Frame-VM ISA Specification

Chiel Bruin

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

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

- val: A generic value, can be any of the datatypes listed below

- int: An integer value

- bool[1]: A boolean value

- frame: A reference to a data frame

- cont: A reference to a control frame, represents an execution point (continuation)

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 dane with caution.

Instructions

---

[1]Currently implemented as an integer, do not rely on this

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: Miscellanious 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: 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*

Table 0.4: Frame operations implemented by the virtual machine

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 0.5: Scoping/dataframe operations implemented by the virtual machine

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	label	frame		−1	*Enters a nested scope by setting the current dataframe to* frame *. This new frame will be linked using* label *to the original frame*
newscope	label1 label2	frame		−1	*Enters a nested scope by setting the current dataframe to* frame *. This new frame will be linked using* label1 *to the original frame. Jumps execution to* label2
mkcurrent		frame		−1	*Make* frame *the current dataframe*

Table 0.6: Control operations implemented by the virtual machine

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 *retruned values after a function call returns*

Table 0.7: Stack manipulation operations implemented by the virtual machine

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 0.8: Continuation operations implemented by the virtual machine

Instruction	Arguments	Pop	Push	Δ	Description
cget	[]		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*
ccall	label	cont		−1	*Call cont and set the current execution point to label*
transfer	int	cont		−(int+1)	*Transfer int elements as returned values to cont*
transfer	int [path]			−(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		cont1, int, cont2		−3	*Store cont1 in slot int of cont2*
csetr	[path]	cont1, cont2		−2	*Store cont1 in the given slot 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		int, cont1	cont2	−1	*Get the continuation in slot int of cont1*
cgetr	[path]	cont1	cont2	0	*Get the continuation at the given location of cont1*

Table 0.9: Exception handling operations implemented by the virtual machine

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

link path lbl	⇒	**get** path
		linkr lbl

cnew lbl	⇒	**get** []
		cnewr lbl

get	⇒	**get** []
		swap
		getr

get path	⇒	**get** []
		getr path

getr path	⇒	**getr** $[i]$, $\forall i \in$ path

getr [slot]	⇒	**ipush** slot
		getr

swap	⇒	**swap** 1

new	⇒	**new** 0

set	⇒	**get** []
		swap 2
		swap
		setr

set path	⇒	**get** path[:-1]
		swap
		setr path[-1:]

setr path	⇒	**getr** $[i]$, $\forall i \in$ path[:-1]
		setr path[-1:]

setr [slot]	⇒	**ipush** slot
		swap
		setr

dup	⇒	**dup** 1

return	⇒	**return** 1

Figure 0.1: Equivalent operations for frame-get, frame-set and linking

exitscope path	⇒	**get** path
		mkcurrent

newscope link	⇒	**dup**
		link [] link
		mkcurrent

exitscope path lbl	⇒	**exitscope** path
		jump lbl

newscope link lbl	⇒	**newscope** link
		jump lbl

Figure 0.2: Equivalent operations for control instructions (cont.)

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

Figure 0.3: 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 0.4: Equivalent operations for control instructions (cont.)

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, the first element of this list should be a string containing the size. 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 (**reg**ist**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.