BEHAVIOR TYPES OF SEVERAL JVM INSTRUCTIONS

JACOPO FREDDI, CHUN TIAN, MICHAEL CANELLA, FABIO BISELLI, GIULIA BACCOLINI

Abstract. Based on the work in previous papers, we define a way to compute behavioural types for most Java bytecode instructions, in a way that they may be handled by automated verifiers to compute cost functions.

1. Conventions and mathematical constructs

1.1. Datatypes.
The datatypes we consider are:

- $\mathbb{N}^+ = \mathbb{C} \cup \mathbb{E} \cup \{-\}$ for integer values (augmented with an undefined value to manage expressions that the verifier cannot handle exactly);
- $VM = \{\alpha, \beta, \gamma, \dots\} \cup \{\texttt{this}\}$ as a set of names for virtual machines;
- $T = \{f, g, \dots\}$ as a set of thread identifiers;

$\mathbb{C}$ stands for the integer constants and is equal to $\mathbb{N}$, while $\mathbb{E}$ is the set of all expressions in which variables are involved, but can be handled by the verifier.

$\Gamma \in \underline{\Gamma}$ is a map $VM \mapsto \sigma = \{\delta, \top, \bot, a, a \downarrow\}$ which holds information about the state of each virtual machine. A general datatype $\mathbb{D} = \mathbb{N}^+ \cup VM \cup T$ is defined to compactly integrate all datatypes stored in the stack and the memory.

1.2. Machine abstractions.
We consider an abstraction of the bytecode program P with n instructions as a function defined $\forall i \in [1; n]$. No checks on the domain of P are made because the bytecode is only valid if all execution path end with a return statement, and in that case the analysis do not consider exceding lines of code.

The other abstractions we consider are:

- a memory $F \in \mathbb{F}$ seen as a map $N \mapsto \mathbb{D}$;
- a stack $S \in \mathbb{S} = \varnothing \cup (\mathbb{D} \times \mathbb{S})$;
- a constant table `Const` where `Const`(i) denotes the constant $\#i$ of the bytecode unit.

To resolve the chain of constants we consider a function $c : \mathbb{N} \mapsto$ `String` that computes the closure of references and builds the identifier for invoked and referred classes and methods.

The JVM state is abstracted with a memory F initially filled with the parameters of the method or program, and a stack frame S initially empty. On the stack S are defined the operations with the trivial semanthics, given the convention that $a \cdot b = \{a, b\}$:

$$top(S) : \mathbb{S} \mapsto \mathbb{D}.top(S) = x \text{ if } S = x \cdot S' \in \mathbb{S}$$

$$pop(S) : \mathbb{S} \mapsto \mathbb{S}.pop(S) = S' \text{ if } S = x \cdot S' \in \mathbb{S}$$

$$push(x, S) : \mathbb{D} \times \mathbb{S} \mapsto \mathbb{S}.push(x, S) = x \cdot S$$

being *top* and *pop* both intentionally partial functions, i.e. not being defined on empty stacks.

2. The Greatest Lower Bound Function

Every symbol used in the stack, memory or environment is involved in a partial ordering relationship with the other symbols of its type. The Greatest Lower Bound of two symbols, when defined, computes the greatest of the symbols lower or equal to them. The function is *not* commutative: when the arguments are different but hold the same informative value, it usually gives priority to the first one. However, the ordering will only affect the value assigned to formal parameters and is not semanthic.

2.1. $\sqcup$ on datatypes.

In general we consider a "lowest type" $\bot$ such that:

$$\forall x \in \mathbb{D} : x \sqcup \bot = \bot \sqcup x = \bot$$

This type marks the "undefined" position and is useful to define the greatest lower bound for the memory, since the domain for it must be equal in both operands. If a greatest lower bound must be computed between F and F' where $\exists i.F(i) \in \mathbb{D} \wedge \nexists F'(i)$ we can say that $\exists F'(i) = \bot \wedge F(i) \sqcup F'(i) = \bot$.

On integers:

$$\sqcup : \mathbb{N}^+ \times \mathbb{N}^+ \mapsto \mathbb{N}^+ : a \sqcup b = \begin{cases} - & a = - \vee b = - \\ b & a \in \mathbb{C} \wedge b \in \mathbb{E} \\ a & else \end{cases}$$

On virtual machines:

$$\sqcup : VM \times VM \mapsto VM : \alpha \sqcup \beta = \alpha$$

On virtual machine states:

$$\sqcup : \sigma \times \sigma \mapsto \sigma : \alpha \sqcup \beta = \begin{cases} \alpha & \alpha = \beta \\ \delta & (\alpha = \top \wedge \beta = \bot) \vee (\alpha = \bot \wedge \beta = \top) \vee (\alpha = \delta \vee \beta = \delta) \\ a \downarrow & (\alpha = a \wedge \beta = \bot) \vee (\alpha = \bot \wedge \beta = a) \vee (\alpha = a \downarrow \vee \beta = a \downarrow) \end{cases}$$

2.2. $\sqcup$ on the memory.

$$\sqcup : \mathbb{F} \times \mathbb{F} \mapsto \mathbb{F}.F_1 \sqcup F_2 = \overline{F}.\forall i \in D(\overline{F}) : \overline{F}(i) = F_1(i) \sqcup F_2(i)$$

2.3. $\sqcup$ on the stack.

$$\sqcup : \mathbb{S} \times \mathbb{S} \mapsto \mathbb{S}.S_1 \sqcup S_2 = \begin{cases} \varnothing & S_1 = S_2 = \varnothing \\ push(top(S_1) \sqcup top(S_2), pop(S_1) \sqcup pop(S_2)) & else \end{cases}$$

Notice that, being *top* and *pop* both partial functions, the function defined on the stack is not total: if there are some S_1, S_2 such that $\nexists S_1 \sqcup S_2$, the stack lengths are different and the program cannot be analysed due to stack overflow errors.

2.4. $\sqcup$ **on the environment.** We define a function $D : \underline{\Gamma} \mapsto \wp(VM)$:

$$D(\Gamma) = d.\forall \alpha \in VM : \Gamma(\alpha) \in \sigma \implies \alpha \in d$$

that computes the domain of the environment.

$$\sqcup : \underline{\Gamma} \times \underline{\Gamma} \mapsto \underline{\Gamma}.\Gamma_1 \sqcup \Gamma_2 = \overline{\Gamma}.D(\overline{\Gamma}) = D(\Gamma_1) \wedge \forall \alpha \in D(\overline{\Gamma}) : \overline{\Gamma}(\alpha) = \Gamma_1(\alpha) \sqcup \Gamma_2(\alpha)$$

2.5. **Definition of informative ordering.** We define a partial ordering function in $\mathbb{N}^+$:

$$<_\Sigma \;\subset \mathbb{N}^+ \times \mathbb{N}^+ : a <_\Sigma b \iff \begin{cases} a = - \wedge b \neq - \\ a \in \mathbb{E} \wedge b \in \mathbb{C} \end{cases}$$

$$=_\Sigma \;\subset \mathbb{N}^+ \times \mathbb{N}^+ : \mathbb{C} \times \mathbb{C} \cup \mathbb{E} \times \mathbb{E} \cup \{(-,-)\}$$

Starting from this function, we define a partial ordering between stacks and between memories:

$$<_\Sigma \;\subset \mathbb{F} \times \mathbb{F} : F_1 <_\Sigma F_2 \iff (F_1 \leq_\Sigma F_2 \wedge \exists i \in D(F_1) : F_1(i) <_\Sigma F_2(i))$$

$$<_\Sigma \subset \mathbb{S} \times \mathbb{S} : S_1 <_\Sigma S_2 \iff (top(S_1) <_\Sigma top(S_2) \vee ((top(S_1) =_\Sigma top(S_2) \wedge pop(S_1) <_\Sigma pop(S_2))$$

The relevance of making stacks and memories comparable with respect to the information value will become clear in next section.

3. Behavioural type function

The behavioural type of a section of P is given by the function $\mathbb{B}_x : \mathbb{F} \times \mathbb{S} \mapsto \mathbb{B}$ where $\mathbb{B}_i(F,S)$ types the program from the instruction i fo the end of the program considering a memory F and a stack S. This way jumps are easily typed and branches are created in a natural way.

We consider that, $\forall i : \exists P[i] \implies \Gamma_i, F_i, S_i \vdash \mathbb{B}_i(F_i, S_i) = b_i(F_i, S_i), R_i \rhd \Gamma'$, being:

- b_i the behavioural type from instruction i to the end of the program, parametric with respect to the stack and the memory (usually written recursively);
- R_i the set of virtual machines released by the instruction (usually an empty set or a singleton);
- Γ' the new environment which will type the next $\mathbb{B}_x$.

Save in particular cases, the main changes happen on the stack and the memory: therefore, if not otherwise stated, we will omit Γ' and R_i assuming that:

- $R_i = \varnothing$;
- $\Gamma' = \Gamma$.

The analysis of the program is performed in iterations of sequential scans, in which each $\mathbb{B}_i$ is computed. Ideally, when a scan discovers that $\mathbb{B}_i$ is defined on parameters F_i, S_i and invoked recursively with some F_j, S_j such that $F_j <_\Sigma F_i \vee S_j <_\Sigma S_i$, the domain of $\mathbb{B}_i$ is updated from F_i, S_i to $F_i \sqcup F_j, S_i \sqcup S_j$, all the types are recomputed to mirror the update and another scan is issued to check that all invocations parameters are at least as informative as the definition. This analysis ends because:

- the number of instruction of a program or method is finite;
- the chains defined by the $<_\Sigma$ relationship are all finite;

- the stack and the memory are limited by definition.

The analysis will then create, at the end of the last iteration, a set of $\mathbb{B}_i$ defined for all instructions, that will be analysed by the theorem prover and converted into cost functions.

4. Multithreading

Multithread computation is partially handled by the theorem prover. Since the whole structure is based on recursion, the cost function for methods and threads will be recursive too. The behaviour types to express method calls are:

- νf, to express the call of a method or the start of a thread;
- $f^{\checkmark}$, to express the synchronization of an asynchronous method or thread. Synchronous methods are synchronized right after being called.

The theorem prover will compute the right cost functions at analysis time.

We consider a simpler use case in which threads and methods cannot return virtual machines to the main thread and can only deallocate the machines that it receives as parameter. As a result, we do not need to check the synchronized sections.

The method itself is typed as b, R where:

- b is the behavioral type of the method, computed as if the method was a program. Since the method body itself starts from instruction 1 like the program, to tell apart the behavior function computing the method's behavioral type from the one computing the program's behavioral type the former one will be called $\mathbb{B}^M$, having otherwise the same semanthics.
- R is the set of released global virtual machines. The local virtual machines are not considered of interest, as their references are forever lost once the method returns.

4.0.1. *Open problems.* To be able to release virtual machines, the thread must save their references somewhere before the method `run()` is called. This may be done either in its constructor or in a custom method. This behavior leads to 2 main problems:

(1) there is no restriction on where the virtual machines are passed to the thread. Consequently, a thread must be managed in a complex way to monitor its fields;
(2) there is no way to understand whether a parameter or field is a VM or an integer, due to lack of information about types in the bytecode. The only way to know that a field is a VM is to see that it is created via a creation or that a release is invoked on it.

These problems bring to a major issue: it is extremely difficult to ensure that a virtual machine from the main thread is assigned to a field of the called thread: the process that brings to the set of released VMs is still unknown.

4.0.2. *Possible solution.* During the typing of the thread, we follow these assumptions:

- the parameters of the constructor are assigned to the fields in the order they are specified. If the stack has the following form: $\alpha \cdot \beta \cdot n \cdot S$, then the thread is constructed as $T(\alpha, \beta, n)$ and its fields with constant pool $\#i, \#i+1, \#i+2$ are actually α, β, n;
- no other method which modifies the fields is invoked before `run`;
- new VM names follow a fixed, shared convention (eg. vm1, vm2, vm3, etc) and are not random;
- when dealing with fields in the `run` method, we initially type each field as non defined until a `release` or `store` VM on them reveals their type as VM fields, updating $\mathbb{B}_1^T$ and all the remaining $\mathbb{B}$ accordingly.

In these hypothesis, we can type safely the thread in the following manner:

(1) the first virtual machines identified in the `run` method are named as the first VM created (say, vm1 and vm2 with respect to the previous example);
(2) if the fields are overwritten, the vm names will obviously change;
(3) no changes are made on the released VM set (all the released VMs are put inside R);
(4) when a `join` occurs, the set R is checked: assuming that the thread released vm1 and overwrote vm2 with vm5 releasing it afterwards, we will have $R = \{vm1, vm5\}$ (as if $vm1 = \alpha, vm2 = \beta$ etc, and $R = \{\alpha, \epsilon\}$) while in the environment the thread will have value $f = (-, run(vm143, vm352, 13 - x), \{vm1, vm5\})$, where vm143 and vm352 are some VMs created in the main thread and vm1, vm5 the machines released in the thread;
(5) since vm1 refers to the first vm of the thread, vm143 will be released in the environment;
(6) since vm5 refers to the 5th vm of the thread but the constructor only contains 2, it is counted as a local vm (it actually is) and its presence in R means nothing with respect to the environment.

5. Typing rules

(T-Program)

$$\frac{}{\Gamma, F, \varnothing \vdash P : \mathbb{B}_1(F, \varnothing)}$$

(T-Method)

$$\frac{}{\Gamma, F, \varnothing \vdash M : \mathbb{B}_1^M(F, \varnothing)}$$

(T-Return)

$$\frac{P[i] = \texttt{return}}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = 0}$$

(T-If)

$$\frac{P[i] = \texttt{if}\ [cond]\ \mathrm{L}, \mathrm{top}(S_i) \neq -}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = [cond](\mathbb{B}_L(F_i, \mathrm{pop}(S_i))) + [\neg cond](\mathbb{B}_{i+1}(F_i, \mathrm{pop}(S_i)))}$$

(T-If-Undefined)

$$\frac{P[i] = \texttt{if}\ [cond]\ \mathrm{L}, top(S_i) = -}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_L(F_i, \mathrm{pop}(S_i)) + \mathbb{B}_{i+1}(F_i, \mathrm{pop}(S_i))}$$

(T-Goto)

$$\frac{P[i] = \texttt{goto}\ L}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_L(F_i, S_i)}$$

(T-New-VM)

$$\frac{P[i] = \texttt{invokevirtual createVM}\ , \beta \text{ fresh}}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \nu\beta \fatsemi \mathbb{B}_{i+1}(F_i, \mathrm{push}(\beta, \mathrm{pop}(S_i))) \rhd \Gamma[\beta \mapsto \top]}$$

(T-Release-VM)

$$\frac{P[i] = \texttt{invokevirtual releaseVM}\ , \beta = \mathrm{top}(S_i), \Gamma(\beta) \neq \bot}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \beta^{\checkmark} \fatsemi \mathbb{B}_{i+1}(F_i, \mathrm{pop}(\mathrm{pop}(S_i))) \rhd \Gamma[\beta \mapsto \bot], \{\beta\}}$$

(T-Release-VM-Null)

$$\frac{P[i] = \texttt{invokevirtual releaseVM}\ , \beta = \mathrm{top}(S_i), \Gamma(\beta) = \bot}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, \mathrm{pop}(\mathrm{pop}(S_i)))}$$

(T-Load)

$$\frac{P[i] = \texttt{load}\ n}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, \mathrm{push}(F(n), S_i))}$$

(T-Store)

$$\frac{P[i] = \texttt{store}\ n}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i[n \mapsto \mathrm{top}(S_i)], \mathrm{pop}(S_i))}$$

(T-Integer-Increment-Defined)

$$\frac{P[i] = \texttt{iinc}\ idx\ n, F_i(idx) \neq -}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i[idx \mapsto (F_i(idx) + n)], S_i)}$$

(T-Integer-Increment-Undefined)

$$\frac{P[i] = \texttt{iinc}\ idx\ n, F_i(idx) = -}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, S_i)}$$

(T-Iconst)

$$\frac{P[i] = \texttt{iconst\_}n}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, \mathrm{push}(n, S_i))}$$

(T-LDC)

$$\frac{P[i] = \texttt{ldc}\ x, \texttt{Const}(x) \in \mathbb{C}}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, \mathrm{push}(n, S_i))}$$

(T-LDC-Undefined)

$$\frac{P[i] = \texttt{ldc}\ x, \texttt{Const}(x) = -}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, \mathrm{push}(-, S_i))}$$

(T-Monitorenter)

$$\frac{P[i] = \texttt{monitorenter}}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, S_i)}$$

(T-Monitorexit)

$$\frac{P[i] = \texttt{monitorexit}}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, S_i)}$$

(T-Thread-New)

$$\frac{\begin{array}{c}P[i] = \texttt{invokespecial}\ x, \texttt{Const}(x) = \texttt{Thread.init},\\ f \text{ fresh},\ \ \texttt{nargs}(\texttt{Thread.init}) = n, S_{i+1} = \mathrm{pop}^n(S_i)\end{array}}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \mathbb{B}_{i+1}(F_i, \mathrm{push}(f, S_{i+1})) \rhd \Gamma[f = (-, \mathrm{run}(\mathrm{top}^0(S_i), \ldots \mathrm{top}^{n-1}(S_i)), R)]}$$

(T-Thread-Run)

$$\frac{P[i] = \texttt{invokevirtual}\ x, \texttt{Const}(x) = \texttt{Thread.run},\ \ \mathrm{top}(S_i) = f \in T}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = \nu f \mathbin{;} \mathbb{B}_{i+1}(F_i, S_i)}$$

(T-Thread-Join)

$$\frac{P[i] = \texttt{invokevirtual}\ x, \texttt{Const}(x) = \texttt{Thread.join}, f = \mathrm{top}(S_i)}{\Gamma \vdash \mathbb{B}_i(F_i, S_i) = f^{\checkmark} \mathbin{;} \mathbb{B}_{i+1}(F_i, S_i)}$$

`More on the way...`