Frame-VM Specification

Chiel Bruin last updated: May 19, 2019

This document contains an overview of all the instructions implemented by the frame-vm and the VM itself. As the VM has two possible modes, these are discussed separately after the general machine overview.

Frame VM

As the VM is built on the concept of scopes as frames, its memory layout is made from data frames representing scopes. These data frames contain indexed slots were data can be stored. These slots correspond to declarations in the scope graph. Like scopes in a scopegraph are linked, data frames in the VM are also linked to form a graph (which can be seen as a form of a heap). Any sub-graph in this graph that is not referenced by any other part of the graph can be garbage collected, simmilar to garbage-collecting of values on the heap.

Besides an alternative for the heap, the VM also has a different view on the control stack. Normally this stack stores frames containing a return address, local variables and the program counter (PC). The VM is similar in that control frames are stored that contain a return address and the program counter. However, there are two big differences: Multiple return addresses are allowed (making the control-stack a control-graph) and local variables are stored in a linked data frame. In addition, there is some extra local memory that, depending on the mode, is a stack or a set of registers.

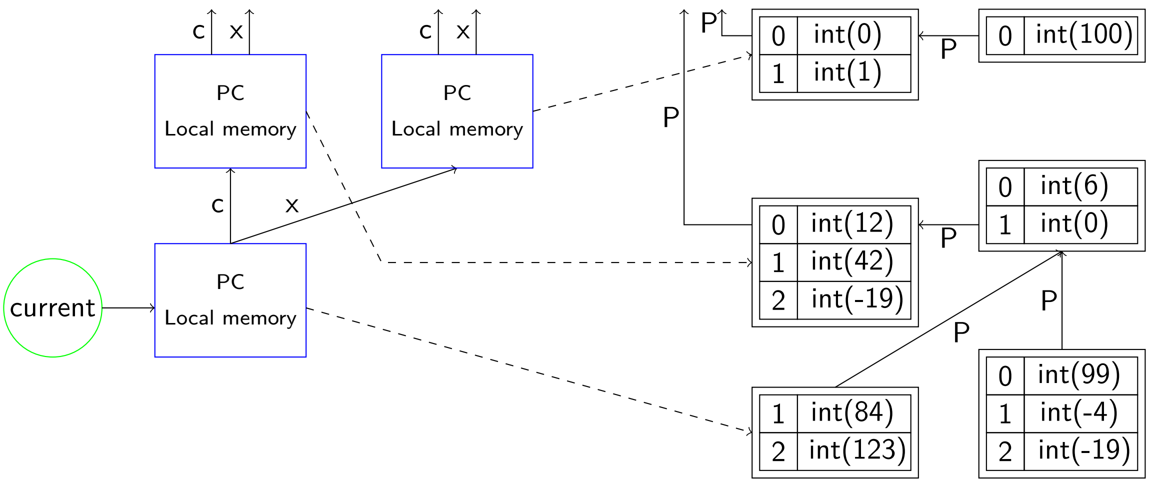

Figure 0.1: Frame VM machine layout. Control frames are displayed in blue, data frames in black. Current points to the currently executing control frame.

Having multiple return addresses for a given control frame allows you to model control-flow more easily. For example, adding exceptions is as simple as adding a second return address to the nearest exception handler. When changing to a new control frame, this extra return address needs to be copied over (to remain pointing to the nearest handler). This process of passing around the return addresses is in a sense similar to writing execution semantics in continuation passing style (CPS).

Figure 0.1 shows a more graphical example of the layout. In this figure the machine is executing a try-catch like part of a program, as a return addres (c) and nearest exception handler (x) are used.

Data Types

Possible data types that can exist in the VM, or are used in this document are:

- string: A string (only used as sugar)
- val: A generic value, can be any of the datatypes listed below
- int: An integer value
- bool[1]: A boolean value
- char[2]: A character
- frame: A reference to a data frame
- cont: A reference to a control frame, represents an execution point (continuation)
- clos: A reference to a data frame and code block, represents a closure

[1] Type alias for int
[2] Type alias for int

Stacy

The first mode of operation of the frameVM is stack-based. The bytecode language used in this mode is called Stacy (**stac**k) and has the extension `.stc`.

For each instruction its effects on the stack are listed, together with a textual description and required arguments. After this, sugared instructions and their desugarings are listed. Understanding these reductions could provide usefull insights in the workings of the VM, but is not neccesary (assuming your language only uses function returns and exception handlers).

As the frame VM uses indexed links and slots internally, you need to define a mapping between names and indices of edge labels and continuation slots. Stacy already predefines a number of these mappings for free (namely `P` $\rightarrow$ `0`, `I` $\rightarrow$ `1`, `c` $\rightarrow$ `0`, `x` $\rightarrow$ `1` and `n` $\rightarrow$ `2`). Adding additional labels should be done with caution.

Block syntax

In Stacy all instructions are grouped in code blocks. These blocks start with label with their unique name, followed by an indented list of instructions. This list of instructions must be followed by a control-influencing instruction to complete a block (these cannot be used inside a block). The instructions that are in this group are listed in figure 0.2

Instruction	Arguments	Instruction	Arguments
exitscope	[path] label	**tailcall**	label
newscope	label1 label2	**tailcall**	
jumpz	label1 label2	**return**	
jump	label	**return**	int
call	label1 label2	**ccall**	label
call	label	**cret**	
yield	label		

Figure 0.2: All the control-influencing instructions of Stacy. Note that some instructions in this list have a similar instruction that does not influence control.

Instructions

Please note that copy operations (specifically those using a policy), will change in the future. If there is the need to make use of these operations, please contact the developer, so the implementation priority can be bumped.

Table 0.1: Arithmetic operations implemented by the virtual machine.

Instruction	Arguments	Pop	Push	Δ	Description
ipush	int		int	1	*Pushes the given int on the stack*
addi		int1, int2	int	-1	*Adds the two values*
subi		int1, int2	int	-1	*Subtracts* int1 *from* int2
muli		int1, int2	int	-1	*Multiplies the two values*
divi		int1, int2	int	-1	*Divides* int2 *by* int1
modi		int1, int2	int	-1	*Calculates* int2 *modulo* int1
eqi		int1, int2	bool	-1	*Checks if the two values are equal*
lti		int1, int2	bool	-1	*Checks if* int2 *is less than* int1
gti		int1, int2	bool	-1	*Checks if* int2 *is greater than* int1
ori		int1, int2	bool	-1	*Calculates the binary or*
xori		int1, int2	bool	-1	*Calculates the binary xor*
andi		int1, int2	bool	-1	*Calculates the binary and*

Table 0.2: Debug operations implemented by the virtual machine.

Instruction	Arguments	Pop	Push	Δ	Description
print		val		-1	*Prints* val *to the console*
debug				0	*Generates a DOT representation of the machine state*
debug!				0	*Generates a DOT representation of the machine state and kill execution*

Table 0.3: Closure operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
newc	policy, lbl	frame	clos	0	*Creates a closure of* frame *with* lbl *as label*
newc*	lbl	frame	clos	0	*Creates a closure of* frame *with* lbl *as label*
cnew	int	clos	cont	0	*Creates* cont *from* clos *with* int *continuation slots*
unpack		clos	frame	0	*Unpacks* frame from closure

Table 0.4: Type operations implemented by the virtual machine.

Instruction	Arguments	Pop	Push	Δ	Description
int?		val	bool	0	*Checks if* val *is an integer*
cont?		val	bool	0	*Checks if* val *is a continuation*
frame?		val	bool	0	*Checks if* val *is a frame*
closure?		val	bool	0	*Checks if* val *is a closure*

Table 0.5: Frame operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
new*			frame	1	*Create a new frame with size 0 and pushes a reference to it on the stack*
new*	int		frame	1	*Create a new frame with size* int *and pushes a reference to it on the stack*
newr		int	frame	1	*Create a new frame with size* int *and pushes a reference to it on the stack*
link*	[path] label	frame		-1	*Link the frame on top of the stack to the given location using* label *as label*
linkr	label	frame1, frame2		-2	*Link* frame2 *to* frame1 *using* label *as label*
copy*			frame	1	*Makes a shallow copy of the current frame*
copy*	policy1, policy2		cont	1	*Makes a copy of the current execution context using* policy1 *for the control frames and* policy2 *for the data frames*
copyr*		frame1	frame2	0	*Makes a shallow copy of* frame
copyr	policy1, policy2	cont1	cont2	0	*Makes a copy of* cont1 *using* policy1 *for the control frames and* policy2 *for the data frames*
size		frame	int	0	*Gets the number of slots of* frame
set*		val, int		-2	*Store* val *in slot* int *of the current frame*
set*	[path]	val		-1	*Store* val *at the given location*
setr		val, int, frame		-3	*Store* val *in slot* int *of* frame
setr*	[path]	val, frame		-2	*Store* val *at the given location, starting* path *at* frame
get*		int	val	0	*Get the value in slot* int *of the current frame*
get*	[path]		val	1	*Get the value at the given location and store it on the stack*
getr		int, frame	val	-1	*Get the value in slot* int *of* frame
getr*	[path]	frame	val	0	*Get the value at the given location, starting from* frame *and store it on the stack*

Table 0.6: Scoping/dataframe operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
exitscope*	[path]			0	*Change the current dataframe to the frame at* path. *Breaks from nested scopes to the nesting scope*
exitscope*	[path] label			0	*Change the current dataframe to the frame at* path. *Breaks from nested scopes to the nesting scope. Jump execution to* label
newscope*	link	frame		-1	*Enters a nested scope by setting the current dataframe to* frame . *This new frame will be linked using* link *to the original frame*
newscope*	link label	frame		-1	*Enters a nested scope by setting the current dataframe to* frame . *This new frame will be linked using* link *to the original frame. Jumps execution to* label
mkcurrent		frame		-1	*Make* frame *the current dataframe*
getcurrent			frame	1	*Get the current dataframe*

Table 0.7: Control operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
jumpz	label1 label2	bool		-1	*Jump to* label1 *if* bool *is false, otherwise jump to* label2
jump	label			0	*Unconditional jump to* label
call*	label1 label2	frame		0	*Calls a function at location* label1 *using* frame *as execution frame. When the function returns, execution is resumed at* label2
call*	label	cont		0	*Calls* cont . *When the function returns, execution is resumed at* label
tailcall*	label	frame		0	*Calls a function at location* label *using* frame *as execution frame. Uses tail-call optimizations*
tailcall*		cont		0	*Calls* cont . *Uses tail-call optimizations*
return*		val		-1	*Return* val
return*	int	val{int}		$-int$	*Return the* int *values on top of the stack*
yield*	label	val		-1	*Yield* val *and the current continuation. Jumps execution to* label
rget*			val	1	*Get the retruned value after a function call returns*
rget	int		val{int}	int	*Get* int *returned values after a function call returns*

Table 0.8: Stack manipulation operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
pop		val		-1	*Discards the element on top of the stack*
dup*		val	val, val	1	*Duplicate the element on top of the stack*
dup	int	val{int-1} val2	val2, val{int-1}, val2	1	*Duplicate the element on the* int*-th positionof the stack*
swap*		val1, val2	val1, val2	0	*Swap the two top elements of the stack*
swap	int	val1, val{int-2} val2	val2, val{int-2}, val1	0	*Swaps the element on top of the stack, withe one on the (*int*-1)-th position of the stack*

Table 0.9: Character handling operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
spush*	string		frame	1	*Convert* string *to a character array and push it on the stack*
cpush	char		char	1	*Push* char *to the stack*
printc		char		-1	*Print* char *to the console*

Table 0.10: Continuation operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
cgetcurrent			cont	1	*Create a continuation of the current execution point*
cnew*	label int	frame	cont	0	*Create a continuation of a new control frame with data frame* frame *, execution point* label *and size* int
cnewr	label	int, frame	cont	-1	*Create a continuation of a new control frame with data frame* frame *, execution point* label *and size* int
ccall	label	cont		-1	*Call* cont *and set the current execution point to* label
cret		cont		-1	*Call* cont*. Do not set a next execution point*
transfer	int	cont		$-(\text{int}+1)$	*Transfer* int *elements as returned values to* cont
transfer*	int [path]			$-(\text{int})$	*Transfer* int *elements as returned values to the given continuation*
cset*		cont, int		-2	*Store* cont *in slot* int *of the current controlframe*
cset*	[path]	cont		-1	*Store* cont *at the given location*
csetr*	[path]	cont1, cont2		-2	*Store* cont1 *in the given slot of* cont2
csetr		cont1, int, cont2		-3	*Store* cont1 *in slot* int *of* cont2
cget*		int	cont	0	*Get the continuation in slot* int *of the current frame*
cget*	[path]		cont	1	*Get the continuation at the given location*
cgetr*	[path]	cont1	cont2	0	*Get the continuation at the given location of* cont1
cgetr		int, cont1	cont2	-1	*Get the continuation in slot* int *of* cont1

Table 0.11: Exception handling operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
throw*		val		-1	*Throw the element on top of the stack to the current exception handler*
try*	label1 label2 label3	frame1, frame2		-2	*Creates a try-catch block with* frame2 *as try-block running* label1 *and* frame1 *as catch-block running* label2. *The next instruction is at* label3
try*	label	cont1, cont2		-2	*Creates a try-catch block with* cont2 *as try-block and* cont1 *as catch-block. The next instruction is at* label

Equivalent Operations

A lot of instructions listed in the previous section are so-called derived instructions. These instructions are a form of syntactic sugar, and therefore do not add any functionality to the VM. In this section the equivalent operations are shown for the derived instructions. Please note that these reductions may not result in a fully reduced set of instructions. It might be needed to recursively apply the rules to find this final form.

dup	⇒	**dup** 1	**return**	⇒	**return** 1
swap	⇒	**swap** 1	**cpush** char	⇒	**ipush** char
new	⇒	**new** 0	**get** []	⇒	**getcurrent**
transfer	⇒	**transfer** 1	**rget**	⇒	**rget** 1

Figure 0.3: Simple equivalent operations

new n	⇒	**ipush** n **newr**	**set**	⇒	**getcurrent** **swap** 2 **swap** **setr**
link path lbl	⇒	**get** path **linkr** lbl			
get	⇒	**getcurrent** **swap** **getr**	**set** [slot]	⇒	**getcurrent** **swap** **setr** path[slot]
get path	⇒	**getcurrent** **getr** path	**set** path	⇒	**get** path[:-1] **swap** **setr** path[-1:]
getr path	⇒	**getr** [i], $\forall i \in$ path	**setr** [slot]	⇒	**ipush** slot **swap** **setr**
getr [slot]	⇒	**ipush** slot **getr**	**setr** path	⇒	**getr** [i], $\forall i \in$ path[:-1] **setr** path[-1:]

Figure 0.4: Equivalent operations for frame-get, frame-set and linking

cnew lbl int	$\Rightarrow$	**ipush** int **cnewr** lbl
cget [c]	$\Rightarrow$	**cgetcurrent** **cgetr** [c]
cget	$\Rightarrow$	**cgetcurrent** **swap** **cgetr**
cgetr [c]	$\Rightarrow$	**ipush** n, $(c \to n) \in \mathbb{A}$ **getr**
cset [c]	$\Rightarrow$	**cgetcurrent** **swap** **csetr** [c]
cset	$\Rightarrow$	**cgetcurrent** **swap** 2 **swap** **csetr**
csetr [c]	$\Rightarrow$	**ipush** n, $(c \to n) \in \mathbb{A}$ **swap** **csetr**

Figure 0.5: Equivalent operations for continuation instructions. $\mathbb{A}$ denotes the set of continuation-aliasses defined in the programs header

exitscope path	$\Rightarrow$	**get** path **mkcurrent**
newscope link	$\Rightarrow$	**dup** **link** [] link **mkcurrent**
exitscope path lbl	$\Rightarrow$	**exitscope** path **jump** lbl
newscope link lbl	$\Rightarrow$	**newscope** link **jump** lbl

Figure 0.6: Equivalent operations for control instructions (cont.)

call lbl	$\Rightarrow$	**dup** **cget** [**x**] **csetr** [**x**] **dup** **cget** [] **csetr** [**c**] **ccall** lbl
tailcall	$\Rightarrow$	**dup** **cget** [**x**] **csetr** [**x**] **dup** **cget** [**c**] **csetr** [**c**] **cret**
call lbl1 lbl2	$\Rightarrow$	**cnew** lbl1 **2** **call** lbl2
tailcall lbl	$\Rightarrow$	**cnew** lbl **2** **tailcall**
return n	$\Rightarrow$	**transfer** n [**c**] **cget** [**c**] **cret**
yield lbl	$\Rightarrow$	**cget** [] **swap** **transfer 2** [**c**] **cget** [**c**] **ccall** lbl
throw	$\Rightarrow$	**transfer 1** [**x**] **cget** [**x**] **cret**

Figure 0.7: Equivalent operations for control instructions

try lbl1 lbl2 lbl3	⇒	**cnew** lbl2 **3**	**try** lbl	⇒	**dup 2**
		dup			**swap**
		cget []			**dup**
		csetr [**n**]			**cget** []
		dup			**csetr** [**n**]
		cget [**x**]			**dup**
		csetr [**x**]			**cget** [**x**]
		dup			**csetr** [**x**]
		cget [**c**]			**dup**
		csetr [**c**]			**cget** [**c**]
		swap			**csetr** [**c**]
		cnew lbl1 **3**			**csetr** [**x**]
		dup			**dup**
		cget []			**cget** []
		csetr [**n**]			**csetr** [**n**]
		dup			**dup**
		swap 2			**cget** [**c**]
		csetr [**x**]			**csetr** [**c**]
		dup			**ccall** lbl
		cget [**c**]			
		csetr [**c**]			
		dup			
		cget []			
		csetr [**n**]			
		ccall lbl3			

Figure 0.8: Equivalent operations for control instructions (cont.)

Strings

The VM does not force a certain representation of arrays (as they can be cons-lists, NULL-terminated or keep track of their sizes). As an effect there is also no clear way to define how strings should be modeled. However, Stacy does provide some help when working with strings, albeit only for one of the representations and only for constructing strings. The `spush`-instruction creates a frame on the stack that contains the length of the string in slot 0, and the individual characters in consecutive slots.

This instruction can therefore be desugared in the following way:

spush string	$\Rightarrow$	**new** *length(*string*) + 1*
		dup
		ipush *length(*string*)*
		setr [0]
		For all characters c *at position* n *in* string *:*
		dup
		cpush c
		setr [n *+ 1***]**

Providing functionality for printing entire strings cannot be done in a simmilar way. This is because this functionality loops over the array and print the individual characters. Therefore the desugared version would result in multiple new code blocks that must be reused between multiple uses of the original instruction. This makes that it is more like a library function that must be included once. Exactly for this reason the VM supports imports of external functions. Furthermore there is a standard library containing functions for, for example, string concatenation and integer to string conversions.

Helper functions

In order to aid code generation for Stacy, a number of Stratego helper strategies are provided.

- `stc-from-flat`: Given a list of Stacy instructions, generate a valid Stacy AST.
  If you want to set the initial frame size, use link aliasses or imports, the list of instructions should be the second alement in a tuple, where the first is a FVM_Header constructor. If a label is found inside this list, a new block is started. This allows you to generate the code without explicitly creating code blocks (the `MAIN` label is placed before the first instruction in the list).
- `framevm-path-from-nabl2`: Given a three-tuple `(name, namespace, property)` gives a Frame VM path which resolves to the declaration of `<namespace>{name}`. `property` refers to the property of the declaration where a slot index is stored.

Roger

The second mode of operation of the frameVM is register-based. The bytecode-language used in this mode is called Roger (**re**g**i**st**er**) and has the extension `.rgr`.

This language is currently still in its Alpha-phase (note the capital A), and therefore not ready for use. When the language reaches any level of (feature-)stability, this document will be updated. In short, Roger will have the same instructions as Stacy but without stack operations and the possibility to make expressions and use (control frame-local) variables.