LIF

Laboratoire d'Informatique Fondamentale
de Marseille

Unité Mixte de Recherche 6166
CNRS - Université de Provence - Université de la Méditerranée

A Functional Scenario for Bytecode Verification of Resource Bounds

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

Rapport/Report 17-2004

4 January, 2004

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

UMR 6166
CNRS - Université de Provence - Université de la Méditerranée

{amadio,solange,dalzilio,jakubiec}@lif.univ-mrs.fr

Abstract/Résumé

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.

On définit une simple machine à pile pour un langage fonctionnel du premier ordre et on montre comment mener des vérifications de type, de taille et de terminaison au niveau du code octet de la machine. En particulier, on montre qu'une combinaison de la vérification de taille basée sur les quasi-interprétations et de la vérification de terminaison basée sur un ordre récursif sur les chemins lexicographique mène à une borne explicite sur l'espace nécessaire à l'exécution.

Relecteurs/Reviewers: Fréderic Dabrowski, Denis Lugiez.

Notes: The authors are partly supported by ACI CRISS.

1 Introduction

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

Initial proposals have focused on the integrity 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 [MWCG99, Nec97]. First, mobile code is shipped around in pre-compiled form (*i.e.*, bytecode) 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: we only have to trust the analyser, not the whole compilation chain.

Approach. The problem of bounding the usage made by programs of their resources has already attracted considerable attention. Automatic extraction of resource bounds has mainly focused on (first-order) functional languages starting from Cobham's characterization [Cob65] of polynomial time functions by bounded recursion on notation. Following work, see, *e.g.*, [BC92, Lei94, Jon97, Hof02], 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 [BMM01] 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 generalizations and variations over Cobham's characterization.

Contribution. The synthesis of termination orderings is a classical topic in term rewriting (see for instance [BN98]). The synthesis of quasi-interpretations—a concept introduced by Marion *et al.* [Mar00]—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 [Ama03]. 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 the program 'implements a PSPACE function', 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 $|v|$ stands for the size of the value v.

Our secondary goal is of a pedagogical nature: present a minimal but still relevant scenario in which problems connected to bytecode verification can be effectively discussed. Our functional virtual machine is based on a set of 6 instructions, a number that has to be compared with the almost 200 opcodes used in the JAVA virtual machine [LY99].

Paper organisation. The paper is organized as follows. Section 2 sketches a first-order functional language with simple types and call-by-value evaluation and recalls some basic facts about quasi-interpretations and termination. Section 3 describes a simple virtual machine comprising a minimal set of 6 instructions that suffice to compile the language described in the previous section. 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 path verification which is described in Section 5. We provide an example of type, size,and termination verifications in Section 8. The presentation of each verification follows a common pattern: (i) definition of constraints on the bytecode and (ii) definition of a predicate which is invariant under machine reduction. The essential technical difficulty is in the structuring of the constraints and the invariants, the proofs are then routine inductive arguments which we delay to the appendix.

Related work. 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 [San01]. 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 [MM03] 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.

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 (first-order) variables, and distinguish the following three syntactic categories:

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

If e is an expression then $\mathit{Var}(e)$ 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 $f(e_1,\ldots,e_n)$.

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 (a variable appears at most once) and do not superpose.

2.1 Types

We use $t, t_1, \ldots$ to range over type identifiers. A type definition associates to each identifier the sequence of the types of its constructors, of the form $\mathsf{c}\ \mathit{of}\ t_1 * \cdots * t_n$. Hence, a type definition has the shape:

$$t \;=\; \mathsf{c_1}\ \mathit{of}\ t_1^1 * \cdots * t_{n_1}^1 \;\mid\; \cdots \;\mid\; \mathsf{c_k}\ \mathit{of}\ t_1^k * \cdots * t_{n_k}^k$$

For instance, we can define the type *bword* of binary words and the type *nat* of natural numbers in unary format:

$$\begin{array}{lcll} \mathit{bword} & = & \mathsf{Nil} & \mid \mathsf{0}\ \mathit{of}\ \mathit{bword} \mid \mathsf{1}\ \mathit{of}\ \mathit{bword} \\ \mathit{nat} & = & \mathsf{z} & \mid \mathsf{s}\ \mathit{of}\ \mathit{nat} \end{array}$$

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.

2.2 Evaluation

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

EVALUATION: $e \Downarrow v$

(Eval Cnst)

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

(Eval Fun)

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

Example 1 *The function add : (nat, nat) → nat, defined by the following two rules, computes the sum of two values of type nat.*

$$\begin{array}{lccl} \mathit{add}(& \mathsf{z}, & y) & \Rightarrow & y \\ \mathit{add}(& \mathsf{s}(x), & y) & \Rightarrow & \mathit{add}(x, \mathsf{s}(y)) \end{array}$$

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

2.3 Quasi-interpretations

Given a program, an *assignment* q associates to constructors and function symbols functions over the non-negative reals $\mathbb{R}^+$ such that:

- if c is a constant then q_c is the constant 0,
- 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 $d \geqslant 1$,
- 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:

$$\begin{array}{lcl} q_x & = & x \ , \\ q_{\mathsf{c}(e_1,\ldots,e_n)} & = & q_{\mathsf{c}}(q_{e_1},\ldots,q_{e_n}) \ , \\ q_{f(e_1,\ldots,e_n)} & = & q_f(q_{e_1},\ldots,q_{e_n}) \ . \end{array}$$

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 all the rules $f(p_1,\ldots,p_n) \Rightarrow e$ in the program, the following inequality holds over $\mathbb{R}^+$:

$$q_{f(p_1,\ldots,p_n)} \ \geqslant \ q_e \ . \tag{1}$$

Example 2 *With reference to the Example 1, consider the assignment $q_{\mathsf{s}} = 1 + x$ 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.*

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. One can show that every value v computed during the evaluation of $f(v_1,\ldots,v_n)$ satisfies the following condition:

$$|v| \ \leqslant \ q_v \ \leqslant \ q_{f(v_1,\ldots,v_n)} \ = \ q_f(q_{v_1},\ldots,q_{v_n}) \ \leqslant \ q_f(d|v_1|,\ldots,d|v_n|) \ .$$

Here d is a constant that depends only on the program and that can be chosen as the largest additive constant in the interpretation of the constructors.

An interesting space for the synthesis of quasi-interpretations is the collection of max-plus polynomials [Ama03], 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 Pressburger formula, and is therefore a decidable problem.

2.4 Termination

Programs can be regarded as a set of term rewriting rules, just associate to every rule $f(p_1,\ldots,p_n) \Rightarrow e$ the *term rewriting rule* $f(p_1,\ldots,p_n) \to e$. Hence termination methods developed for term rewriting systems apply. In particular, under the hypothesis that the rules are orthogonal, termination of the TRS is equivalent to the termination of the call-by-value evaluation strategy [Gra96].

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.

Notation. 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 $\vec{y}$ a vector $(y_1,\ldots,y_n)$ of elements. Then, $\vec{y}_i$ stands for the element y_i and $|\vec{y}|$ is the number n of elements in the vector. In the following, we will often manipulate *vectors of sequences* and use the notation $\vec{y}_i[k]$ to denote the k^{th} element in the i^{th} sequence of vector $\vec{y}$.

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* modeled 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 the table below and write $M \to M'$ if the configuration M reduces to M' by applying exactly one of the transformations.

BYTECODE INTERPRETER: $M \to M'$

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

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

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

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

(Stop)
$$\frac{f[pc] = \texttt{stop}}{M \cdot (f, pc, \ell) \to \epsilon}$$

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

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

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 $f[pc] = \texttt{return}$. In this case the result of the evaluation is v_0. All the other cases of blocked configuration, such that $M \not\to$, are considered as runtime errors.

Definition 1 *We say that the configuration M is a result v_0, denoted $M \downarrow v_0$, if $M \equiv (f, pc, \ell \cdot v_0)$ with $1 \leqslant pc \leqslant |f|$ and $f[pc] = \texttt{return}$.*

Informal semantics. We provide an informal explanation of the semantics of the machine instructions. The instruction `load` $(i : nat)$ takes as parameter a valid index in the current stack ℓ. Upon execution, the stack is updated by pushing a copy of the i^{th} value in ℓ to the top. New values may also be created in the stack using the instruction `build` $(\mathsf{c} : ident)$ $(n : nat)$, which takes an identifier that corresponds to a constructor with arity n. When executed, the current stack, ℓ, is updated as follows: the n values, $v_1, \dots, v_n$, on top of ℓ are discarded and replaced by the single value $\mathsf{c}(v_1, \dots, v_n)$.

The following instructions `call`, `return`, and `stop` are the only ones with an effect on the number of frames contained in the configuration. A function call is implemented by the instruction `call` $(g : ident)$ $(n : nat)$, with first parameter an identifier corresponding to a function with arity n. Upon execution, a new frame is created, which is initialized with a copy of the n values on top of the caller's stack. The lifetime of the current frame is controlled by two instructions, `return` and `stop`. The former discards the current frame and returns the value at the head of the stack to the caller (i.e., the previous frame in the configuration) while the latter stops the virtual machine in an erroneous state.

Finally, the instruction `branch` $(\mathsf{c} : ident)$ $(j : nat)$ implements a conditional jump on the shape of the value, v, found on top of the current stack. If v is of the form $\mathsf{c}(v_1, \dots, v_n)$, then the top of the current execution stack is discarded and replaced by the n sub-values $v_1, \dots, v_n$. Otherwise, the stack is left unchanged and the execution jumps to position j in the code (where $1 \leqslant j \leqslant |f|$).

3.1 Compilation

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.

Example 3 *We give the result of the compilation of the function add in example 1, and show in parallel the computation of the expression* $add(\mathsf{s}(\mathsf{z}), \mathsf{z})$.

$add : (nat, nat) \to nat$	$(add,\ 1,\ \mathsf{s(z)} \cdot \mathsf{z})$
1 `load` 1	$\to (add,\ 2,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{s(z)})$
2 `branch` s 7	$\to (add,\ 3,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z})$
3 `load` 2	$\to (add,\ 4,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z} \cdot \mathsf{z})$
4 `build` s 1	$\to (add,\ 5,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z} \cdot \mathsf{s(z)})$
5 `call` *add* 2	$\to (add,\ 5,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z} \cdot \mathsf{s(z)})\ (add,\ 1,\ \mathsf{z} \cdot \mathsf{s(z)})$
6 `return`	$\to (add,\ 5,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z} \cdot \mathsf{s(z)})\ (add,\ 2,\ \mathsf{z} \cdot \mathsf{s(z)} \cdot \mathsf{z})$
7 `load` 2	$\to (add,\ 5,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z} \cdot \mathsf{s(z)})\ (add,\ 7,\ \mathsf{z} \cdot \mathsf{s(z)} \cdot \mathsf{z})$
8 `return`	$\to (add,\ 5,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \mathsf{z} \cdot \mathsf{s(z)})\ (add,\ 8,\ \mathsf{z} \cdot \mathsf{s(z)} \cdot \mathsf{s(z)})$
	$\to (add,\ 6,\ \mathsf{s(z)} \cdot \mathsf{z} \cdot \underline{\mathsf{s(z)}})$
	returned result is $\mathsf{s(z)}$

Clearly, a realistic implementation should at least include:

1. a mechanism to execute efficiently tail recursive calls (when a `call` instruction is immediately followed by `return`): in this case, the new frame can just replace the calling frame.
2. a mechanism to share common sub-values in a configuration. For instance, one could keep a stack of pointers to values which are allocated on a heap. Various policies could then be considered to garbage collect the heap.

We leave the considerations on the possible enhancements of the virtual machine for future work.

3.2 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. For instance, to guarantee that the program counter stays within the intended bounds.

Since a new frame may only originate from a `call` instruction, we can easily define a well-formedness condition on the configurations reachable during an execution. Indeed, for any pair of contiguous frames, $\cdots(f, pc, \ell)\cdot(g, pc', \ell')\cdots$, occurring in a "reachable configuration", 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.

Definition 2 (Frame Parameters) *The expression $arg(M, j)$ stands for the vector of arguments with which, under suitable hypothesis, the j^{th} frame in M has been called: if $M \equiv (f_1, i_1, \ell_1)\cdots(f_m, i_m, \ell_m)$ and $1 < j \leqslant m$, we have:*

$$arg(M, j) = (v_1, \ldots, v_k) \qquad \text{where } ar(f_j) = k \text{ and } \ell_{j-1} = \ell \cdot v_1 \cdots v_k\ .$$

By convention, we use $arg(M, 1)$ for the sequence of values used to initialize the execution of the machine, that is, the values occurring in the initial stack of the initial frame.

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 $f[i] =$ `load` k then $k \geqslant 1$ and (2) if $f[i] =$ `branch 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 $f_j[i_j] =$ `call` f_{j+1}. Well-formedness is preserved during execution, in particular, the configuration ϵ is well-formed.

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

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. It can be directly used as the basis of an algorithm for validating the bytecode before its execution by the interpreter; the only difference is that we do not have to consider access modifiers and object initialization 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 $\vec{T}$ of type stacks such that $|\vec{T}| = |f|$.

To express that an abstract execution $\vec{T}$ is coherent with the instructions in f, we define the notion of *well-typed instruction* based on the auxiliary relation $wt_i(f, \vec{T})$, given below. Informally, we have $wt_i(f, \vec{T})$ if $\vec{T}_i = t_1 \cdots t_k$ and 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_i is a value of type t_i for every $i \in 1..k$.

Well-Typed Instructions: $wt_i(f, \vec{T})$

The definition of the relation $wt_i(f, \vec{T})$, where $|f| = |\vec{T}|$, is by case analysis on the instruction $f[i]$.

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

It is now possible to 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 3 (Well-Typed Function) *A sequence $\vec{T}$ is a* valid abstract execution *for the function f with signature $(t_1, \ldots, t_n) \to t_0$, denoted $wt(f, \vec{T})$, if and only if $\vec{T}_1 = t_1 \cdots t_n$ and $wt_i(f, \vec{T})$ 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.

Proposition 2 *If every node in the flow graph $(\{1, \ldots, |f|\}, E_f)$ is reachable form the node 1 then there is at most one abstract execution, $\vec{T}$, such that $wt(f, \vec{T})$. Moreover, $\vec{T}$ can be effectively computed.*

Proof. Assume there is a solution $\vec{T}$. For every $i \in 1..|f|$, show by induction on the length of the shortest path from 1 to i that $\vec{T}_i$ is uniquely determined. □

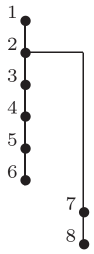
In the following, we assume that every node in the flow graph is accessible. If $\vec{T}$ is "the valid abstract execution" of f, we say that f is a function (code) of type $\vec{T}$.

Example 4 *We continue with our running example and display the type of each instructions in the code of add. We also show the flow graph associated to the function, which is a tree corresponding to the two possible "execution paths" in the code of add.*

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

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

E_{add}



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 a value of type t_i for all $i \in 1..n$.

WELL-TYPED CONFIGURATIONS: $wt(M)$

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

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

Assume the bytecode of the function f has passed the type verification. If $f(v_1, \ldots, v_n)$ is a well-typed expression of the functional languge then it is easy to check that the initial configuration $(f, 1, v_1 \cdots v_n)$ is also well-typed.

Proposition 3 (Type Invariant) *Let M be a configuration. If $wt(M)$ and $M \rightarrow M'$ then $wt(M')$.*

The soundness of the type verification follows from a progress property.

Proposition 4 (Progress) *Assume M is a well-typed configuration. Then either (1) $M \equiv \epsilon$, or (2) M is a result (cf. definition 1), or (3) M reduces, $\exists M' \ (M \rightarrow M')$.*

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

5 Shape Verification

We define a shape verification on the bytecode which appears to be original. Instead of simply 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).

Notation. We suppose that the code of every function f in the program passes the type verification of Section 4 and that $wt(f, \vec{T})$. We denote with $\vec{h}$ a vector of numbers such that $\vec{h}_i$ is the height of the stack for instruction i, that is $\vec{h}_i = |\vec{T}_i|$ for all $i \in 1..|f|$. Furthermore, for every instruction index i and position $k \in 1..\vec{h}_i$ 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.

A first attempt. We start by giving some intuitions on the approach used for the shape verification. The idea is to use the variable $x_{i,k}$ to express constraints on the value occurring in k^{th} position in the stack during the evaluation of the instruction $f[i]$. For example, these constraints could be expressed in the first-order theory of equality between expressions. The following table defines for every instruction index $i \in 1..|f|$ a formula ϕ_i constraining the variables $x_{j,l}$, with $j \in 1..|f|$ and $l \in 1..\vec{h}_j$. There are no constraints for `return` or `stop` instructions and we assume that ϕ_1 is the empty constraint. For instance, if $f[i] =$ `build c` n, the constraints in ϕ_{i+1} entails that the value represented by $x_{i+1,\vec{h}_{i+1}}$ (the value on top of the stack at the time of the evaluation of $f[i+1]$) is of the form $\mathsf{c}(\ldots)$.

$f[i]$	Shape constraints ϕ_i
`load` k	$\phi_{i+1} = \bigwedge_{1 \leqslant l < \vec{h}_{i+1}} (x_{i+1,l} = x_{i,l}) \wedge (x_{i+1,\vec{h}_{i+1}} = x_{i,k})$ where $\vec{h}_{i+1} = \vec{h}_i + 1$
`build c` n	$\phi_{i+1} = \bigwedge_{1 \leqslant l < \vec{h}_{i+1}} (x_{i+1,l} = x_{i,l})$ $\wedge \left(x_{i+1,\vec{h}_{i+1}} = \mathsf{c}(x_{i,\vec{h}_{i+1}}, \ldots, x_{i,\vec{h}_i})\right)$ where $\vec{h}_{i+1} = \vec{h}_i - ar(\mathsf{c}) + 1$
`call` g n	$\phi_{i+1} = \bigwedge_{1 \leqslant l < \vec{h}_{i+1}} (x_{i+1,l} = x_{i,l})$ $\wedge \left(x_{i+1,\vec{h}_{i+1}} = g(x_{i,\vec{h}_{i+1}}, \ldots, x_{i,\vec{h}_i})\right)$ where $\vec{h}_{i+1} = \vec{h}_i - ar(g) + 1$
`branch c` j	$\phi_{i+1} = \bigwedge_{1 \leqslant l \leqslant \vec{h}_i - 1} (x_{i+1,l} = x_{i,l})$ $\wedge \left(x_{i,\vec{h}_i} = \mathsf{c}(x_{i+1,\vec{h}_i}, \ldots, x_{i+1,\vec{h}_{i+1}})\right)$ $\phi_j = \bigwedge_{1 \leqslant l \leqslant \vec{h}_j} (x_{j,l} = x_{i,l})$ where $\vec{h}_{i+1} = \vec{h}_i + ar(\mathsf{c}) - 1$ and $\vec{h}_j = \vec{h}_i$

Example 5 *The following example displays the* solved form *of the constraints for the code of the function add in our running example. For the sake of clarity, we use simpler variable identifiers.*

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

1	$x_1\ x_2$	: `load 1`	
2	$x_1\ x_2\ x_1$	: `branch s 7`	
3	$\mathsf{s}(x_3)\ x_2\ x_3$	: `load 2`	$\mathsf{s}(x_3) = x_1$
4	$\mathsf{s}(x_3)\ x_2\ x_3\ x_2$	: `build s 1`	
5	$\mathsf{s}(x_3)\ x_2\ x_3\ \mathsf{s}(x_2)$	: `call` *add* `2`	
6	$\mathsf{s}(x_3)\ x_2\ add(x_3, \mathsf{s}(x_2))$	: `return`	
7	$x_6\ x_2\ x_6$	: `load 2`	$x_6 = x_1$
8	$x_6\ x_2\ x_6\ x_2$	: `return`	

Thus, if we initialize a new frame for the function add with initial parameters $v_1\, v_2$, we can guarantee that if the execution reaches instruction 5 then $v_1 = \mathsf{s}(v_3)$, for some value v_3, and the value on top of *the stack is* $\mathsf{s}(v_2)$. *In particular, we may deduce that the termination of* $add(\mathsf{s}(v_3), v_2)$ *is equivalent to the termination of* $add(v_3, \mathsf{s}(v_2))$, *and that the two calls have the same outcome. We note that the shape verification is performed on the bytecode without any knowledge of the sources at the programming language level—actually, the bytecode is not necessarily obtained from the compilation of a functional program. Also, we could improve our analysis by replacing the constraint* $x_6 = x_1$ *in the shape verification of the branch instruction, by the more refined condition:* $x_6 = \mathsf{z}$. *More generally, if* $f[i] =$ `branch c` n, *we could replace the condition* $x_{j,\vec{h}_i} = x_{i,\vec{h}_i}$ *by the formula* $(x_{j,\vec{h}_i} = x_{i,\vec{h}_i}) \wedge (\forall \vec{x}\ \ (x_{j,\vec{h}_i} \neq \mathsf{c}(\vec{x}))$.

A second attempt. Next, we show that under some restrictions on the form of the code, we can solve the shape constraints and associate to every reachable instruction a substitution, $\vec{\sigma}_i$, and to every position of the related stack an expression, $e_{i,j}$. We can compare the shape verification 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 (non-optimized) compilation of functional programs. To accommodate sharing of code, the first condition could be relaxed to allow directed acyclic graphs rather than trees. These restrictions 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. In particular, thanks to condition (2), in any given path, we cannot find an instruction `call` before a `branch`. Thus, it is not possible to infer from a (shape) formula ϕ_i a constraint of the form $\big(x_{i,j} = \mathsf{c}(\ldots)\big) \wedge \big(x_{i,j} = f(\ldots)\big)$, where c is a constant and f a function symbol.

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 verification may also be used to locate dead code.

Shape Constraints at Instruction i: $wsh_i(f, \vec{\sigma}, \vec{E})$

The definition of the relation $wsh_i(f, \vec{\sigma}, \vec{E}\,)$, where $|f| = |\vec{\sigma}| = |\vec{E}|$, is by case analysis on the instruction $f[i]$. There are no constraints on $\vec{\sigma}$ and $\vec{E}$ if $f[i]$ is a `return` or `stop` instruction.

$f[i]$	$wsh_i(f, \vec{\sigma}, \vec{E}\,)$
`load` k	$\vec{\sigma}_{i+1} = \vec{\sigma}_i$ and $\vec{E}_{i+1} = \vec{E}_i \cdot \vec{E}_i[k]$
`build c` n	$\vec{\sigma}_{i+1} = \vec{\sigma}_i$, $\vec{E}_i = E \cdot e_1 \cdots e_n$ and $\vec{E}_{i+1} = E \cdot \mathsf{c}(e_1, \ldots, e_n)$
`call` g n	$\vec{\sigma}_{i+1} = \vec{\sigma}_i$, $\vec{E}_i = E \cdot e_1 \cdots e_n$ and $\vec{E}_{i+1} = E \cdot g(e_1, \ldots, e_n)$
`branch c` j	let $\vec{E}_i = E \cdot p$ in if p is a variable x let $\sigma' = [\mathsf{c}(x_{i+1,\vec{h}_i}, \ldots, x_{i+1,\vec{h}_{i+1}})/x]$ in $\vec{\sigma}_j = \vec{\sigma}_i$, $\vec{E}_j = \vec{E}_i$, $\vec{\sigma}_{i+1} = \sigma' \circ \vec{\sigma}_i$

and $\vec{E}_{i+1} = \sigma'(E) \cdot x_{i+1,\vec{h}_i} \cdots x_{i+1,\vec{h}_{i+1}}$

if $p = \mathsf{c}(e_1, \ldots, e_n)$
$\vec{\sigma}_{i+1} = \vec{\sigma}_i$ and $\vec{E}_{i+1} = E \cdot e_1 \cdots e_n$

if $p = \mathsf{d}(\ldots)$, with $\mathsf{d} \neq \mathsf{c}$
$\vec{\sigma}_j = \vec{\sigma}_i$ and $\vec{E}_j = \vec{E}_i$
where $\vec{h}_{i+1} = \vec{h}_i + ar(\mathsf{c}) - 1$ and $\vec{h}_j = \vec{h}_i$

The soundness of path 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 4 (Well-Shaped Function) *A pair $(\vec{\sigma}, \vec{E})$ is a valid shape for the function $f : (t_1, \ldots, t_n) \to t_0$, denoted $wsh(f, \vec{\sigma}, \vec{E})$, if $\vec{\sigma}_1$ is the identity substitution, id, $\vec{E}_1 = x_{1,1} \cdots x_{1,ar(f)}$, and $wsh_i(f, \vec{\sigma}, \vec{E})$ for all $i \in 1..|f|$.*

Assume we have a well-formed configuration M containing the frame (f, i, ℓ) in j^{th} position, with $\ell = v_1 \cdots v_{\vec{h}_i}$ and that $\arg(M, j) = (u_1, \ldots, u_k)$ are the parameters used to initialize this frame. The substitution $\vec{\sigma}_i$ relates the values, $u_1, \ldots, u_k$ to the values occurring in the stack ℓ. More precisely, $\vec{\sigma}_i(x_{1,l})$ is a pattern with variables in $(x_{i,j})_{j \in 1..\vec{h}_i}$ and there is at most one matching substitution ρ such that $\rho \circ \vec{\sigma}_i(x_{1,l}) = u_l$ 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 $v_j = \rho(e_{i,j})$.

Proposition 5 *Assume $(\vec{\sigma}, \vec{E})$ is a valid shape for the function f. Then for all $i \in 1..|f|$:*

1. *The sequence of patterns $\vec{\sigma}_i(x_{1,1}) \cdots \vec{\sigma}_i(x_{1,ar(f)})$ is linear (a variable appears in at most one pattern and at most once);*
2. *If $\vec{E}_i = e_1 \cdots e_{\vec{h}_i}$ and $x \in Var(e_j)$ then x occurs in one of the patterns $\vec{\sigma}_i(x_{1,j})$, for $j \in 1..ar(f)$.*

Example 6 *The shape constraints computed for the function add of the running example are as follows:*

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

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 initialize the frame.

Well-Shaped Configurations: $wsh(M)$

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

$$\frac{M \equiv (f_1, i_1, \ell_1) \ldots (f_m, i_m, \ell_m) \qquad wt(M) \qquad \vec{u}_j = arg(M, j) \qquad wsh(f_j, \vec{u}_j, i_j, \ell_j) \qquad j \in 1..m}{wsh(M)}$$

Assume the bytecode of the function f has passed the type and shape verifications. If we study the evaluation of a well-typed expression $f(v_1, \ldots, v_n)$, it is easy to check that the initial configuration $(f, 1, v_1 \cdots v_n)$ is well-shaped. Indeed, by definition, $arg(M, 1) = (v_1, \ldots, v_n)$, $\vec{\sigma}_1$ is the identity substitution and $\vec{E}_1$ is the sequence $x_{1,1} \cdots x_{1,ar(f)}$. In the same way as for type verification, our main property relies on the fact that the shape predicate is invariant under reduction.

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

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 $\vec{E}_i$ and the substitution $\vec{\sigma}_i$ are determined. We also know $\vec{h}_i$, the height of the stack at instruction i, as computed during the type verification.

A first step is to check that the size annotations given with the bytecode are correct.

Definition 5 *We say that the size annotations for the function f are correct if the following condition holds for all $i \in 1..|f|$. Assume $\vec{E}_i = e_1 \cdots e_{\vec{h}_i}$, then:*

$$\forall j \in 1..\vec{h}_i \qquad q_f(q_{\vec{\sigma}_i(x_{1,1})}, \ldots, q_{\vec{\sigma}_i(x_{1,ar(f)})}) \geqslant q_{e_j} \quad \text{over } \mathbb{R}^+ \tag{2}$$

Remark 7 *The complexity of verifying condition (2) depends on the space of quasi-interpretations selected. This problem has the same complexity as verifying the correction of the quasi-interpretation at the level of the functional language, see (1) in Section 2.3. We also notice that the condition is quite redundant. First, by the definition of quasi-interpretation the requirement is automatically verified for all instructions which are not preceded by a* `build` *or* `call` *instruction. Second, for the remaining instructions we could just perform one verification for every path that terminates with a* `return` *instruction provided that: (i) on a path terminating with a* `stop` *instruction, no* `build` *or* `call` *instructions occur and (ii) on a path terminating with a* `return` *instruction, the expressions built with* `build` *and* `call` *actually appear as subexpressions of the* returned *expression. These two conditions are needed because otherwise, a malicious bytecode could allocate on the frame large values which are not actually used.*

Next, we show (Corollary 9) 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 result follows directly from the definition of a predicate $wsz(M)$ (for well-sized) on well-shaped configurations, and an associated invariant property.

Well-Sized Configurations: $wsz(M)$

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

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

Assume the bytecode of the function f has passed the type and shape verifications. If we study the evaluation of a well-typed expression $f(v_1,\ldots,v_n)$, it is easy to check that the initial configuration $(f,1,v_1\cdots v_n)$ is well-sized. Indeed, by definition, $arg(M,1)=(v_1,\ldots,v_n)$, $\vec{\sigma}_1$ is the identity substitution and $\vec{E}_1$ is the sequence $x_{1,1}\cdots x_{1,ar(f)}$. In the same way as for type and shape verification, the main property is an invariant.

Proposition 8 *If $wsz(M)$ and $M\to M'$ then $wsz(M')$.*

Corollary 9 *Assume that all the functions in the program are well-shaped. If $f(v_1,\ldots,v_n)$ is a well-typed expression 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 8, it follows that $wsz(M\cdot(g,i,\ell))$. Let $\vec{u}=(u_1,\ldots,u_k)$ be the parameters used in the initialization of the top frame: $\vec{u}=arg(M\cdot(g,i,\ell),|M|+1)$. Since the configuration is well-sized, we have $wsz(g,\vec{u},i,\ell)$ 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) $\rho\circ\vec{\sigma}_i(x_{1,j})=u_j$ for all $j\in 1..k$,

(c3) $wsh(g,\vec{\sigma},\vec{E})$ and $\vec{E}_i=e_1\cdots e_n$ 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 (2) we have: $q_{g(\vec{\sigma}_i(x_{1,1}),\ldots,\vec{\sigma}_i(x_{1,k}))}\geqslant q_{e_j}$ for all $j\in 1..n$. Thus:

$$\begin{array}{lll} q_{f(v_1,\ldots,v_n)} & \geqslant q_{g(u_1,\ldots,u_k)} & \text{by (c1)} \\ & = q_{g(\rho\circ\vec{\sigma}_i(x_{1,1}),\ldots,\rho\circ\vec{\sigma}_i(x_{1,k}))} & \text{by (c2)} \\ & \geqslant q_{\rho(e_j)} & \text{by (2) and monotonicity} \\ & \geqslant q_v & \text{by (c3)} \\ & \geqslant |v| & q_v \text{ is a quasi-interpretation ,} \end{array}$$

as needed. □

7 Termination Verification

In this section, we adapt recursive path orderings, a popular technique for checking termination (see, *e.g.*, [BN98]), to prove termination of the evaluation of the virtual machine. We suppose that the code succeeds the shape verification. 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* to 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', that is, on values the relation $>_l$ coincides with the relation generated by the following two rules. Hence, $v >_l v'$ implies $|v| > |v'|$.

$$\frac{s \geqslant t}{\mathsf{c}(\ldots, s, \ldots) > t} \qquad \frac{s_i \geqslant t_i \quad i \in 1..n \quad \exists j \ \in 1..n\,(s_j > t_j)}{\mathsf{c}(s_1, \ldots, s_n) > \mathsf{c}(t_1, \ldots, t_n)}$$

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

$$\forall j \in 1..\vec{h}_i \qquad f\big(\vec{\sigma}_i(x_{1,1}), \ldots, \vec{\sigma}_i(x_{1,ar(f)})\big) >_l \ e_i \tag{3}$$

Next, we introduce a predicate *ter* (for terminating) on well-shaped configurations M.

Termination Invariant on Configurations: $ter(M)$

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

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

Like with value size verification, it is easy to check that the initial configuration $(f, 1, v_1 \cdots v_n)$ corresponding to the evaluation of a well-typed expression $f(v_1, \ldots, v_n)$ satisfies the predicate *ter*. As expected, the termination predicate is an invariant.

Proposition 10 *If $ter(M)$ and $M \to M'$ then $ter(M')$.*

Corollary 11 *Assume that all the functions in the program have correct termination information. Then every execution starting with a frame $(f, 1, v_1 \cdots v_n)$ terminates.*

Proof. We define a well-ordering 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 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 to a configuration M the following measure:

$$\begin{aligned}
\mu(\epsilon) &=_{\text{def}} \emptyset \\
\mu\big((f_1, i_1, \ell_1) \cdots (f_m, i_m, \ell_m)\big) &=_{\text{def}} \{\!|(f_1\, arg(M,1), acc(i_1) - 1), \ldots, \\
&\qquad (f_{m-1}\, arg(M, m-1), acc(i_{m-1}) - 1), \\
&\qquad (f_m\, arg(M,m), acc(i_m))|\!\}
\end{aligned}$$

We are only left to check that all the reduction rules decrease this measure. The proof is by case analysis on the instruction $f_m[i_m]$. Assume $f_m[i_m] = \texttt{call}\ g\ n$. An element $(f(\vec{v}), i)$ of the multiset is replaced by the two elements $(f(\vec{v}), i-1)$ and $(g(\vec{u}), acc(1))$, where $f(\vec{v}) >_l g(\vec{u})$ (by the invariant *ter*) so that, with respect to the lexicographic order:

$$(f(\vec{v}), i) > (f(\vec{v}), i-1) \quad \text{and} \quad (f(\vec{v}), i) > (g(\vec{u}), acc(1))$$

In the other cases, an element $(f(\vec{v}), i)$ is either removed or replaced by $(f(\vec{v}), j)$ with $i > j$, as needed. □

As observed in [BMM01], 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 12 *Suppose that the quasi-interpretations are bound by polynomials and that the bytecode succeeds the value size and termination verifications. Then every execution starting from a frame $(f, 1, v_1 \cdots v_n)$ (terminates and) runs in space polynomial in the size of the arguments $|v_1|, \ldots, |v_k|$.*

Proof. Note that if $f(\vec{v}) >_l g(\vec{u})$ then either $f >_\Sigma g$ or $f =_\Sigma g$ and $\vec{v} >_l \vec{u}$. In a sequence $f_1(\vec{v}_1) >_l \cdots >_l f_m(\vec{v}_m)$, the first case can occur a constant number of times (the number of equivalence classes of function symbols with respect to $\geqslant_\Sigma$) thus it is enough to 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. □

8 Example

We consider a classical example: a program that checks the validity of a quantified propositional formula. The source code in the syntax of a prototype compiler is displayed below

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

not (T) -> F        or (T,y) -> T       eq (Z,Z)        -> T
not (F) -> T        or (F,T) -> T       eq (Z,S(y))     -> F
```

```
                    or (F,F) -> F       eq (S(x),S(y)) -> eq(x,y)

member (x,Nil)     -> F
member (x,C(y,l))  -> or(eq(x,y),member(x,l))

check (Var(x), l)    -> member(x,l)
check (Not(f1), l)   -> not(check(f1,l))
check (Or(f1,f2), l) -> or(check(f1,l),check(f2,l))
check (Ex(x,f1), l)  -> or(check(f1,l),check(f1, C(x,l)))

qbf (f) -> check(f,Nil)
```

Types, quasi-interpretations, and orders for the program are given below. We assume *check* has lexicographic status from left to right and the remaining functions have product status. In the specific case, this information could be obtained automatically. It is interesting to note that in a recursive call of the *check* function the size of the first argument is always decreased. This means that the number of frames that the *check* function allocates on the stack is *linear* in the size of the input data rather than quadratic as foreseen by Corollary 12. Clearly, we could perform a finer analysis of the termination order to have a better bound on the memory usage.

```
not   : bool -> bool            or     : (bool, bool) -> bool
eq    : (nat,  nat) -> nat      member : (nat,   env) -> bool
check : (form, env) -> bool     qbf    : form -> bool

q_S = q_Var = q_Not = x + 1     q_C = q_Or = q_Ex = x + y + 1
q_not = q_qbf = x               q_or = q_eq = max(x,y)
q_check = x + y

qbf > check > member > not, or, eq
```

Next, we display the bytecode resulting from a possible compilation of the function `check` as well as the result of the type verification. (We leave a blank line between the different call blocks.)

```
1  : load 1             form env
2  : branch Var 7       form env form
3  : load 3             form env nat
4  : load 2             form env nat nat
5  : call member 2      form env nat nat env
6  : return             form env nat bool

7  : branch Not 13      form env form
8  : load 3             form env form
9  : load 2             form env form form
10 : call check 2       form env form form env
11 : call not 1         form env form bool
12 : return             form env form bool

13 : branch Or 22       form env form
14 : load 3             form env form form
15 : load 2             form env form form form
16 : call check 2       form env form form form env
```

```
17 : load 4              form env form form bool
18 : load 2              form env form form bool form
19 : call check 2        form env form form bool form env
20 : call or 2           form env form form bool bool
21 : return              form env form form bool

22 : branch Ex 33        form env form
23 : load 4              form env nat form
24 : load 2              form env nat form form
25 : call check 2        form env nat form form env
26 : load 4              form env nat form bool
27 : load 3              form env nat form bool form
28 : load 2              form env nat form bool form nat
29 : build C             form env nat form bool form nat env
30 : call check 2        form env nat form bool form env
31 : call or 2           form env nat form bool bool
32 : return              form env nat form bool

33 : stop                form env form
```

We also give the result of the path verification for the first block of instructions, corresponding to the compilation of the first rule of *check*, that is: `check(Var(x), l) = member(x, l)`.

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

For this block, the relevant conditions that need to be checked for proving that the size and termination annotations are correct are:

$$q_{\mathit{check}(\mathit{Var}(x_{3.3}),x_{1,2})} \geqslant q_{\mathit{member}(x_{3,3},x_{1,2})} \qquad \text{(size condition)}$$

$$\mathit{check}(\mathit{Var}(x_{3.3}), x_{1,2}) >_l \mathit{member}(x_{3,3}, x_{1,2}) \qquad \text{(termination condition)}$$

9 Conclusion

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

In this paper, we study the resource bounds problem in a simple stack machine and show how to perform type, size, and termination verifications at the level of the bytecode. In particular, we show that a combination of size verification based on quasi-interpretations and of termination verification based on lexicographic path orders leads to an explicit polynomial bound on the space required for the execution. 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 formalizing 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 optimized implementation of the virtual machine. Yet in another direction, we could consider an extended machine that interprets the annotations carried by unverified bytecode as directives to be enforced during the execution.

References

[Ama03] R. Amadio. Max-plus quasi-interpretations. In *Proc. Typed Lambda Calculi and Applications (TLCA '03)*, LNCS 2701, Springer, 2003.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BMM01] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Andrei Ershov Fourth International Conference "Perspectives of System Informatics", LNCS*. Springer, 2001.

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[Cob65] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II, North Holland*, 1965.

[Gra96] B. Gramlich. On proving termination by innermost termination. In *Proc. 7th Int. Conf. on Rewriting Techniques and Applications (RTA '96)*, volume 1103 of *LNCS*, pp. 93–107. Springer, 1996.

[Hof02] M. Hofmann. The strength of non size-increasing computation. In *Proc. POPL*, ACM Press, 2002.

[Jon97] N. Jones. *Computability and complexity, from a programming perspective.* MIT-Press, 1997.

[Lei94] D. Leivant. Predicative recurrence and computational complexity i: word recurrence and polytime. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser:320–343, 1994.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1999.

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

[MM03] J.-Y. Marion, J.-Y. Moyen. *Termination and resource analysis of assembly programs by Petri Nets.* Technical Report, Université de Nancy, 2003.

[MWCG99] 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.

[Nec97] G. Necula. Proof carrying code. In *Proc. POPL*, ACM Press, 1997.

[San01] D. Sannella. Mobile resource guarantee. Ist-global computing research proposal, U. Edinburgh, 2001. `http://www.dcs.ed.ac.uk/home/mrg/`.

A Proof of Proposition 3: Type invariant

Proposition 3 Type Invariant Let M be a configuration. If $wt(M)$ and $M \to M'$ then $wt(M')$.

Proof. By case analysis on the rule applied in the derivation of $M \to M'$. Assume $M \equiv M_0 \cdot (f, i, \ell)$. Since $wt(M)$, we have $wt(f, \vec{T})$ for some valid abstract execution $\vec{T}$ and $\ell : \vec{T}_i$ with $i \in 1..|\vec{T}|$.

(Load) we have $f[i] = \mathtt{load}\, k$. Immediate since $\vec{T}_{i+1}[\vec{h}_i + 1] = \vec{T}_i[k]$.

(Build) we have $f[i] = \mathtt{build}\,\mathsf{c}\, k$ and $\mathsf{c} : (t_1, \ldots, t_k) \to t_0$, where $\vec{T}_i = T \cdot t_1 \cdots t_k$, and $\ell = \ell' \cdot v_1 \cdots v_k$ and $M' = M_0 \cdot (f, i+1, \ell' \cdot \mathsf{c}(v_1, \ldots, v_k))$. Then $v_i : t_i$ for all $i \in 1..k$ and the property follows since $\vec{T}_{i+1}[\vec{h}_{i+1}] = T \cdot t_0$.

(Call) we have $f[i] = \mathtt{call}\, g\, k$ and $g : (t_1, \ldots, t_k) \to t_0$, where $\vec{T}_i = T \cdot t_1 \cdots t_k$, $\ell = \ell' \cdot v_1 \cdots v_k$, and $M' = M \cdot (g, 1, v_1 \cdots v_k)$. By definition, g must be well-typed and $v_i : t_1$ for all $i \in 1..k$. Then $wt(g, 1, v_1 \cdots v_k)$ and M' is well-formed, as needed.

(Return) we have $f[i] = \mathtt{return}$ and $M \equiv M_1 \cdot (g, i', \ell'' \cdot v_1 \cdots v_k) \cdot (f, i, \ell' \cdot v) \to M_1 \cdot (g, i'+1, \ell'' \cdot v)$. Since M is well-typed (and well-formed), the vector $arg(M, m)$ is defined (where $m = |M|$), the function g is well-typed and $g[i'] = \mathtt{call}\ f\ k$ with $f : t_1, \ldots, t_k \to t_0$ and $v : t_0$. Assume $\vec{T'}$ is the type of g, the property follows since $\vec{T}'_{i'+1}[\vec{h}'_{i'} + 1] = t_0$.

(Stop) we have $f[i] = \mathtt{stop}$ and $M' \equiv \epsilon$. By definition, $wt(\epsilon)$.

(BranchThen) we have $f[i] = \mathtt{branch}\,\mathsf{c}\, j$, $\ell = \ell' \cdot \mathsf{c}(v_1, \ldots, v_k)$, and $M' \equiv M_0 \cdot (f, i+1, \ell \cdot v_1 \cdots v_k)$ where $\mathsf{c} : (t_1, \ldots, t_k) \to t_0$ and $t_0 = \vec{T}_i[\vec{h}_i]$. Then $v_i : t_i$ for all $i \in 1..k$ and the property follows by definition of $\vec{T}_{i+1}$.

(BranchElse) we have $f[i] = \mathtt{branch}\,\mathsf{c}\, j$ and $\ell = \ell' \cdot \mathsf{d}(\ldots)$ where $\mathsf{c} \neq \mathsf{d}$ and $M' \equiv M_0 \cdot (f, j, \ell)$. The property follows since $\vec{T}_j = \vec{T}_i$.

□

B Proof of Proposition 4: Progress

Proposition 4 (Progress) Assume M is a well-typed configuration. Then either (1) $M \equiv \epsilon$, or (2) M is a result, or (3) M reduces, $\exists M\ '(M \to M')$.

Proof. If M is not the error configuration, ϵ, we have $M \equiv M_0 \cdot (f, i, \ell)$. Since M is well-typed (and well-formed), we have $wt(f, \vec{T})$ for some valid abstract execution $\vec{T}$ and $\ell : \vec{T}_i$ with $i \in 1..|\vec{T}|$. In particular, the instruction $f[i]$ is defined and $|\ell| = |\vec{T}_i|$. The proof proceeds by case analysis on $f[i]$.

(Load) we have $f[i] = \mathtt{load}\ k$ and $|\ell| \geqslant k$ and $i < |f|$. Hence rule (Load) applies and M reduces.

(Build) we have $f[i] = \mathtt{build}\ \mathsf{c}\ n$ and $|\ell| \geqslant ar(\mathsf{c})$ and $i < |f|$. Hence rule (Build) applies and M reduces.

(Call) we have $f[i] = \mathtt{call}\ g\ n$ and $|\ell| \geqslant ar(g)$ and $i < |f|$. Hence rule (Call) applies and M reduces.

(Return) we have $f[i] = \mathtt{return}$. Let $m = |M|$. There are two different cases: either $m = 1$ and M is a result, or $m \geqslant 2$ and, since M is well-formed, the vector $arg(M, m)$ is defined. In the latter case, we have $M \equiv M_1 \cdot (g, i', \ell') \cdot (f, i, \ell)$ with $|\ell| \geqslant 1$ and $|\ell'| \geqslant ar(f)$. Hence rule (Return) applies and M reduces.

(Stop) By definition M reduces to ϵ.

(Branch) we have $f[i] = \mathtt{branch}\ \mathsf{c}\ j$ and $i < |f|$ and $j \in 1..|f|$. There are two different cases subject to the value on top of the stack ℓ. Assume $\ell = \ell' \cdot v$ then either (1) $v = \mathsf{c}(v_1, \ldots, v_n)$ and rule (BranchThen) applies or (2) $v = \mathsf{d}(\ldots)$ with $\mathsf{c} \neq \mathsf{d}$ and rule (BranchElse) applies. Therefore, in both cases the configuration M reduces. □

C Proof of Proposition 5: Shape Substitutions

In the following proofs, we apply the restrictions on the flow graph given in Sections 4 and 5: for every function, f, appearing in the code of the program we have that (C1) every node in the flow graph E_f is accessible; (C2) the flow graph E_f is a tree rooted at instruction 1 whose leaves correspond to the instructions `return` or `stop`; and (C3) on any given path from the root, every `branch` instruction is preceded only by `load` or `branch` instructions.

Proposition 5 Assume $(\vec{\sigma}, \vec{E})$ is a valid shape for the function f. Then for all $i \in 1..|f|$:

1. the sequence of patterns $\vec{\sigma}_i(x_{1,1}) \cdots \vec{\sigma}_i(x_{1,ar(f)})$ is linear (a variable appears in at most one pattern and at most once);
2. assume $\vec{E}_i = e_1 \cdots e_{\vec{h}_i}$, if $x \in \mathit{Var}(e_j)$ then x occurs in one of the patterns $\vec{\sigma}_i(x_{1,j})$, for $j \in 1..ar(f)$.

Proof. By induction on the length of the path from 1 to i. Since $\vec{\sigma}_1 = id$ and $e_{1,j} = x_{1,j}$ for all $j \in 1..ar(f)$, properties (1) and (2) of Proposition 5 are true in the case where $i = 1$. In the cases where $i \neq 1$, the proof proceeds by case analysis of the instruction associated to the unique immediate predecessor of i in the tree E_f. Assume i' is the predecessor of i.

(Load) we have $f[i'] = \texttt{load}\ k$ and $i = i' + 1$. Since $\mathit{wsh}_{i'}(f, \vec{\sigma}, \vec{E})$, we have $\vec{\sigma}_i = \vec{\sigma}_{i'}$ and $\vec{E}_i = \vec{E}_{i'} \cdot \vec{E}_{i'}[k]$. Hence properties (1) and (2) hold by inductive hypothesis.

(Build) we have $f[i'] = \texttt{build}\ \mathsf{c}\ n$ and $i = i' + 1$, where $\mathsf{c} : (t_1, \ldots, t_n) \to t_0$. Since $\mathit{wsh}_{i'}(f, \vec{\sigma}, \vec{E})$, we have $\vec{\sigma}_i = \vec{\sigma}_{i'}$ and $\vec{E}_{i'} = E \cdot e_1 \cdots e_n$ and $\vec{E}_i = E \cdot \mathsf{c}(e_1, \ldots, e_n)$. Hence properties (1) and (2) hold by inductive hypothesis. Note that $\mathit{Var}(e_{i,\vec{h}_i}) = \mathit{Var}(e_{i',\vec{h}_i - ar(\mathsf{c})+1}) \cup \cdots \cup \mathit{Var}(e_{i,\vec{h}_i})$. The proof is similar if the preceding instruction is a function call, `call` $g\ n$.

(Branch) we have $f[i'] = \texttt{branch}\ \mathsf{c}\ j$ and we need to consider two possible cases corresponding to the two possible outcomes of a branch instruction. Either instruction i directly follows the branch instruction and $i = i' + 1$, or it is the target of the branch instruction, in which case $j = i$.

Assume $\vec{E}_{i'} = E \cdot p$ where p is a variable x. In the first case, we have $\vec{\sigma}_i = \sigma' \circ \vec{\sigma}_{i'}$ and $\vec{E}_i = \sigma'(E) \cdot x_{i,\vec{h}_i - ar(\mathsf{c})+1} \cdots x_{i,\vec{h}_i}$ where σ' is the substitution $[\mathsf{c}(x_{i,\vec{h}_i - ar(\mathsf{c})+1}, \ldots, x_{i,\vec{h}_i})/x]$. We start by proving property (1). By inductive hypothesis, there is a unique index k such that x occurs once in $\vec{\sigma}_i x_{1,k}$ and therefore:

$$\vec{\sigma}_i(x_{1,1}) \cdots \vec{\sigma}_i(x_{1,k}) \cdots = \vec{\sigma}_{i'}(x_{1,1}) \cdots \sigma'(\vec{\sigma}_{i'}(x_{1,k})) \cdots$$

Since the variables $(x_{i,j})_{j \in 1..\vec{h}_i}$ are fresh and distinct the patterns occurring in this sequence are linear, as needed. For property (2), suppose $y \in \mathit{Var}(e_{i,j})$ for some $j \in 1..\vec{h}_i$. The property follows by the fact that $y \neq x$ and that either y occurs in $e_{i',j'}$ for some j' or it belongs to the fresh variables $x_{i,j}$ introduced in $\vec{E}_i$ which do occur in the related linear pattern.

If p is a variable x and i is the target of the branch instruction, $i = j$ (since E_f is a tree we have $i' + 1 \neq j$), then $\vec{\sigma}_i = \vec{\sigma}_{i'}$ and $\vec{E}_i = \vec{E}'_i$ and properties (1) and (2) hold immediately by inductive hypothesis.

If p is a pattern $\mathsf{c}(e_1, \ldots, e_n)$ then $i = i' + 1$ and $\vec{\sigma}_i = \vec{\sigma}_{i'}$ and $\vec{E}_i = E \cdot e_1 \cdots e_n$. The proof is similar to the case where the preceding instruction is a build.

The remaining case is for p a pattern of the from $\mathsf{d}(\ldots)$ with $\mathsf{d} \neq \mathsf{c}$. In this case $i = j$, $\vec{\sigma}_i = \vec{\sigma}_{i'}$ and $\vec{E}_i = \vec{E}'_i$. Properties (1) and (2) hold immediately by inductive hypothesis. $\square$

D Proof of Proposition 6: Shape Invariant

Proposition 6 Let M be a stack. If $\mathit{wsh}(M)$ and $M \to M'$ then $\mathit{wsh}(M')$.

Proof. By case analysis on the rule applied in the derivation of $M \to M'$. Assume $M \equiv M_0 \cdot (f, i, \ell)$. Since $wsh(M)$, we have $wsh(f, \vec{\sigma}, \vec{E})$ for some valid shape $(\vec{\sigma}, \vec{E})$ and if $\vec{E}_i[j]$ is a pattern then $\ell[j] = \rho(\vec{E}_i[j])$ where ρ is the solution of $match\big((\vec{\sigma}_i(x_{1,1}), \ldots, \vec{\sigma}_i(x_{1,ar(f)})), \vec{u}\big)$, $\vec{u} = arg(M, m)$, and $m = |M|$.

(Load) we have $f[i] = \texttt{load}\ k$ and $\vec{\sigma}_{i+1} = \vec{\sigma}_i$ and $\vec{E}_{i+1} = \vec{E}_i \cdot \vec{E}_i[k]$. Hence the substitution ρ is also solution of the matching of $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$ with the vector $\vec{u}$ and if $\vec{E}_{i+1}[j]$ is a pattern then the j^{th} element of $\ell \cdot \ell[k]$ is equal to $\rho(\vec{E}_{i+1}[j])$. Therefore $wsh(f, \vec{u}, i+1, \ell \cdot \ell[k])$, as needed. The proof is similar in the case of rule (Build).

(Call) we have $f[i] = \texttt{call}\ g\ n$, where $g : (t_1, \ldots, t_n) \to t_0$, ℓ is of the form $\ell_0 \cdot v_1 \cdots v_k$, and $M' = M \cdot (g, 1, v_1 \cdots v_k)$. Since M is well-shaped, the proposition follows from the fact that $wsh(g, v_1 \cdots v_k, 1, v_1 \cdots v_k)$. This property is immediate since, by definition, $\vec{\sigma}_1 = id$ and we can take $\rho = [v_1/x_{1,1}, \ldots, v_k/x_{1,k}]$.

(Return) we have $f[i] = \texttt{return}$ and $M \equiv M_1 \cdot (g, i', \ell_g \cdot \ell') \cdot (f, i, \ell_f \cdot v) \to M_1 \cdot (g, i'+1, \ell_g \cdot v)$ where $\ell' = v_1 \cdots v_k$ and the instruction $g[i']$ is $\texttt{call}\ f\ k$ with $f : (t_1, \ldots, t_k) \to t_0$. Since M is well-shaped, we know that $wsh(g, \vec{\sigma}', \vec{E}')$ and $wsh(g, \vec{u}, i', \ell_g \cdot \ell')$ where $\vec{u} = arg(M, m-1)$ are the parameters used to initialize the frame for g. In this case, $\vec{\sigma}'_{i'+1} = \vec{\sigma}'_{i'}$ and $\vec{E}'_{i'+1}$ contains at most the patterns in $\vec{E}'_{i'}$: the only new element is the expression $\vec{E}'_{i'+1}[\vec{h}_{i'+1}]$ that is of the form $f(\ldots)$. Hence $wsh(g, \vec{u}, i'+1, \ell_g \cdot v)$, as needed.

(Stop) we reduce to the error configuration, ϵ, which satisfies wsh by definition.

(BranchThen) we have $f[i] = \texttt{branch}\ \mathsf{c}\ j$ and $\ell = \ell_0 \cdot \mathsf{c}(v_1, \ldots, v_k)$ and $M' \equiv M_0 \cdot (f, i+1, \ell_0 \cdot v_1 \cdots v_k)$ where $\mathsf{c} : t_1, \ldots, t_k \to t_0$. Assume $arg(M, m)$ is the vector $\vec{u} = (u_1, \ldots, u_{ar(f)})$. Since M is well-shaped, we also know that $(\rho \circ \vec{\sigma}_i)(x_{1,j}) = u_j$ for all $j \in 1..ar(f)$, $\rho(\vec{E}_i[\vec{h}_i + j - 1]) = v_j$ if $1 \leqslant j \leqslant ar(\mathsf{c})$, and $\vec{E}_i[\vec{h}_i + j - 1]$ is a pattern.

Let p be the pattern $\vec{E}_i[\vec{h}_i]$, which describes the shape of the value tested in the branch statement. We have three different cases to consider depending on the form of p.

If p is a variable x, then $\vec{\sigma}_{i+1} = \sigma' \circ \vec{\sigma}_i$ and $\vec{E}_{i+1} = \sigma'(\vec{E}_i) \cdot x_{i+1,\vec{h}_i} \cdots x_{i+1,\vec{h}_{i+1}}$ where σ' is the substitution $[\mathsf{c}(x_{i+1,\vec{h}_i}, \ldots, x_{i+1,\vec{h}_{i+1}})/x]$. Let ρ' be the substitution $[v_1/x_{i+1,\vec{h}_i}, \ldots, v_k/x_{i+1,\vec{h}_{i+1}}] \circ \rho$. We verify that (i) ρ' is the solution of matching the patterns $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$ with the values in $\vec{u}$, (ii) $\rho'(\vec{E}_{i+1}[j]) = (\rho' \circ \sigma')(\vec{E}_i[j]) = \ell_0[j]$ if $1 \leqslant j < \vec{h}_i$ and $\vec{E}_{i+1}[j]$ is a pattern, and (iii) $\rho'(\vec{E}_{i+1}[\vec{h}_i + j - 1]) = v_j$ for all $j \in 1..ar(\mathsf{c})$. Hence, $wsh(f, \vec{u}, i+1, \ell_0 \cdot v_1 \cdots v_k)$, as needed.

If $p = \mathsf{c}(e_1, \ldots, e_k)$ then $\vec{\sigma}_{i+1} = \vec{\sigma}_i$ and $\vec{E}_{i+1} = \vec{E}_i \cdot e_1 \cdots e_k$. Hence the substitution ρ is also solution of the matching of $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$ with the sequence ℓ' and if $\vec{E}_{i+1}[j]$ is a pattern then the j^{th} element of $\ell_0 \cdot v_1 \cdots v_k$ is equal to $\rho(\vec{E}_{i+1}[j])$. Therefore $wsh(f, \ell', i+1, \ell_0 \cdot v_1 \cdots v_k)$, as needed.

The remaining cases are $p = f(\ldots)$ and $p = \mathsf{d}(\ldots)$, where $\mathsf{c} \neq \mathsf{d}$. The first case cannot arises since only `call` instructions may introduce function symbols in the shape constraints and a `branch` instruction cannot be preceded by a `call` (see restriction (C3) in Appendix C). The latter case is also impossible since, by M well-shaped, we have $\rho(p) = \mathsf{c}(v_1, \ldots, v_k)$.

The proof is similar in the case of rule (BranchElse). □

E Proof of Proposition 8: Size invariant

Proposition 8 Let M be a stack. If $wsz(M)$ and $M \to M'$ then $wsz(M')$.

Proof. By case analysis on the rule applied in the derivation of $M \to M'$. Assume $M \equiv M_0 \cdot (f, i, \ell)$. Since $wsz(M)$, we have $wsh(f, \vec{\sigma}, \vec{E})$ for some valid shape $(\vec{\sigma}, \vec{E})$ and there exists a substitution ρ solution of matching the patterns $(\vec{\sigma}_i(x_{1,1}), \ldots, \vec{\sigma}_i(x_{1,ar(f)}))$ with the values in $\vec{u}$, where $\vec{u} = arg(M, m)$ and $m = |M|$.

(Load) we have $f[i] = \texttt{load}\ k$ and $M' \equiv M_0 \cdot (f, i+1, \ell \cdot \ell[k])$ with $k \leqslant |\ell|$ and $\vec{\sigma}_{i+1} = \vec{\sigma}_i$ and $\vec{E}_{i+1} = \vec{E}_i \cdot \vec{E}_i[k]$. Hence the substitution ρ is also solution of matching $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$

with the vector $\vec{u}$ and for all $j \in 1..\vec{h}_i + 1$ we have $q_{\rho(e_j)} \geqslant q_{v_j}$: the only new inequality is for $j = \vec{h}_i + 1$ in which case $\rho(e_j) = \rho(e_k)$ and $v_j = v_k$. Therefore $wsz(f, \vec{u}, i+1, \ell \cdot \ell[k])$, as needed.

(Build) we have $f[i] = \texttt{build c}\ k$, where $\texttt{c} : t_1, \ldots, t_k \to t_0$, and $\ell = \ell_0 \cdot v_1 \cdots v_k$ and $M' = M_0 \cdot (f, i+1, \ell_0 \cdot \texttt{c}(v_1, \ldots, v_k))$. Since M is well-shaped we have $\vec{E}_i = E \cdot e_1 \cdots e_k$ where $e_1, \ldots, e_k$ are expressions associated with the values $v_1, \ldots, v_k$ and $\vec{E}_{i+1} = E \cdot \texttt{c}(e_1, \ldots, e_k)$. The proposition follows by proving that $wsz(f, \vec{u}, i+1, \ell_0 \cdot \texttt{c}(v_1, \ldots, v_k))$. As for the previous case, the substitution ρ is also solution of the matching between the shape patterns and the values in $\vec{u}$. Hence we are left with proving that $q_{\rho(\texttt{c}(e_1,\ldots,e_k))} \geqslant q_{\texttt{c}(v_1,\ldots,v_k)}$, which is a direct consequence of the monotonicity of $q_{\texttt{c}}$ and the fact that $q_{\rho e_i} \geqslant q_{v_i}$ for all $i \in 1..k$.

(Call) we have $f[i] = \texttt{call}\ g\ k$, where $g : (t_1, \ldots, t_k) \to t_0$, and ℓ is of the form $\ell_0 \cdot v_1 \cdots v_k$ and $M' = M \cdot (g, 1, v_1 \cdots v_k)$. Since M is well-sized, the proposition follows by proving $wsz(g, (v_1, \ldots, v_k), 1, v_1 \cdots v_k)$ and $q_{g(v_1,\ldots,v_k)} \leqslant q_{f(\vec{u})}$.

The validity of the invariant for the frame g is immediate since $wsh(g, \vec{\sigma}', \vec{E}')$ for some valid shape $(\vec{\sigma}', \vec{E}')$ such that, by definition, $\vec{\sigma}'_1 = id$ and $\vec{E}'_1 = x_{1,1} \cdots x_{1,k}$. Indeed, the solution for the pattern-matching is the substitution $\rho' = [v_1/x_{1,1}, \ldots, v_k/x_{1,k}]$ and we have $\rho(x_{1,i}) = v_i$ for all $i \in 1..k$.

To prove the validity of the inequality, we use the fact that size annotations are correct (see condition (2) in Definition 5). Let $e_1, \ldots, e_k$ be the expressions associated to $v_1, \ldots, v_k$ in $\vec{E}_1$. Condition (2) applied to instruction $i+1$ leads to the following relation:

$$q_{f(\vec{\sigma}_{i+1} x_{1,1}, \ldots, \vec{\sigma}_{i+1} x_{1,ar(f)})} \geqslant q_{\vec{E}_{i+1}[\vec{h}_{i+1}]} = q_{g(e_1,\ldots,e_k)} \ ,$$

and since $\vec{\sigma}_{i+1} = \vec{\sigma}_i$, we obtain:

$$\begin{aligned} q_{f(\vec{u})} &= q_{f(\rho \circ \vec{\sigma}_i(x_{1,1}), \ldots, \rho \circ \vec{\sigma}_i(x_{1,ar(f)}))} \\ &\geqslant q_{\rho g(e_1,\ldots,e_k)} \\ &\geqslant q_{g(v_1,\ldots,v_k)} \ . \end{aligned}$$

(Return) we have $f[i] = \texttt{return}$ and $M \equiv M_1 \cdot (g, i', \ell_g \cdot \ell') \cdot (f, i, \ell_f \cdot v)$ and $M' \equiv M_1 \cdot (g, i'+1, \ell_g \cdot v)$ where $\ell' = v_1 \cdots v_k$ and the instruction $g[i']$ is $\texttt{call}\ f\ k$ with $f : (t_1, \ldots, t_k) \to t_0$. Let $\vec{u}' = (v_1, \ldots, v_k) = arg(M, |M|)$. Since M is well-shaped, we know that $wsh(g, \vec{\sigma}', \vec{E}')$ and $wsh(g, \vec{u}, i', \ell_g \cdot \ell')$ where $\vec{u} = arg(M, m-1)$ are the parameters used to initialize the frame for g. We also know that $\vec{\sigma'}_{i'+1} = \vec{\sigma}'_{i'}$, $\vec{E}'_{i'} = E' \cdot e'_1 \cdots e'_k$ and $\vec{E}'_{i'+1} = E' \cdot f(e'_1, \ldots, e'_k)$.

The proposition follows by proving that $wsz(g, \vec{u}, i'+1, \ell_g \cdot v)$. More particularly, if ρ' denotes the solution of the pattern-matching for the frame g, the same substitution is also solution for the resulting frame and we only need to prove that $q_{\rho'(f(e'_1,\ldots,e'_k))} \geqslant q_v$.

Assume e is the expression $\vec{E}_i[\vec{h}_i]$, which describes the shape of the value returned by the frame f. By condition (2) on *correct size annotations*, we know that $q_{f(\vec{\sigma}_i(x_{1,1}),\ldots,\vec{\sigma}_i(x_{1,k}))} \geqslant q_e$. Since the (decorated) frame $(f, \vec{u}', i, \ell_f \cdot v)$ is well-sized, we also know that $q_{\rho(e)} \geqslant q_v$ and $\rho \circ \vec{\sigma}_i(x_{1,j}) = v_j$ for all $j \in 1..k$. Therefore we have $(c1)$ $q_{f(v_1,\ldots,v_k)} \geqslant q_v$. From the fact that the frame g is well-sized, we know that $(c2)$ $q_{\rho'(e'_j)} \geqslant q_{v_j}$ for all $j \in 1..k$, and therefore:

$$\begin{aligned} q_{\rho'(f(e'_1,\ldots,e'_k))} &= q_f(q_{\rho'(e'_1)}, \ldots, q_{\rho'(e'_k)}) \\ &\geqslant q_f(q_{v_1}, \ldots, q_{v_k}) && \text{by } (c2) \\ &= q_{f(v_1,\ldots,v_k)} \\ &\geqslant q_v && \text{by } (c1) \end{aligned}$$

(Stop) we reduce to the error configuration, ϵ, which satisfies *wsz* by definition.

(BranchThen) we have $f[i] = \texttt{branch c}\ j$ and $\ell = \ell_0 \cdot \texttt{c}(v_1, \ldots, v_k)$ and $M' \equiv M_0 \cdot (f, i+1, \ell_0 \cdot v_1 \cdots v_k)$ where $\texttt{c} : t_1, \ldots, t_k \to t_0$. From the proof of Proposition 6, Appendix D, we know that there exists a substitution ρ', solution of matching the patterns $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$ with the values in ℓ', such that $\rho'(\vec{E}_{i+1}[\vec{h}_i + j - 1]) = v_j$ for all $j \in 1..ar(\mathsf{c})$. Hence $q_{\rho'(\vec{E}_{i+1}[\vec{h}_i+j-1])} \geqslant q_{v_j}$ and we have $wsz(f, \ell', i+1, \ell \cdot v_1 \cdots v_k)$ as needed.

The proof is similar in the case of rule (BranchElse). □

F Proof of Proposition 10: Termination Invariant

Proposition 10 If $ter(M)$ and $M \to M'$ then $ter(M')$.

Proof. By case analysis on the rule applied in the derivation of $M \to M'$. Assume $M \equiv M_0 \cdot (f, i, \ell)$. Since $ter(M)$, we have $wsh(f, \vec{\sigma}, \vec{E})$ for some valid shape $(\vec{\sigma}, \vec{E})$ and there exists a substitution ρ solution of matching the patterns $(\vec{\sigma}_i(x_{1,1}), \ldots, \vec{\sigma}_i(x_{1,ar(f)}))$ with the values in $\vec{u}$, where $\vec{u} = arg(M, m)$ and $m = |M|$.

(Load) we have $f[i] = \texttt{load}\ k$ and $M' \equiv M_0 \cdot (f, i+1, \ell \cdot \ell[k])$ with $k \leqslant |\ell|$ and $\vec{\sigma}_{i+1} = \vec{\sigma}_i$ and $\vec{E}_{i+1} = \vec{E}_i \cdot \vec{E}_i[k]$. Hence the substitution ρ is also solution of matching $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$ with the vector $\vec{u}$ and for all $j \in 1..\vec{h}_i + 1$ we have $\rho(e_j) \geqslant_l v_j$: the only new inequality is for $j = \vec{h}_i + 1$ in which case $\rho(e_j) = \rho(e_k)$ and $v_j = v_k$. Therefore $ter(f, \vec{u}, i+1, \ell \cdot \ell[k])$, as needed.

(Build) we have $f[i] = \texttt{build}\ \mathsf{c}\ k$, where $\mathsf{c} : t_1, \ldots, t_k \to t_0$, and $\ell = \ell_0 \cdot v_1 \cdots v_k$ and $M' = M_0 \cdot (f, i+1, \ell_0 \cdot \mathsf{c}(v_1, \ldots, v_k))$. Since M is well-shaped we have $\vec{E}_i = E \cdot e_1 \cdots e_k$ where $e_1, \ldots, e_k$ are expressions associated with the values $v_1, \ldots, v_k$ and $\vec{E}_{i+1} = E \cdot \mathsf{c}(e_1, \ldots, e_k)$. The proposition follows by proving that $ter(f, \vec{u}, i+1, \ell_0 \cdot \mathsf{c}(v_1, \ldots, v_k))$. As for the previous case, the substitution ρ is also solution of the matching between the shape patterns and the values in $\vec{u}$. Hence we are left with proving that $\rho(\mathsf{c}(e_1, \ldots, e_k)) \geqslant_l \mathsf{c}(v_1, \ldots, v_k)$, which is a direct consequence of the fact that $\rho(e_i) \geqslant_l v_i$ for all $i \in 1..k$.

(Call) we have $f[i] = \texttt{call}\ g\ k$, where $g : (t_1, \ldots, t_k) \to t_0$, and ℓ is of the form $\ell_0 \cdot v_1 \cdots v_k$ and $M' = M \cdot (g, 1, v_1 \cdots v_k)$. Since M obeys the termination invariant, the proposition follows by proving $ter(g, (v_1, \ldots, v_k), 1, v_1 \cdots v_k)$ and $g(v_1, \ldots, v_k) <_l f(\vec{u})$.

The validity of the invariant for the frame g is immediate since $wsh(g, \vec{\sigma}', \vec{E}')$ for some valid shape $(\vec{\sigma}', \vec{E}')$ such that, by definition, $\vec{\sigma}'_1 = id$ and $\vec{E}'_1 = x_{1,1} \cdots x_{1,k}$. Indeed, the solution for the pattern-matching is the substitution $\rho' = [v_1/x_{1,1}, \ldots, v_k/x_{1,k}]$ and we have $\rho(x_{1,i}) = v_i$ for all $i \in 1..k$. To prove the validity of the inequality, we use the fact that termination annotations are correct (see condition (3) in Definition 6). Let $e_1, \ldots, e_k$ be the expressions associated to $v_1, \ldots, v_k$ in $\vec{E}_1$. Condition (3) applied to instruction $i+1$ leads to the following relation:

$$f(\vec{\sigma}_{i+1} x_{1,1}, \ldots, \vec{\sigma}_{i+1} x_{1,ar(f)}) >_l \vec{E}_{i+1}[\vec{h}_{i+1}] = g(e_1, \ldots, e_k) \ ,$$

and since $\vec{\sigma}_{i+1} = \vec{\sigma}_i$, we obtain:

$$\begin{aligned} f(\ell') &= f(\rho \circ \vec{\sigma}_i(x_{1,1}), \ldots, \rho \circ \vec{\sigma}_i(x_{1,ar(f)})) \\ &>_l \rho(g(e_1, \ldots, e_k)) \\ &\geqslant_l g(v_1, \ldots, v_k) \ . \end{aligned}$$

(Return) we have $f[i] = \texttt{return}$ and $M \equiv M_1 \cdot (g, i', \ell_g \cdot \ell') \cdot (f, i, \ell_f \cdot v)$ and $M' \equiv M_1 \cdot (g, i'+1, \ell_g \cdot v)$ where $\ell' = v_1 \cdots v_k$ and the instruction $g[i']$ is $\texttt{call}\ f\ k$ with $f : (t_1, \ldots, t_k) \to t_0$. Let $\vec{u}' = (v_1, \ldots, v_k) = arg(M, |M|)$. Since M is well-shaped, we know that $wsh(g, \vec{\sigma}', \vec{E}')$ and $wsh(g, \vec{u}, i', \ell_g \cdot \ell')$ where $\vec{u} = arg(M, m-1)$ are the parameters used to initialize the frame for g. We also know that $\vec{\sigma}'_{i'+1} = \vec{\sigma}'_{i'}$, $\vec{E}'_{i'} = E' \cdot e'_1 \cdots e'_k$ and $\vec{E}'_{i'+1} = E' \cdot f(e'_1, \ldots, e'_k)$.

The proposition follows by proving that $ter(g, \vec{u}, i'+1, \ell_g \cdot v)$. More particularly, if ρ' denotes the solution of the pattern-matching equation for the frame g, the same substitution is also solution for the resulting frame and we only need to prove that $\rho'(f(e'_1, \ldots, e'_k)) \geqslant_l v$.

Assume e is the expression $\vec{E}_i[\vec{h}_i]$, which describes the shape of the value returned by the frame f. By condition (2) on *correct termination annotations*, we know that $f(\vec{\sigma}_i(x_{1,1}), \ldots, \vec{\sigma}_i(x_{1,k})) >_l e$. Since the (decorated) frame $(f, \vec{u}', i, \ell_f \cdot v)$ is well-shaped and obeys the termination invariant, we also know that $\rho(e) \geqslant_l v$ and $\rho \circ \vec{\sigma}_i(x_{1,j}) = v_j$ for all $j \in 1..k$. Therefore we have $(c1)$ $f(v_1, \ldots, v_k) \geqslant_l v$. From the fact that the frame g obeys the termination invariant, we know that $(c2)$ $\rho'(e'_j) \geqslant_l v_j$ for all $j \in 1..k$, and therefore:

$$\begin{array}{rcll} \rho'(f(e'_1, \ldots, e'_k)) & = & f(\rho'(e'_1), \ldots, \rho'(e'_k)) & \\ & \geqslant_l & f(v_1, \ldots, v_k) & \text{by } (c2) \\ & = & f(v_1, \ldots, v_k) & \\ & >_l & v & \text{by } (c1) \end{array}$$

(Stop) we reduce to the error configuration, ϵ, which satisfies *ter* by definition.

(BranchThen) we have $f[i] = \texttt{branch}\ \texttt{c}\ j$ and $\ell = \ell_0 \cdot \texttt{c}(v_1, \ldots, v_k)$ and $M' \equiv M_0 \cdot (f, i+1, \ell_0 \cdot v_1 \cdots v_k)$ where $\texttt{c} : t_1, \ldots, t_k \to t_0$. From the proof of Proposition 6, Appendix D, we know that there exists a substitution ρ', solution of matching the patterns $(\vec{\sigma}_{i+1}(x_{1,1}), \ldots, \vec{\sigma}_{i+1}(x_{1,ar(f)}))$ with the values in $\vec{u}$, such that $\rho'(\vec{E}_{i+1}[\vec{h}_i + j - 1]) = v_j$ for all $j \in 1..ar(\texttt{c})$. Hence $\rho'(\vec{E}_{i+1}[\vec{h}_i + j - 1]) \geqslant_l v_j$ and we have $ter(f, \vec{u}, i+1, \ell \cdot v_1 \cdots v_k)$ as needed. $\square$