

FRAME-VM SPECIFICATION

Chiel Bruin

last updated: June 20, 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 where 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, similar 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.

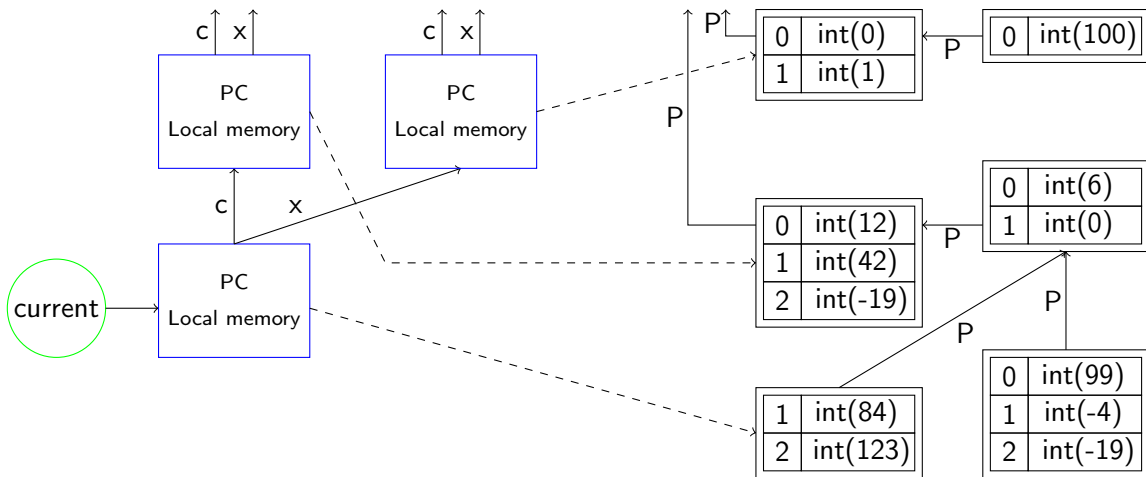


Figure 0.1: Frame VM machine layout. Control frames are displayed in blue, data frames in black. Current points to the currently executing control frame.

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 address (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
- bool¹: A boolean value
- char²: 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

Header

Any program running on the Frame VM starts with an optional header element. This header contains information about the initial state of the VM and imports that are used. Possible header elements are:

- #init n: Use an initial dataframe of size n. A frame with size 0 is used when not specified.
- #start lbl: The block where executions starts. MAIN is used when not specified.
- #cont c -> index: Add a mapping from a continuation label to an internal index.
- #link l -> index: Add a mapping from a link label to an internal index.
- export lbl as f: Export a block as a function that can be imported by other files.
- from lib import f+: Import functions from a library.
- from lib import f as f': Import a function from a library with a new name.

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.

¹Type alias for int

²Type alias for int

Block syntax

In both Stacy and Roger 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 and 0.3.

Both Stacy and Roger have a maximum size to their control frame-local memory. When creating a new control frame, you must therefore specify the maximum size of this memory. This can be done by specifying this size as the first instruction of the first block of the new control frame. The syntax for this is `#stack := n` and `#local := n` for Stacy and Roger respectively.

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	[c] label
call	label	ccall	label
yield	label	cret	[c]
try	label1 label2 label3	cret	
try	label	throw	

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.

Instruction	Arguments	Instruction	Arguments
return	val*	tailcall	exp
yield	val+	tailcall	exp label
throw	exp	ccall	exp label
jumpz	exp label1 label2	cret	exp
jump	label	try	exp1 label1 exp2 label2 label3
call	exp label	try	exp1 exp2 label
call	exp label1 label2		

Figure 0.3: All the control-influencing instructions of Roger.

Stacy

The first mode of operation of the frameVM is stack-based. The bytecode language used in this mode is called Stacy (**stack**) 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 useful insights in the workings of the VM, but is not necessary (assuming your language only uses function returns and exception handlers).

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	<i>Pushes the given int on the stack</i>
negi		int	int	0	<i>Inverts the sign of int</i>
addi		int1, int2	int	-1	<i>Adds the two values</i>
subi		int1, int2	int	-1	<i>Subtracts int1 from int2</i>
muli		int1, int2	int	-1	<i>Multiplies the two values</i>
divi		int1, int2	int	-1	<i>Divides int2 by int1</i>
modi		int1, int2	int	-1	<i>Calculates int2 modulo int1</i>
eqi		int1, int2	bool	-1	<i>Checks if the two values are equal</i>
lti		int1, int2	bool	-1	<i>Checks if int2 is less than int1</i>
gti		int1, int2	bool	-1	<i>Checks if int2 is greater than int1</i>
ori		int1, int2	bool	-1	<i>Calculates the binary or</i>
xori		int1, int2	bool	-1	<i>Calculates the binary xor</i>
andi		int1, int2	bool	-1	<i>Calculates the binary and</i>

Table 0.2: Debug operations implemented by the virtual machine.

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

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

Instruction	Arguments	Pop	Push	Δ	Description
newc	lbl, int	frame	clos	0	<i>Creates a closure of frame with lbl as label</i>
cnw		clos	cont	0	<i>Creates cont from clos with int continuation slots</i>

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
eqr		val1, val2	bool	-1	Checks if val1 is (reference) equal to val2

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: Empty slot operations implemented by the virtual machine. Derived instructions are marked with *.

Instruction	Arguments	Pop	Push	Δ	Description
empty*		int		−1	Empty slot int of the current frame
empty*	[path]			0	Empty the slot at the given location
emptyr		int, frame		−2	Empty slot int of frame
emptyr*	[path]	frame		−1	Empty the given slot, starting path at frame
empty?*		int	bool	0	Check if slot int of the current frame is empty
empty?*	[path]		bool	1	Check if the slot at the given location is empty
emptyr?		int, frame	bool	−1	Check if slot int of frame is empty
emptyr?*	[path]	frame	bool	0	Check if the slot at the given location, starting from frame is empty

Table 0.7: 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 0.8: 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 0.9: 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 position of the stack
swap*		val1, val2	val1, val2	0	Swap the two top elements of the stack
swap	int	val1, val2	val2, val1	0	Swaps the element on top of the stack, with the one on the (int-1)-th position of the stack

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

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

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	<i>Throw the element on top of the stack to the current exception handler</i>
try*	label1 label2 label3	frame1, frame2		-2	<i>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</i>
try*	label	cont1, cont2		-2	<i>Creates a try-catch block with cont2 as try-block and cont1 as catch-block. The next instruction is at label</i>

Table 0.12: 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
ccall*	[c] label			0	Call the given continuation and set the current execution point to label
cret		cont		-1	Call cont. Do not set a next execution point
cret*	[c]			0	Call the given continuation. Do not set a next execution point
transfer*		cont, val		-1	Transfer val as returned value to cont
transfer	int	cont, val{int}		-(int+1)	Transfer val as returned values to cont
transfer*	int [path]	val{int}		-(int)	Transfer val as returned values to the given continuation
cset*		cont, int		-2	Store cont in slot int of the current controlframe
cset*	[c]	cont		-1	Store cont at the given location
csetr*	[c]	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*	[c]		cont	1	Get the continuation at the given location
cgetr*	[c]	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
cunpack		cont	frame	0	Get the dataframe from cont

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. These reductions can automatically be applied to any Stacy program by running the *Transform -> Transform* action.

dup	\Rightarrow	dup 1	cpush char	\Rightarrow	ipush char
swap	\Rightarrow	swap 1	get []	\Rightarrow	getcurrent
new	\Rightarrow	new 0	set []	\Rightarrow	mkcurrent
transfer	\Rightarrow	transfer 1	rget	\Rightarrow	rget 1
return	\Rightarrow	return 1			

Figure 0.4: Simple equivalent operations

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

Figure 0.5: Equivalent operations for frame-get, frame-set and linking

empty?	\Rightarrow	getcurrent swap emptyr?	empty	\Rightarrow	getcurrent swap emptyr
empty? path	\Rightarrow	getcurrent emptyr? path	empty path	\Rightarrow	getcurrent emptyr path
emptyr? [slot]	\Rightarrow	ipush slot emptyr?	emptyr [slot]	\Rightarrow	ipush slot emptyr
emptyr? path	\Rightarrow	getr [-1] emptyr? [-1:]	emptyr path	\Rightarrow	getr path[:-1] emptyr path[-1:]

Figure 0.6: Equivalent operations for set-empty and is-empty

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

Figure 0.7: Equivalent operations for continuation instructions. \mathbb{A} denotes the set of continuation-aliases defined in the programs header

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

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

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

Figure 0.9: Equivalent operations for control instructions

try lbl1 lbl2 lbl3	\Rightarrow	cnew lbl2 3 dup cgetcurrent csetr [n] dup cget [x] csetr [x] dup cget [c] csetr [c] swap cnew lbl1 3 dup cgetcurrent csetr [n] dup swap 2 csetr [x] dup cget [c] csetr [c] dup cgetcurrent csetr [n] ccall lbl3	try lbl	\Rightarrow	dup 2 swap dup cgetcurrent csetr [n] dup cget [x] csetr [x] dup cget [c] csetr [c] csetr [x] dup cgetcurrent csetr [n] dup cget [c] csetr [c] ccall lbl
---------------------------	---------------	--	----------------	---------------	--

Figure 0.10: Equivalent operations for control instructions (cont.)

Roger

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

In short, Roger has the same instructions as Stacy but without stack operations and with the possibility to make expression trees and use (control frame-local) variables. This has the effect that most of the instructions present in Stacy are expressions in Roger. More specifically all Stacy instructions that pushed values to the stack are expressions in Roger. Stacy instructions that only consume values from the stack are still instructions in Roger. In addition, Roger adds an instruction that evaluates an expression and stores the result in a register. This instruction is complemented with an expression that reads values from these registers. All instructions in Roger are listed below, followed by all instructions.

Instructions

Table 0.13: Instructions implemented by the virtual machine.

Instruction	Arguments	Description
link	frame path lbl	<i>Link frame to the frame at path using label lbl</i>
link	frame1 frame2 lbl	<i>Link frame1 to frame2 using label lbl</i>
set	path val	<i>Set value at path to val</i>
set	int val	<i>Set value in slot with index int to val</i>
set	frame path val	<i>Set value at path relative to frame to val</i>
set	frame int val	<i>Set value in slot with index int of frame to val</i>
empty	path	<i>Empty the slot at path</i>
empty	int	<i>Empty the slot with index int</i>
empty	frame path	<i>Empty the slot at path relative to frame</i>
empty	frame int	<i>Empty the slot with index int of frame</i>
transfer	val+ c	<i>Transfer the values val to the controlframe at c</i>
transfer	val+ cont	<i>Transfer the values val to cont</i>
cset	c cont	<i>Set the continuation at c to cont</i>
cset	int cont	<i>Set the continuation in slot with index int to cont</i>
cset	cont1 c cont2	<i>Set the continuation at c relative to cont1 to cont2</i>
cset	cont1 int cont2	<i>Set the continuation in slot with index int of controlframe cont1 to cont2</i>
exitscope	path	<i>Set the current dataframe to the dataframe at path</i>
newscope	frame label	<i>Set the current dataframe to frame and link to the current with label</i>
mkcurrent	frame	<i>Set the current dataframe to frame</i>
print	val	<i>Print val to the console</i>
printc	int	<i>Print int as a character to the console</i>
debug		<i>Print the debug representation of the VM</i>
debug!		<i>Print the debug representation of the VM and terminate</i>

Table 0.14: Control instructions implemented by the virtual machine.

Instruction	Arguments	Description
return	val*	<i>Return the values exp</i>
yield	val	<i>Return exp and the current execution point</i>
throw	val	<i>Throw val</i>
jumpz	int label1 label2	<i>If exp evaluates to 0, jump to label1 else jump to label2</i>
jump	label	<i>Jump execution to label</i>
call	frame label1 label2	<i>Call a function at label1 with dataframe frame and return address label2</i>
call	cont label	<i>Call cont with return address label</i>
tailcall	frame label	<i>Call a function at label with dataframe frame without a return address</i>
tailcall	cont	<i>Call cont without a return address</i>
ccall	cont label	<i>Call cont with label as return address</i>
cret	cont	<i>Call cont without a return address</i>
try	frame1 label1 frame2 label2 label3	<i>Try-catch block with frame1/label1 as try block and frame2/label2 as catch block</i>
try	cont1 cont2 label	<i>Try-catch block with cont as try block and cont2 as catch block</i>

Expressions

Table 0.15: Continuation operations implemented by the virtual machine.

Instruction	Arguments	out	Description
rget		val	Get a returned value
rget n		val	Get n returned values
cgetcurrent		cont	Get the current control frame
cnew	frame, label, int	cont	Create a new control frame with dataframe frame , execution point label and size int
cnew	clos	cont	Create a new control frame from clos
cget	c	cont	Get the continuation at c
cget	int	cont	Get the continuation in slot with index int
cget	cont c	cont	Get the continuation at c relative to cont
cget	cont int	cont	Get the continuation in slot with index int of control-frame cont
cunpack	cont	frame	Get the dataframe of controlframe cont
newc	frame lbl int	clos	Create a closure with dataframe frame, int continuation slots and code point lbl

Table 0.16: Arithmetic operations implemented by the virtual machine.

Instruction	Arguments	out	Description
sload	string	frame	Evaluates to a frame representing string
cload	char	int	Evaluates to an int representing char
iload	int	int	Evaluates to int
negi	int	int	Flips the sign of int
addi	int1 int2	int	Adds int1 and int2
mul	int1 int2	int	Multiplies int1 and int2
subi	int1 int2	int	Subtracts int1 from int2
divi	int1 int2	int	Divides int1 by int2
modi	int1 int2	int	Calculates int1 modulo int2
eqi	int1 int2	bool	Checks if int1 and int2 are equal
lti	int1 int2	bool	Checks if int1 is less than int2
gti	int1 int2	bool	Checks if int1 is greater than int2
ori	int1 int2	int	Calculates int1 or int2
xori	int1 int2	int	Calculates int1 xor int2
andi	int1 int2	int	Calculates int1 and int2

Table 0.17: Type comparison expressions implemented by the virtual machine.

Instruction	Arguments	out	Description
int?	val	bool	Checks if val is an integer
frame?	val	bool	Checks if val is a frame
closure?	val	bool	Checks if val is a closure
cont?	val	bool	Checks if val is a continuation

Table 0.18: Frame operations implemented by the virtual machine.

Instruction	Arguments	out	Description
new		frame	Create a new frame with size 0
new	int	frame	Create a new frame with size int
new	{val+}	frame	Create a new frame with val in its slots
size	frame	int	Get the size of frame
get	path	val	Get the value at path
get	int	val	Get the value in slot with index int
get	frame path	val	Get the value at path relative to frame
get	frame int	val	Get the value in slot with index int of frame
empty?	path	bool	Check if slot at path is empty
empty?	int	bool	Check if slot with index int is empty
empty?	frame path	bool	Check if slot at path relative to frame is empty
empty?	frame int	bool	Check if slot with index int of frame is empty
getcurrent		frame	Get the current data frame

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 aliases or imports, the list of instructions should be the second element 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 if there is no label in the first position). Providing a maximum stack size can be done by providing a tuple with a label and an integer instead of a plain label.
- **rgr-from-flat**: Given a list of Roger instructions, generate a valid Roger AST.
If you want to set the initial frame size, use link aliases or imports, the list of instructions should be the second element 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 if there is no label in the first position). Providing a maximum number of registers can be done by providing a tuple with a label and an integer instead of a plain label.
- **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. When property is an integer, this integer will be used as the index instead.

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 and Roger do provide some help when working with strings, albeit only for one of the representations and only for constructing strings. The `spush` (in Stacy) and `sload` (in Roger) instructions create a frame on the stack that contains the length of the string in slot 0, and the individual characters in consecutive slots.

For Stacy, this instruction can therefore be desugared in the following way:

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

For Roger, this desugared form is similar. Providing functionality for printing entire strings cannot be done in a similar 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.