstraction is a combination of a sequence of *size variables*, which stands for the best size bounds we can obtain, together with arithmetic constraints between these variables. Contrary to the size verification defined in [1], we directly infer a size bound, without using an auxiliary "shape verification" (that is a static analysis which provides partial informations on the structure of the elements in the stack). The advantage of a direct approach is to get rid of the restrictions imposed by the shape analysis, especially: (1) that the CFG of functions must be a tree and (2) that along each execution path, we must not have a `branch` instruction after a `call` instruction.

We suppose that the bytecode is well-typed, which means that we know the number $h_{f,i}$ of elements on the stack before executing the instruction $f[i]$. We associate to each checkpoint i of the function f: (1) a sequence of fresh (size) variables $\vec{x}_{f,i} =_{\mathrm{def}} (x_1, \ldots, x_{h_{f,i}})$ and (2) a polynomial expression $q_{f,i}$ with variables $\vec{x}_{f,i}$ and coefficients in $\mathbb{Q}$.

The size analysis is formally defined by the predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ given in Table 2. The definition of $wsz_i(f, \vec{S}, \vec{\Phi})$ is by case analysis on the instruction $f[i]$ and expresses that (1) the size of every element on the stack at instruction i is bounded by the expression $q_{f,PC_i}(\vec{x}_{f,PC_i})$, and (2) the quasi-interpretations decrease every time we pass a new checkpoint. We say that the size analysis is successful if there are two sequences $\vec{S} = (S_1, \ldots, S_{|f|})$ and $\vec{\Phi} = (\Phi_1, \ldots, \Phi_{|f|})$ such that $wsz_i(f, \vec{S}, \vec{\Phi})$ for all $i \in 1..|f|$, and $S_i = \vec{x}_{f,i}$ and $\Phi_i = \emptyset$ if i is a checkpoint of f. We note this relation $wsz(f, \vec{S}, \vec{\Phi})$.

The size analysis is compositional (we only need to analyze each functions separately) and always terminates (since every instruction is visited at most once). The size analysis for a function f associates to every instruction i of f a

Table 2. Size Analysis ($wsz_i(f, \vec{S}, \vec{\Phi})$)

Let $j = PC_i$ be the checkpoint of i. Case $f[i]$ of:

(`load` k) let x_k be the k^{th} variable of S_i, and x a fresh size variable. If $Control(i+1)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \big(\Phi_i \wedge x = x_k\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S_i \cdot x)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_{i+1} = S_i \cdot x\big)$ and $\big(\Phi_{i+1} = \Phi_i \wedge x = x_k\big)$.

(`build c` n) Assume $n = ar(\mathtt{c})$ and $S_i = S' \cdot (x_1, \ldots, x_n)$, and let x_0 be a fresh size variable. First we check the validity of $\psi_{build} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{\mathtt{c}}(x_1, \ldots, x_n)\big)$. If $Control(i+1)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \big(\Phi_i \wedge x_0 = q_{\mathtt{c}}(x_1, \ldots, x_n)\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S' \cdot x_0)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_{i+1} = S' \cdot x_0\big)$ and $\big(\Phi_{i+1} = \Phi_i \wedge x_0 = q_{\mathtt{c}}(x_1, \ldots, x_n)\big)$.

(`branch c` k) Assume $n = ar(\mathtt{c})$ and $S_i = S' \cdot x_0$, and let $x_1, \ldots, x_n$ be fresh size variables. The predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ is true iff the following two conditions are true (one condition for each successor of i in f).

- **(C1)** if $Control(i+1)$ then $\psi_{then} =_{\text{def}} \big(\Phi_i \wedge x_0 = q_{\mathtt{c}}(x_1, \ldots, x_n)\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S' \cdot (x_1 \ldots x_n))\big)$ is a tautology otherwise $\big(S_{i+1} = S' \cdot (x_1 \ldots x_n)\big)$ and $\big(\Phi_{i+1} = \Phi_i \wedge x_0 = q_{\mathtt{c}}(x_1, \ldots, x_n)\big)$.
- **(C2)** if $Control(k)$ then $\psi_{else} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,k}(S_i)\big)$ is a tautology otherwise $\big(S_k = S_i\big)$ and $\big(\Phi_k = \Phi_i\big)$.

(`call` g n) Assume $n = ar(g)$ and $S_i = S' \cdot (x_1, \ldots, x_n)$ and let x_0 be a fresh size variable. First we check the validity of $\psi_{call} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{g,1}(x_1, \ldots, x_n)\big)$. If $Control(i+1)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \big(\Phi_i \wedge x_0 \leqslant q_{g,1}(x_1, \ldots, x_n)\big) \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,i+1}(S' \cdot x_0)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_{i+1} = S' \cdot x_0\big)$ and $\big(\Phi_{i+1} = \Phi_i \wedge x_0 \leqslant q_{g,1}(x_1, \ldots, x_n)\big)$.

(`tcall` g n) Assume $n = ar(g)$ and $S_i = S' \cdot (x_1, \ldots, x_n)$ and let x_0 be a fresh size variable. The predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ is true iff the formula ψ_{tcall} is valid, where $\psi_{tcall} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{g,1}(x_1, \ldots, x_n)\big)$.

(`jump` k n) Assume $S_i = S' \cdot (x_1, \ldots, x_n)$. If $Control(k)$ then $wsz_i(f, \vec{S}, \vec{\Phi})$ iff the formula $\psi_{succ} =_{\text{def}} \Phi_i \Rightarrow \big(q_{f,j}(\vec{x}_{f,j}) \geqslant q_{f,k}(x_1 \ldots x_n)\big)$ is a tautology. Otherwise $wsz_i(f, \vec{S}, \vec{\Phi})$ iff $\big(S_k = (x_1, \ldots, x_n)\big)$ and $\big(\Phi_k = \Phi_i\big)$.

(`stop` or `return`) Then the predicate $wsz_i(f, \vec{S}, \vec{\Phi})$ is true.

sequence of variables of size $h_{f,i}$, denoted S_i, and a set of constraints between linear combinations of these variables, denoted Φ_i. Intuitively, the k^{th} variable of S_i is a bound on the size of the k^{th} element of the execution stack when the instruction $f[i]$ is executed, while Φ_i contains valid constraints between the bounds. For example, if $f[i] = \texttt{load}\ k$ and $S_i = (x_1, \ldots, x_n)$ we impose that $S_{i+1} = S_i \cdot x$ and that Φ_{i+1} implies $x = x_k$, meaning that we add a value on top of the stack ℓ, whose size is bounded by x_k, our best known bound on the size of the k^{th} value in ℓ. The analysis generates also a set of *proof obligations*, $\psi_{succ}, \psi_{call}, \ldots$ which are arithmetic formulas that should be checked in order to prove the validity of the size certificates (i.e. the quasi-interpretation). We say that the formula $\Phi \Rightarrow p(\vec{x}) \geqslant q(\vec{y})$ with free size variables $\vec{y}$ is a tautology if for all valuation σ from $\vec{y}$ to positive natural numbers such that $\sigma(\Phi)$ is true then the inequality $\sigma(p(\vec{x}) \geqslant \sigma(q(\vec{y}))$ is true.

We show the result of the size analysis for our running example, *tdble*. We give in front of each instruction the size stack S_i and the constraint Φ_i such that $wsz(f, \vec{S}, \vec{\Phi})$. Then we check the validity of the various auxiliary conditions: there is one condition to prove each time the successor of an instruction is a checkpoint and one condition to prove for each `build`, `call` and `tcall` instruction.

$\underline{1}$: `load` 1	x_1	$\emptyset$
$\underline{2}$: `load` 1	$y_1\ y_2$	$\emptyset$
3 : `branch s` 7	$y_1\ y_2\ z_1$	$(z_1 = y_1)$
4 : `load` 2	$y_1\ y_2\ z_2$	$(z_1 = y_1) \wedge (z_1 = z_2 + 1)$
5 : `build s` 1	$y_1\ y_2\ z_2\ z_3$	$(z_1 = y_1) \wedge (z_1 = z_2 + 1) \wedge (z_3 = y_2)$
6 : `jump` 2 2	$y_1\ y_2\ z_2\ z_4$	$(z_1 = y_1) \wedge (z_1 = z_2 + 1) \wedge (z_3 = y_2) \wedge (z_4 = z_3 + 1)$
7 : `load` 2	$y_1\ y_2\ z_1$	$(z_1 = y_1)$
8 : `return`	$y_1\ y_2\ z_1\ z_5$	$(z_1 = y_1) \wedge (z_5 = y_2)$

We need to check three proof obligations in the size verification of *tdble*. The first formula corresponds to the `build` instruction $\psi_5 = \Phi_5 \Rightarrow \big(q_{tdble,2}(y_1, y_2) \geqslant q_{\mathsf{s}}(z_3)\big)$. The two others formulas correspond to the possible transitions to checkpoint 2 (from instructions 1 and 6) which gives $\psi_1 =_{\text{def}} \Phi_1 \Rightarrow \big(q_{tdble,1}(x_1) \geqslant q_{tdble,2}(x_1, x_1)\big)$ and $\psi_6 =_{\text{def}} \Phi_6 \Rightarrow \big(q_{tdble,2}(y_1, y_2) \geqslant q_{tdble,2}(z_2, z_4)\big)$. Once simplified, we can easily show that these constraints are valid: ψ_5 is equivalent to $(z_1 = y_1) \wedge (z_1 = z_2 + 1) \wedge (z_3 = y_2) \Rightarrow y_1 + y_2 \geqslant z_3 + 1$, while $\psi_1 \equiv 2x_1 \geqslant x_1 + x_1$ and $\psi_6 \equiv (y_1 = z_2 + 1) \wedge (z_4 = y_2 + 1) \Rightarrow (y_1 + y_2 \geqslant z_2 + z_4)$.

Next, we show the result of the size analysis for the function *sum*. We assume that the quasi-interpretations of *sum* and *add* are the functions $q_{sum}(x) = \frac{1}{2}x(x+1)$ and $q_{add}(x, y) = x + y$.Instruction 5 of *sum* is a checkpoint and we assume that $q_{sum,5}(x) = x$.

$\underline{1}$: `load` 1	x_1	$\emptyset$
2 : `branch s` 5	$x_1\ x_2$	$(x_2 = x_1)$
3 : `call` *sum* 1	$x_1\ y_1$	$(x_2 = x_1) \wedge (x_2 = y_1 + 1)$
4 : `call` *add* 2	$x_1\ y_2$	$(x_2 = x_1) \wedge (x_2 = y_1 + 1) \wedge (y_2 \leqslant q_{sum}(y_1))$
$\underline{5}$: `return`	z_1	$\emptyset$

The analysis of *sum* (note that the functions *add* and *sum* may be analysed separately) gives only two non-trivial proof obligations that are related to the two `call` instructions in the code. These formulas are $\psi_3 = \Phi_3 \Rightarrow \big(q_{sum}(x_1) \geqslant q_{sum}(y_1)\big)$ and $\psi_4 = \Phi_4 \Rightarrow \big(q_{sum}(x_1) \geqslant q_{add}(x_1, y_2)\big)$. Once simplified, we can easily show that these constraints are valid: ψ_3 is equivalent to $q_{sum}(y_1 + 1) \geqslant q_{sum}(y_1)$, while ψ_4 is a consequence of $q_{sum}(y_1 + 1) \geqslant q_{sum}(y_1) + y_1 + 1$.

Finally, we can check that the function *xdble*, our example of malicious code, does not succeed the size analysis. Let us consider the proof obligations generated in the analysis of the function *xdble*:

$\underline{1}$: `load` 1	x_1	$\emptyset$
2 : `build s` 1	$x_1\ x_2$	$x_2 = x_1$
3 : `build s` 1	$x_1\ x_3$	$x_2 = x_1 \wedge x_3 = x_2 + 1$
4 : `call` *xdble* 1	$x_1\ x_4$	$x_2 = x_1 \wedge \cdots \wedge x_4 = x_3 + 1$
5 : `return`	$x_1\ x_5$	$x_2 = x_1 \wedge \cdots \wedge x_5 \leqslant q_{xdble,1}(x_4)$

The condition corresponding to instruction 4, the only `call` instruction, is $\Phi[4] \Rightarrow q_{xdble,1}(x_1) \geqslant q_{xdble,1}(x_4)$, that is equivalent to $x_4 = x_1 + 2 \Rightarrow q_{xdble,1}(x_1) \geqslant q_{xdble,1}(x_4)$, which is obviously not satisfiable since $q_{xdble,1}(x)$ is monotone.

Deriving Size Bounds from the Size Analysis. We prove that if the size analysis returns a solution for all the functions of a program, then we can extract a bound on the size of the values computed during the execution. In order to prove this property, we extend the predicate *wsz* to frames and then configurations of the virtual machine.

Assume $wsz(f, \vec{S}, \vec{\Phi})$ and let ρ be the annotation $(g(\ell_o), k, (v_1 \dots v_n))$. We say that the frame $(f, i, \ell)_\rho$ is *well-sized* if $q_{f,k}(q_{v_1}, \dots, q_{v_n})$ bounds the size of all the values in ℓ and if the constraint Φ_i is verified when we replace the variables of S_i by the quasi interpretation of the corresponding values in ℓ and the variables of $\vec{x}_{f,k}$ by the values $q_{v_1}, \dots, q_{v_n}$. We denote this last property $(f, i, \ell)_\rho \models (\vec{S}, \vec{\Phi})$. Then we say that the configuration M is well-sized if all the frames in M are well-sized and if the quasi-interpretations of the checkpoints decrease, see the table below which defines the relation $wsz(M)$. We introduce some auxiliary notations to help us define the relation *wsz* formally. Assume $wsz(f, \vec{S}, \vec{\Phi})$ and let $(f, i, \ell)_\rho$ be a frame such that ρ is the annotation $(g(\ell_o), k, \ell_c)$. Assume $\ell_o = (u_1, \dots, u_n)$ and $\ell_c = (u'_1, \dots, u'_m)$. We define the two expressions $\hat{q}(\rho)$ and $q(f, \rho)$ as follows: $\hat{q}(\rho) =_{\mathrm{def}} q_{g,1}(q_{u_1}, \dots, q_{u_n})$ and $q(f, \rho) =_{\mathrm{def}} q_{f,k}(q_{u'_1}, \dots, q_{u'_m})$. The value of $\hat{q}(\rho)$ denotes the best size bound known when the frame is initialized, while $q(f, \rho)$ denotes the best size bound known when we reached the last checkpoint. Let $\ell = (v_1, \dots, v_n)$ be a sequence of values and $\vec{x} = (x_1, \dots, x_n)$ a sequence of variables of the same length. We write $\left[{}^{\ell}/_{\vec{x}}\right]^{||}$ the substitution $\left[{}^{q_{v_1}}/_{x_1}\right] \dots \left[{}^{q_{v_n}}/_{x_n}\right]$. The constraint $\vec{\Phi}$ is true for the frame $(f, i, \ell)_\rho$, denoted $(f, i, \ell)_\rho \models (\vec{S}, \vec{\Phi})$ if and only if the constraint $\Phi_i\left[{}^{\ell}/_{S_i}\right]^{||}\left[{}^{\ell_c}/_{\vec{x}_{f,k}}\right]^{||}$ is valid.

Well-Sized Configurations: $wsz(M)$

$$\frac{wsz(f,\vec{S},\vec{\Phi}) \quad (f,pc,\ell)_\rho \models (\vec{S},\vec{\Phi}) \quad \ell=(v_1,\dots,v_n) \quad \hat{q}(\rho) \geqslant q(f,\rho) \geqslant |v_i| \quad i \in 1..n}{wsz(f,pc,\ell)_\rho}$$

$$\frac{\begin{array}{c} M \equiv (f_1,i_1,\ell_1)_{\rho_1}\dots(f_m,i_m,\ell_m)_{\rho_m} \\ \ell_k^o = arg(M,k+1) \quad wsz(f_k,i_k,\ell_k\cdot\ell_k^o)_{\rho_k} \\ wsz(f_m,i_m,\ell_m)_{\rho_m} \quad q(f_j,\rho_j) \geqslant q(f_{j+1},\rho_{j+1}) \\ k \in 1..m-1 \quad j \in 1..m-1 \end{array}}{wsz(M)}$$

We can show that the predicate *wsz* is preserved by reduction.

Theorem 1 (Preservation). *If $wsz(M)$ and $M \to M'$ then $wsz(M')$.*

Proof. By induction on the derivation of $M \to M'$, see [8] for a detailed proof.

A corollary of this result is that for every program succeeding the size analysis, if the initial configuration $(f,1,(v_1\dots v_n))$ is well-sized then the values computed during the execution are bounded by $q_f(q_{v_1},\dots,q_{v_n})$.

Theorem 2 (Size Bound). *Assume f is a function in a program that succeeds the size analysis. If the initial configuration $(f,1,(v_1\dots v_n))$ reduces to M then for all value v occurring in a frame of M we have $|v| \leqslant q_f(q_{v_1},\dots,q_{v_n})$.*

Proof. Let ℓ be a stack of the form $(v_1,\dots,v_n)$. By hypothesis we have $(f,1,\ell)_\rho \to^* M$ with $\rho = (f(\ell),1,\ell)$ and $M \equiv (f_1,i_1,\ell_1)_{\rho_1} \ \dots \ (f_m,i_m,\ell_m)_{\rho_m}$, where ρ_1 is of the form $(f(\ell),PC_{i_1},\ell_c^1)$. By Theorem 1 we have that M is well-sized, that is $wsz(M)$. Hence (1) $q(f_j,\rho_j) \geqslant q(f_{j+1},\rho_{j+1})$ for all $j \in 1..m-1$ and (2) $wsz(f_k,i_k,\ell_k\cdot\ell_k^o)_{\rho_k}$ for all $k \in 1..m-1$ and $wsz(f_m,i_m,\ell_m)_{\rho_m}$. By property (2) and definition of the predicate *wsz* on frames, $|v| \leqslant q(f_k,\rho_k) \leqslant \hat{q}(\rho_k)$ for all value v occurring in the k^{th} frame of M and by property (1) we obtain that $|v| \leqslant q(f_1,\rho_1) \leqslant \hat{q}(\rho_1) = q_f(q_{v_1},\dots,q_{v_n})$, as needed.

4 Solving Size Constraints

Size verification generates a system of auxiliary arithmetical constraints that we need to solve. On the whole, each constraint is of the form $\Phi \Rightarrow p(\vec{x}) \leqslant q(\vec{y})$, where Φ is a conjunction of equality and inequality constraints (see the discussion below) and p,q are polynomial expressions with coefficients in $\mathbb{Q}$. A constraint is generated for each `build`, `call` and `tcall` instruction and for each transition from an instruction to a checkpoint. In this section we study the problem of checking the validity of these constraints and show that we can always reduce to the problem of checking the sign of a polynomial expression.

We start by partitioning the set V of all size variables used in the size verification. We define the sets $V_{\texttt{load}}$, $V_{\texttt{build}}$, $V_{\texttt{branch}}$ and $V_{\texttt{call}}$ of variables that were introduced respectively when checking a `load`, `build`, `branch` and a `call` or a `tcall` instruction. We also define the set V_o of all variables associated to checkpoints. To simplify our result, we assume that `branch` instructions never act on variables in $V_{\texttt{build}}$ (and transitively on variables introduced by a `load` instruction that corresponds to a variable of $V_{\texttt{build}}$). Intuitively, this corresponds to forbid cases where a `branch` instruction is applied to a value whose head constructor is known at compile time (indeed it is possible to trace back the `build` instruction that created it). A consequence of this assumption is to avoid "dead-code", i.e. a part of the code that cannot be reached during an execution.

A brief inspection of the definition of *wsz* shows that the proof obligations generated during the size analysis are all of the form $\Phi \Rightarrow q_f(\vec{x}_f) \geqslant q_g(y_1, \ldots, y_n)$, where $\vec{x}_f$ is a vector of fresh variables and Φ is a conjunction of atoms of the form:

(Load)	$y = x$	where $y \in V_{\texttt{load}}$
(Build)	$y = \sum_{i \in I} x_i + d$	where $y \in V_{\texttt{build}}$, and $d \geqslant 1$
(Branch)	$\sum_{i \in I} y_i + d = x$	where $y_i \in V_{\texttt{branch}}$ for all $i \in I$ and $d \geqslant 1$
(Call)	$y \leqslant q(x_1, \ldots, x_n)$	where $y \in V_{\texttt{call}}$ and q is a polynomial expression with the properties of quasi-interpretations.

We can solve this kind of constraints using the following simple steps:

- first, we eliminate the variables of $V_{\texttt{load}}$ and $V_{\texttt{build}}$ by substitution. This step eliminates the constraints of type (Load) and (Build). The system resulting after this step is made up of (Branch) and (Call) constraints and all the remaining variables are in $V \setminus (V_{\texttt{load}} \cup V_{\texttt{build}})$;
- then we use the hypothesis that we never apply a `branch` instruction on a value introduced by a `build`. So we can replace every (Branch) constraint by a simple substitution. Hence all the constraints of the resulting system are of type (Call) with variables in $V_o \cup V_{\texttt{call}}$;
- finally we are left to check an inequality of the form $\sigma(g(y_1, \ldots, y_n)) \leqslant \sigma(f(\vec{x}_f))$ where σ is the substitution obtained after the first two steps. By construction there are no variables of $V_{\texttt{call}}$ in $\sigma(f(\vec{x}_f))$. Let $C_1, \ldots, C_m$ be the remaining (Call) constraints. For every $i \in 1..m$ the constraint C_i is of the kind $z_i \leqslant p(\vec{a_i})$. Since there are no variables of $V_{\texttt{call}}$ in $\sigma(f(\vec{x}_f))$ we can simply check the inequality after replacing the occurrences of z_i by the expression $p(\vec{a_i})$ (since we work with quasi-interpretation the function p is monotone). Hence it is equivalent to check the sign of the (polynomial) expression: $\big(f(\vec{x}_f) - g(y_1, \ldots, y_n)\big)(\sigma \circ \left[{}^{p(\vec{a_i})}/_{z_i}\right]_{i \in 1..m})$.

5 Conclusion and Related work

Ensuring bounds on the resources needed for executing a program is a critical safety property. In this paper, we define a new "size analysis" and show how to derive a bound on the size of the values computed by a program. This method has several advantages. The size-bound obtained with our approach is a polynomial expression on the size of the input parameters of the program. Also, programs can be analyzed incrementally (each function is analyzed separately), at the level of the bytecode. These features are particularly interesting in the context of mobile code applications, in which programs can be dynamically loaded from untrusted, possibly malicious sites.

The problem of bounding computational resources has already attracted considerable attention. Many works have focused on (first-order) functional languages starting from Cobham's characterization of polynomial time functions by bounded recursion on notation [6]. Following works, see e.g. [5,9,10], have developed various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms. None of these works have been applied to bytecode languages. Actually, most of the researches on bytecode verification tends to concentrate on the integrity of the execution environment. We have presented in [1] a virtual machine and a corresponding bytecode for a first-order functional language and shown how size and termination annotations can be formulated and verified at the level of the bytecode. In this paper, we extend this language with instructions for "tail recursive" calls and unconditional jumps, which are vital to implement common program optimizations. In particular, we can analyze bytecode sequences whose control flow graph includes cycles, whereas the size analysis defined in [1] can only handle tree shaped control flow graphs. Work on resource bounds for "Java-like" bytecode languages is carried out in the MRG project [15]. One main technical difference is that they rely on a general proof carrying code approach while we follow a Typed Assembly Language (TAL) approach. Also, their analysis focuses on the size of the heap while we only consider stack allocated values. Crary and Weirich [7] define a TAL for resource bound certification. Their approach is based on a dependent type-system where types include a "resource skeleton", that is a set of functions (expressed in a ML-like language) computing the resource behavior of the program. Resource skeleton cannot be inferred and should be written by the programmer. Another related work is due to Marion and Moyen [13] who define a resource analysis for counter machines by reduction to a certain type of termination in Petri Nets. Their virtual machine is much more restricted than the one studied here: natural numbers is the only data type and the stack can only contain return addresses.

We are currently experimenting with the automatic derivation of quasi-interpretation at the bytecode level. At the moment, we only have methods to infer quasi-interpretations (with max-plus polynomials) from functional code [4]. Plans for future works also include extending our approach to a more complicated virtual machine, e.g. with support for objects (as in the Java Virtual Machine), heap references or subroutines.

References

1. R. Amadio, S. Coupet-Grimal, S. Dal Zilio and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *International Conference on Computer Science Logic (CSL)*, LNCS, vol. 3210, Springer, 2004.
2. R. Amadio and S. Dal Zilio. Resource control for synchronous cooperative threads. In *15th International Conference on Concurrency Theory (CONCUR)*, LNCS vol. 3170, Springer, 2004.

3. R. Amadio. Max-plus quasi-interpretations. In *6th International Conference on Typed Lambda Calculi and Applications (TLCA)*, LNCS vol. 2701, Springer, 2003.
4. R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2005.
5. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
6. A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings Logic, Methodology, and Philosophy of Science II*, North Holland, 1965.
7. K. Crary and S. Weirich. Resource bound certification. In *Principles of Programming Languages (POPL)*. ACM, 2000.
8. S. Dal Zilio and R. Gascon. Resource Bound Certification for a Tail-Recusive Virtual Machine. LIF Research Report 26, 2005.
9. M. Hofmann. The strength of non size-increasing computation. In *Principles of Programming Languages (POPL)*, ACM, 2002.
10. N. Jones. *Computability and complexity, from a programming perspective.* MIT-Press, 1997.
11. T. Lindholm and F. Yellin. *The Java virtual machine specification.* Addison-Wesley, 1999.
12. J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique.* Habilitation à diriger des recherches, Université de Nancy, 2000.
13. J.-Y. Marion and J.-Y. Moyen. *Termination and resource analysis of assembly programs by Petri Nets.* Technical Report, Université de Nancy, 2003.
14. 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.
15. D. Sannella. Mobile Resource Guarantee. IST-Global Computing research project, 2001. `http://www.dcs.ed.ac.uk/home/mrg/`.
16. G. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages (POPL)*. ACM, 1973.
17. G Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*. ACM, 1997.
18. M. Tofte and J.-P. Talpin. Region-Based Memory Management. In *Information and Computation*, 132(2):109–176, 1997.