October 25, 2016

RegisterName contains pc, but is otherwise some undefined, finite set.

$$\begin{aligned}
\text{Addr} &::= \mathbb{N} & \text{Word} &::= \text{Cap} + \mathbb{Z} \\
\text{Reg} &::= \text{RegisterName} \to \text{Word} & \text{Heap} &::= \text{Addr} \to \text{Word} \\
\text{Perm} &::= \{\text{o}, \text{ro}, \text{rw}, \text{rx}, \text{e}, \text{rwx}\} & \text{Mem} &::= \text{Reg} \times \text{Heap} \\
\text{Cap} &::= \text{Perm} \times \text{Addr} \times \text{Addr} \times \text{Addr} & \text{Conf} &::= \text{Mem} + \{\mathit{failed}, \mathit{halted} \times \text{Mem}\} \\
\text{HeapSegment} &::= \text{Addr} \rightharpoonup \text{Word}
\end{aligned}$$

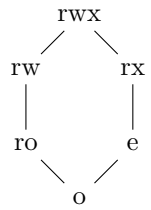

Figure 1: Permission hierarchy

Notation:

$$\begin{aligned}
i &\in \text{Instructions} \\
r &\in \text{RegisterName} \\
\mathit{mem} &::= (\mathit{reg}, \mathit{heap}) \\
\mathit{pc} &\in \text{Cap} \\
\text{pc} &\in \text{RegisterName} \\
\Phi &::= \mathit{mem} \in \text{Conf} \\
\Phi.\text{heap} &\in \text{Heap} \\
\Phi.\text{reg} &\in \text{Reg} \\
a &\in \text{Addr} \\
\mathit{perm} &\in \text{Perm} \\
(\mathit{perm}, \mathit{base}, \mathit{end}, a) &\in \text{Cap} \\
n &\in \mathbb{Z}
\end{aligned}$$

Further definitions:

$$\begin{aligned}
\mathit{lv} &::= \lfloor r \rfloor \\
\mathit{hv} &::= \langle r \rangle_h \\
\mathit{rv} &::= n \mid \mathit{lv} \\
i &::= \texttt{fail} \mid \texttt{halt} \mid \texttt{jmp}\ \mathit{lv} \mid \texttt{jnz}\ \mathit{lv}\ \mathit{rv} \mid \texttt{isptr}\ \mathit{lv}\ \mathit{rv} \mid \texttt{setptr}\ \mathit{lv}\ \mathit{rv} \mid \\
&\quad \texttt{lea}\ \mathit{lv}\ \mathit{rv} \mid \texttt{move}\ \mathit{lv}\ \mathit{rv} \mid \texttt{load}\ \mathit{lv}\ \mathit{hv} \mid \texttt{store}\ \mathit{hv}\ \mathit{rv} \mid \\
&\quad \texttt{restrict}\ \mathit{lv}\ \mathit{rv}\ \mathit{rv} \mid \texttt{subseg}\ \mathit{lv}\ \mathit{rv}\ \mathit{rv} \mid \texttt{plus}\ \mathit{lv}\ \mathit{rv}\ \mathit{rv}
\end{aligned}$$

Semantics

Assume a *decode* function that decodes integer to instructions:

$$\mathit{decode} : \text{Word} \to \text{Instructions}$$

Assume an *encodePerm* function that encodes a permission as an integer:

$$encodePerm : \text{Perm} \to \mathbb{Z}$$

Further, assume an inverse function, *decodePerm*, that decodes permissions

$$decodePerm : \mathbb{Z} \to \text{Perm}$$

$$\begin{array}{ll} & \text{if } \Phi.\text{reg}(\text{pc}) = (perm, base, end, a) \\ \Phi \to [\![decode(\Phi.\text{heap}(a))]\!] (\Phi) & \quad \text{and } base \leq a < end \\ & \quad \text{and } perm \in \{\text{rx}, \text{rwx}\} \\ \Phi \to failed & \text{otherwise} \end{array}$$

It is assumed that address 0 is used for I/O, so whatever resides in location 0 after an execution can be seen as a result. As we will see in the semantics, it is assumed that the initial process starts with a readwrite capability for this location.

$$executeAllowed(perm) = \begin{cases} true & \text{if } perm \in \{\text{rwx}, \text{rx}, \text{e}\} \\ false & \text{otherwise} \end{cases}$$

$$readAllowed(perm) = \begin{cases} true & \text{if } perm \in \{\text{rwx}, \text{rx}, \text{rw}, \text{ro}\} \\ false & \text{otherwise} \end{cases}$$

$$writeAllowed(perm) = \begin{cases} true & \text{if } perm \in \{\text{rwx}, \text{rw}\} \\ false & \text{otherwise} \end{cases}$$

$$updatePcPerm(w) = \begin{cases} (\text{rx}, base, end, a) & \text{if } w = (\text{e}, base, end, a) \\ w & \text{otherwise} \end{cases}$$

$$nonZero(w) = \begin{cases} true & \text{if } w \in \text{Cap or } w \in \mathbb{Z} \text{ and } w \neq 0 \\ false & \text{otherwise} \end{cases}$$

$$withinBounds((\_, base, end, a)) = \begin{cases} true & \text{if } base \leq a \leq end \\ false & \text{otherwise} \end{cases}$$

$$updatePc(\Phi) = \begin{cases} \Phi[\text{reg.pc} \mapsto newPc] & \text{if } \Phi.\text{reg}(\text{pc}) = (perm, base, end, a) \\ & \quad \text{and } newPc = (perm, base, end, a+1) \\ failed & \text{otherwise} \end{cases}$$

$$[\![\texttt{fail}]\!](\Phi) = \mathit{failed}$$

$$[\![\texttt{halt}]\!](\Phi) = (\mathit{halted}, \Phi)$$

$$[\![\texttt{jmp}\ lv]\!](\Phi) = \Phi[\text{reg.pc} \mapsto \mathit{updatePcPerm}(\Phi.\text{reg}(lv))]$$

$$[\![\texttt{jnz}\ lv\ rv]\!](\Phi) = \begin{cases} \Phi[\text{reg.pc} \mapsto \mathit{updatePcPerm}(\Phi.\text{reg}(lv))] & \text{if } \mathit{nonZero}(\Phi.\text{reg}(rv)) \\ \mathit{updatePc}(\Phi) & \text{if not } \mathit{nonZero}(\Phi.\text{reg}(rv)) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{load}\ \lfloor r_1 \rfloor\ \langle r_2 \rangle_h]\!] = \begin{cases} \mathit{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) & \text{if } \Phi.\text{reg}(r_2) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) = c \\ & \quad \text{and } \mathit{readAllowed}(\mathit{perm}) \text{ and } \mathit{withinBounds}(c) \\ & \quad \text{and } w = \Phi.\text{heap}(a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{store}\ \langle r_1 \rangle_h\ \lfloor r_2 \rfloor]\!] = \begin{cases} \mathit{updatePc}(\Phi[\text{heap}.a \mapsto w]) & \text{if } \Phi.\text{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) = c \\ & \quad \text{and } \mathit{writeAllowed}(\mathit{perm}) \text{ and } \mathit{withinBounds}(c) \\ & \quad \text{and } w = \Phi.\text{reg}(r_2) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{move}\ \lfloor r_1 \rfloor\ rv]\!] = \mathit{updatePc}(\Phi[\text{reg}.r_1 \mapsto \Phi.\text{reg}(rv)])$$

$$[\![\texttt{lea}\ \lfloor r_1 \rfloor\ rv]\!] = \begin{cases} \mathit{updatePc}(\Phi[\text{reg}.r_1 \mapsto c]) & \text{if either } n = rv \text{ or } rv = \lfloor r_2 \rfloor \text{ and } n = \Phi.\text{reg}(r_2) \\ & \quad \text{and in either case } n \in \mathbb{Z} \\ & \quad \text{and } \Phi.\text{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \\ & \quad \text{and } c = (\mathit{perm}, \mathit{base}, \mathit{end}, a + n) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{restrict}\ \lfloor r_1 \rfloor\ rv_1\ rv_2]\!] = \begin{cases} \mathit{updatePc}(\Phi[\text{reg}.r_1 \mapsto c]) & \text{if } \Phi.\text{reg}(rv_1) = c' \\ & \quad \text{and } c' = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \\ & \quad \text{and either } rv_2 = n \text{ or } \Phi.\text{reg}(rv_2) = n \\ & \quad \text{and in either case } n \in \mathbb{Z} \\ & \quad \text{and } \mathit{newPerm} = \mathit{decodePerm}(n) \\ & \quad \text{and } \mathit{newPerm} \sqsubseteq \mathit{perm} \\ & \quad \text{and } c = (\mathit{newPerm}, \mathit{base}, \mathit{end}, a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{plus}\ \lfloor r_1 \rfloor\ rv_1\ rv_2]\!] = \begin{cases} \mathit{updatePc}(\Phi[\text{reg}.r_1 \mapsto n_1 + n_2]) & \text{if for } i \in \{1, 2\} \\ & \quad n_i = rv_i \text{ or } n_i = \Phi.\text{reg}(rv_i) \\ & \quad \text{and in either case } n_i \in \mathbb{Z} \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{isptr}\ lv\ rv]\!] = \mathit{undefined}$$

$$[\![\texttt{setptr}\ lv\ rv]\!] = \mathit{undefined}$$

$$[\![\texttt{subseg}\ lv\ rv\ rv]\!] = \mathit{undefined}$$

1 Examples

1.1 Ticket Dispenser

Assume the instructions of some adversary resides in the memory starting at the memory location marked with adv_{base}. Assume that the register r_{io} initially contains a capability for the address in memory where I/O is written to (we assume this address is 0). We assume that entry capabilities for *adv* and *malloc* is ambiently availableDD: what does this mean?

. The following is a test program for a ticket dispenser:

$init_{base}$:	`store` $\langle r_{io}\rangle_h\ -1$	Initialize io to -1
	`move` $\lfloor r_0 \rfloor$ $\lfloor$pc$\rfloor$	Store the io capability on the stack
	`lea` $\lfloor r_0 \rfloor$ *iocap*	
	`store` $\langle r_0\rangle_h$ $\lfloor r_{io} \rfloor$	
	`move` $\lfloor r_{io} \rfloor$ 0	Overwrite io register
	`move` $\lfloor r_1 \rfloor$ *size*	Allocate memory for the ticket dispenser program (including memory for counter and capability for counter)
	`move` $\lfloor r_0 \rfloor$ $\lfloor$pc$\rfloor$	
	`lea` $\lfloor r_0 \rfloor$ 3	
	`jmp` *malloc*	
ret_{malloc} :	`store` $\langle r_1\rangle_h$ $(encode(i_1))$	Store the ticket dispenser program in the newly allocated memory
	`lea` $\lfloor r_1 \rfloor$ 1	
	`store` $\langle r_1\rangle_h$ $(encode(i_2))$	
	`lea` $\lfloor r_1 \rfloor$ 1	
	$\vdots$	
	`store` $\langle r_1\rangle_h$ $(encode(i_7))$	
	`lea` $\lfloor r_1 \rfloor$ 1	
	`move` $\lfloor r_0 \rfloor$ $\lfloor r_1 \rfloor$	Store a capability for the counter address after the ticket dispenser program
	`lea` $\lfloor r_0 \rfloor$ 1	
	`store` $\langle r_1\rangle_h$ $\lfloor r_0 \rfloor$	
	`lea` $\lfloor r_1 \rfloor$ 1	
	`move` $\lfloor r_2 \rfloor$ $\lfloor$pc$\rfloor$	Store a capability for the counter in this code
	`lea` $\lfloor r_2 \rfloor$ *counter*	
	`store` $\langle r_2\rangle_h$ r_0	
	`move` $\lfloor r_2 \rfloor$ 0	Overwrite capability for counter
	`store` $\langle r_1\rangle_h$ 0	Initialize counter to 0
	`lea` $\lfloor r_1 \rfloor\ -8$	Capability points to start of td code [1]
	`restrict` $\lfloor r_1 \rfloor$ $\lfloor r_1 \rfloor$ $(encodePerm(\text{e}))$	Restrict cap. to td
	`move` $\lfloor r_0 \rfloor$ $\lfloor$pc$\rfloor$	Setup a return pointer for the adversary
	`lea` $\lfloor r_0 \rfloor$ 5	
	`restrict` $\lfloor r_0 \rfloor$ $\lfloor r_0 \rfloor$ $(encodePerm(\text{e}))$	
	`jmp` *adv*	Jump to the adversary
	0	Address reserved for io cap.
	0	Address reserved for counter cap.
ret_{adv} :	`move` $\lfloor r_0 \rfloor$ $\lfloor$pc$\rfloor$	Retrieve the io capability
	`lea` $r_0\ -2$	
	`load` $\lfloor r_{io} \rfloor$ $\langle r_0\rangle_h$	
	`lea` r_0 1	Retrieve the counter capability
	`load` $\lfloor r_1 \rfloor$ $\langle r_0\rangle_h$	
	`load` $\lfloor r_0 \rfloor$ $\langle r_1\rangle_h$	Read the counter and write it to io address
	`store` $\langle r_{io}\rangle_h$ $\lfloor r_0 \rfloor$	
$init_{end}$:	`halt`	

Here *size* is 9. The variables *counter* and *iocap* are respectively the offsets to the addresses reserved for the counter and io capabilities. $i_1 \ldots i_7$ refers to the

instructions in the following ticket dispenser:

$$
\begin{array}{ll}
i_1: & \texttt{move}\ \lfloor r_2 \rfloor\ \lfloor \mathrm{pc} \rfloor \\
i_2: & \texttt{load}\ \lfloor r_2 \rfloor\ size - 2 \\
i_3: & \texttt{load}\ \lfloor r_1 \rfloor\ \langle r_2 \rangle_h \\
i_4: & \texttt{plus}\ \lfloor r_3 \rfloor\ \lfloor r_1 \rfloor\ 2 \\
i_5: & \texttt{store}\ \langle r_2 \rangle_h\ \lfloor r_3 \rfloor \\
i_6: & \texttt{move}\ \lfloor r_2 \rfloor\ 0 \\
i_7: & \texttt{jmp}\ \lfloor r_0 \rfloor
\end{array}
$$

The heap layout for the above program can be seen in Figure 2.

Addr.	Content
0	i_1
1	i_2
2	i_3
3	i_4
4	i_5
5	i_6
5	i_7
7	Cap. for addr. 8
8	The counter

Figure 2: The heap layout for the ticket dispenser.

In the JS paper, the ticket dispenser program dereferences the counter and returns the value. It does not seem like a similar thing would work here as the adversary can do even more. In particular, the adversary can execute a `halt` instruction that will cause the machine to halt (successfully). The adversary could also choose never to return to the return address that we set up for him.

Lemma 1 (Ticket dispenser test)**.** DD: I would prefer more precise, symbolic statements of assumptions and less prose.

Given a configuration Φ*, if*

Φ.heap *contains the ticket dispenser test program starting at the address labelled* $init_{base}$*,*

Φ.heap *contains the adversary code starting at label* adv_{base}*. The adversary is assumed to have no capabilities on the heap, but it has access to the ambient malloc capability!* LS: maybe mention that malloc is also on the heap

Φ.heap *contains the malloc code, and the malloc code satisfy* ι_{malloc}*, i.e.,* $\forall n.\, heap|_{\{malloc_{base},\ldots,malloc_{end}\}} :_n W_{malloc}$*.*

All the above programs are disjoint.

adv* and *malloc *capabilities are ambiently available at all times.*

Φ.reg(pc) *is* $(\text{rwx}, init_{base}, init_{end}, init_{base})$.

Φ.reg(r_{io}) *is* $(\text{rw}, 0, 0, 0)$.

the remaining registers *all contains 0,*

and $\Phi \rightarrow^* (halted, \Phi')$*, then the I/O address (that is* $\Phi'.\text{heap}(0)$*) contains either* -1 *or a number that is* ≥ 0 *and even.* ■

In the initial configuration the pc register is assumed to have an execute and write permission for the area of the heap where the program resides.

The ticket dispenser test program uses *malloc*. To be able to reason about programs that use *malloc*, we assume that it follows the following specification. If you provide it with one argument, namely the size of the allocation you wish and a return capability, then at some point pc will contain an executable capability based on this return capability and the return register will contain a rwx capability for the freshly allocated memory.

Specification 1 (Malloc v.2)**.**

$$
\begin{array}{l}
\forall \Phi \in \text{Mem}.\, \forall hs_{footprint}, hs_{frame} \in \text{HeapSegment}. \\
\quad \forall n, size \in \mathbb{N}.\, \forall b, e, a, m_b, m_e \in \text{Addr}.\, \forall p \in \text{Perm}.\, \forall W \in \text{World}. \\
\quad\quad \exists i, \iota_{malloc}. \\
\quad\quad\quad W = [i \mapsto \iota_{malloc}] \wedge \\
\quad\quad\quad \text{dom}(hs_{footprint}) = [m_b, m_e] \wedge \\
\quad\quad\quad \Phi.\text{heap} = hs_{footprint} \uplus hs_{frame} \wedge \\
\quad\quad\quad hs_{footprint} :_n W \wedge \\
\quad\quad\quad \Phi.\text{reg}(r_1) = size \wedge \\
\quad\quad\quad size \geq 0 \wedge \\
\quad\quad\quad \Phi.\text{reg}(r_0) = (p, b, e, a) \wedge \\
\quad\quad\quad \Phi.\text{reg}(\text{pc}) = (\text{e}, m_b, m_e, m_b) \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad \exists \Phi' \in \text{Mem}.\, \exists hs'_{footprint} \in \text{HeapSegment}. \\
\quad\quad\quad\quad\quad \exists j, k \in \mathbb{N}.\, \exists b', e' \in \text{Addr}.\, \exists W' \in \text{World}. \\
\quad\quad\quad\quad\quad\quad \Phi \rightarrow_j \Phi' \wedge \\
\quad\quad\quad\quad\quad\quad \Phi'.\text{heap} = hs'_{footprint} \uplus hs_{frame} \wedge \\
\quad\quad\quad\quad\quad\quad W' = [k \mapsto \iota^0_{b',e'}, i \mapsto \iota_{malloc}] \\
\quad\quad\quad\quad\quad\quad W' \sqsupseteq W \wedge \\
\quad\quad\quad\quad\quad\quad hs'_{footprint} :_{n-j} W' \wedge \\
\quad\quad\quad\quad\quad\quad \Phi'.\text{reg}(\text{pc}) = (updatePcPerm(p), b, e, a) \wedge \\
\quad\quad\quad\quad\quad\quad size = e' - b' \\
\quad\quad\quad\quad\quad\quad \Phi'.\text{reg}(r_1) = (\text{rwx}, b', e', b') \wedge \\
\quad\quad\quad\quad\quad\quad \forall r \in \text{RegisterName} \setminus \{\text{pc}, r_1\}.\, \Phi'.\text{reg}(r) = 0
\end{array}
$$

■

The specification uses the malloc island defined as LS: I am not satisfied with the pc in the premise. The exact capability necessary very much depend on the implementation (this is mostly a note to myself.)
DD: I agree with the remark about the pc. Perhaps leave the form of the pc unspecified as it is not important for malloc users?
DD: Resulting frame should be equal to original.
LS: Rewrite the specification when we have settled on a reasonable way to handle ABI.

Specification 2 (Malloc)**.** DD: can you state this more systematically, with less prose, more explicit assumptions and conclusion?
Take W_{malloc} *the world with exactly one island, namely the unspecified island* ι_{malloc} *that governs the internal state of malloc. If we invoke malloc in a memory* $h = hs_{frame} \uplus hs_{footprint}$ *that is the disjoint sum of a frame part and a footprint part and the latter satisfies this world up to steps* n*, i.e.,* $hs_{footprint} :_n W_{malloc}$*. Then malloc will successfully terminate and return a result in a heap* $h' = hs_{frame} \uplus hs'_{footprint}$ *that is the disjoint sum of the same unmodified frame part and a potentially new footprint part, such that there is a future world* W' *of* W_{malloc} *consisting of exactly two islands, where the first is an instance of* ι_{malloc} *and the second is the region* $\iota^0_{base,end}$ *for an appropriately-sized interval* $base, end$ *and the new footprint part satisfies this new world* $hs'_{footprint} :_n W'$*.*

To invoke *malloc means that the capability in* pc *points to a* `jmp` *malloc instruction (and that this address is within its capabilities). Further in the register* r_0 *there has to be a return capability, i.e., a capability of the form* $(perm, returnStart, returnEnd, returnAddr)$ *where perm is either,* rwx*,* rx*, or* e*. Lastly, the register* r_1 *has to contain the argument for malloc which is the desired length of the piece of memory to be allocated.*

To return *from malloc means that the register* pc *contains the capability* $(perm', returnStart, returnEnd, returnAddr)$ *where* $perm'$ *is* rwx *if perm was* rwx *and* rx *otherwise. Further, register* r_1 *contains* $(\text{rwx}, base, end, base)$ *which is a capability for the allocated memory.* ■

Proof (Lemma 1). Let configuration Φ be given where the heap is shaped as follows:

- Φ.heap contains the ticket dispenser test program (tdtp) starting at address $init_{base}$.
- Φ.heap contains the adversary code starting at address adv_{base} and ending at address adv_{end}. The adversary code contains no capabilities.
- Φ.heap contains the malloc code.
- (and so on)

and the register file contains:

1. $\Phi.\text{reg}(r_{io}) = (\text{rw}, 0, 0, 0)$
2. $\Phi.\text{reg}(\text{pc}) = (\text{rwx}, \mathit{init}_{base}, \mathit{init}_{end}, \mathit{init}_{base})$
3. $\Phi.\text{reg}(r) = 0$ for the remaining registers r in the register file.

If we start running in Φ until `jmp` *malloc* has just been executed, then the configuration is the same as Φ except:

- $\Phi.\text{heap}(0)$ contains -1
- the io capability is saved to the designated address in the tdtp
- r_1 contains the size of the ticket dispenser program (9)
- r_0 contains the return capability $(\text{rwx}, \mathit{init}_{base}, \mathit{init}_{end}, \mathit{ret}_{malloc})$

Call this new heap Φ'

To be able to say something about the jump to *malloc*, we use its specification. Take $hs_{footprint}$ to be the part of the heap *malloc* resides in and everything that has not been allocated according to the assumptions. Take hs_{frame} to be everything else. We have $hs_{footprint} :_n W_{malloc}$ as the world only has one region, namely the one that governs *malloc*.DD: explicit assumption about this needed? According to the specification of *malloc*, we get $\Phi' \rightarrow^* \Phi''$ where

- $\Phi'.\text{heap} = hs_{frame} \uplus hs'_{footprint}$
- $\Phi'.\text{reg}(r_1) = (\text{rwx}, \mathit{base}, \mathit{end}, \mathit{base})$ for some addresses *base* and *end*
- $\Phi'.\text{reg}(\text{pc}) = (\text{rwx}, \mathit{init}_{base}, \mathit{init}_{end}, \mathit{ret}_{malloc})$

Further for W' with exactly two regions ι_{malloc} and $\iota^0_{base,end}$ we have that the new footprint satisfies this world, i.e., $hs'_{footprint} :_n W'$.

If we continue running from Φ' until just after the point we have executed `jmp` *adv*, then pc points to an execute capability for the adversary, and we reach a configuration Φ'' which is the same as Φ except:

- $\Phi''.\text{heap}(0)$ contains -1
- capabilities for *the counter* and io are saved to the designated addresses in the tdtp
- the ticket dispenser code resides on the heap starting at the address *base* (the code is as seen in Figure 2)
- $\Phi''.\text{reg}(\text{pc}) = (\text{rx}, \mathit{adv}_{base}, \mathit{adv}_{end}, \mathit{adv}_{base})$
- $\Phi''.\text{reg}(r_0) = (\text{e}, \mathit{init}_{base}, \mathit{init}_{end}, \mathit{ret}_{adv})$
- $\Phi''.\text{reg}(r_1) = (\text{e}, \mathit{base}, \mathit{end}, \mathit{base})$
- all other registers contain 0

We now want to use the FTLR. To do so, we need to define a world that represents our current heap. We pick the following world:

- ι_{io} the region governs the heap segment that only has address 0. The value of address 0 should be -1 or ≥ 0 and even.
- ι_{td} this region governs heap segments with the domain $[base, end]$ for the addresses $[base, end - 1]$ they are the same as in Φ''. $hs(base)$ is positive and even.
- ι_{tdtp} this region governs the ticket dispenser test programs. The domain of a heap segment in this region is that of the ticket dispenser test program and the contents of the heap segment should be the same as it is in Φ''.
- $\iota_{adv_{base}, adv_{end}}$
- ι_{malloc}

To use the FTLR, we have to argue that the read condition is satisfied for the region, the adversary is in. This is easily shown, as we have chosen the region for the adversary to be the standard one, so with adv_{base} and adv_{end} as witnesses, we need to show $\iota_{adv_{base}, adv_{end}} \subseteq \iota_{adv_{base}, adv_{end}}$.

So we can conclude that $(n, (\mathrm{rx}, adv_{base}, adv_{end}, adv_{base})) \in \mathcal{E}(W'')$ where W'' is the world with the following regions: Assuming we can show the following three things:

1. $\Phi''.\mathrm{reg} \in \mathcal{R}(W'')$
2. $(n, c = (\mathrm{rx}, init_{base}, init_{end}, ret_{adv}))] \in \mathcal{K}(W'')$
3. $\Phi''.\mathrm{heap}| :_n W''$ where the we have limited the heap to the segments governed by the regions in W''.

then it follows from the fact that the adversary program is in the expression predicate that

$$\Big(n, \Phi^{(3)} = (\Phi''[\mathrm{reg}.r_0 \mapsto c][\mathrm{reg}.r_1 \mapsto (\mathrm{rx}, adv_{base}, adv_{end}, adv_{base})].\mathrm{reg}, \Phi''.\mathrm{heap})\Big) \in \mathcal{O}(W'').$$

From this and the fact that we know the execution terminates, we know that $\Phi^{(3)} \rightarrow^k (halted, h)$ for some heap h and $k \leq n$. From this we can conclude that the heap segment that only consists of address 0 is accepted by region ι_{io} which means that either $h(0)$ is -1 or positive and even which is exactly what the lemma says.

It remains to show that 1-3 holds. For 1, all registers but r_0 and r_1 are easy as they contain 0 which is always in the value predicate. For r_0 and r_1, we do need to do some reasoning. Both capabilities have one things in common, namely that the underlying programs do not use the arguments.

$\Phi''.\mathrm{reg}(r_0)$ contains the capability $(\mathrm{e}, init_{base}, init_{end}, ret_{adv})$ and we need to show that it is in $\mathcal{V}(W'')$. Let $n' < n$ and $W^{(3)} \sqsupseteq W''$ be given and

show $(n', (\text{rx}, init_{base}, init_{end}, ret_{adv})) \in \mathcal{E}(W^{(3)})$. Let $reg \in \mathcal{R}(W^{(3)})$, $(n', c) \in \mathcal{K}(W^{(3)})$, $heap :_{n'} W^{(3)}$ and $heap_f$ be given. We do not really care about the register file and the continuation as the ticket dispenser test program does not use them. As $heap$ satisfies $W^{(3)}$, parts of the heap must satisfy the invariants ι_{tdtp} and ι_{td} which means the ticket dispenser program and the ticket dispenser test program is unaltered. It also means that the counter in the ticket dispenser is ≥ 0 and even. We need to show $\big(n', (reg[r_0 \mapsto c][r_1 \mapsto](\text{rx}, init_{base}, init_{end}, ret_{adv}), heap \uplus heap_f)\big) \in \mathcal{O}(W^{(3)})$. If n' is sufficiently small, then we run out of steps before the program halts in which case the configuration is trivially in the $\mathcal{O}$ predicate. If the execution halts successfully, then the ticket dispenser test program will have moved the value of the counter in the ticket dispenser to address 0, the designated I/O address. As this is a numerical value, it will trivially be in the value predicate. Further, the island that governs address 0 is ι_{io}, so we have to argue that the value is ≥ 0 and even. The value came from the ticket dispenser which is governed by the region ι_{td} so it is indeed ≥ 0 and even.

In register r_1 we have the capability $(\text{e}, base, end, base)$ and we need to show $(n, (\text{rx}, base, end, base)) \in \mathcal{E}(W'')$. To this end, we go through the same motion as for r_0: Let $n' < n$, $W^{(3)} \sqsupseteq W''$, $reg \in \mathcal{R}(W^{(3)})$, $(n', c) \in \mathcal{K}(W^{(3)})$, $heap :_{n'} W^{(3)}$ and $heap_f$ be given. Show

$$\big(n', (reg[r_0 \mapsto c][r_1 \mapsto](\text{rx}, base, end, base), heap \uplus heap_f)\big) \in \mathcal{O}(W^{(3)}).$$

Again the heap satisfaction makes sure that the ticket dispenser program is as we expect, so we can execute it. If we run out of steps before we reach `jmp` $\lfloor r_0 \rfloor$, then it is in the observation predicate. Otherwise, we reach a new configuration $\Phi^{(3)}$ where the registers are the same as in reg except for r_0, r_1, r_2, and pc which will be c, the previous counter value, 0, and $updatePcPerm(()c)$ respectively. 0 is trivially in the value predicate and c is in the value predicate as it is in the continuation predicate, so reg is in the register-file predicate. If we split the heap part of $\Phi^{(3)}$ into the same two parts as we had in the initial assumption, then we need to argue that the non-frame part still satisfies the world. The only change made to the heap was updating the counter. The heap segments that did not change still satisfy their regions. We have to argue that the heap segment $[base, end]$ still satisfies the region ι_{td}. The ticket dispenser code remains unchanged and the counter is ≥ 0 and even as it was computed by adding two to a value which was ≥ 0 and even, so the ticket dispenser heap segment satisfies ι_{td}. Now we have all the necessary conditions satisfied to be able to use the fact that c is in the continuation predicate to conclude

LS: did I mess up the steps here?

$$\big(n', (reg[r_0 \mapsto c][\text{pc} \mapsto updatePcPerm(c)], heap' \uplus heap_f)\big) \in \mathcal{O}$$

(where $heap'$ is the new heap). From this we get that continuing from the configuration we end up with no more steps, failing, or in a halting state that satisfies the world.

DD: this duplicates the argument about $(\text{e}, init_{base}, init_{end}, init_{base} + ret) \in \mathcal{V}(W'')$. Find a way to share this?

For condition 2, we need to show

$$(n, c = (\mathrm{e}, \mathit{init}_{base}, \mathit{init}_{end}, \mathit{ret}_{adv}))] \in \mathcal{K}(W'').$$

To this end, let $W^{(3)} \sqsupseteq W''$, $\mathit{reg} \in \mathcal{R}(W^{(3)})$, $\mathit{heap} :_{n'} W^{(3)}$ and heap_f be given. We then need to argue

$$\big(n, (\mathit{reg}[\mathrm{pc} \mapsto \mathit{updatePcPerm}(c)], \mathit{heap} \uplus \mathit{heap}_f)\big) \in \mathcal{O}(W^{(3)}).$$

As part of the heap satisfies a future world of W'', we know that the ticket dispenser test program remains unaltered. If we run it (and do not run out of steps), then it loads the counter, stores it to address 0, and halts. From the heap satisfaction, we get that the counter is positive and even, so the value we store to address 0 is positive and even. The value is trivially in the value relation as it is an integer. Further the region that governs address 0 is ι_{io} which requires the value there to be positive and even which it is. So in this case the configuration is in the observation predicate. If we run out of steps, then it is also in the observation relation.

Finally, we need to show $\Phi''.\mathrm{heap}| :_n W''$. The domains of the heap segments are assumed to be disjoint, and we have limited the heap to only be the parts considered in the regions, so the first condition of heap satisfaction is okay. Let us take the regions one at a time and briefly argue that the corresponding heap segment satisfies the region:

- ι_{io} governs address 0, which i initial -1. This region requires it to be -1 or ≥ 0 and even, so it is initially satisfied.
- ι_{tdtp}, this region was constructed so the part of the heap it governs remains unaltered.
- ι_{td}, apart from the address with the counter, this region is also constructed so this heap segment does not change. With respect to the counter, it requires it to be ≥ 0 and even. The counter starts as 0, so also this cell satisfies the region.
- $\iota_{adv_{base}, adv_{end}}$ the adversary code is assumed to be instructions only which are trivially satisfied by the standard island.
- ι_{malloc}, from the specification of *malloc* we got that the resulting heap after the allocation satisfied a world with ι_{malloc}. We use the same region and we have not changed that part of the heap, so it still satisfies this region.

We have now shown the three conditions and thus proven Lemma 1. □

2 Logical Relation

2.1 Recursive Domain Equation

The goal is to solve the following domain equation:

$$\mathrm{Wor} = \mathbb{N} \xrightarrow{fin} (\mathrm{State} \times \mathrm{Rel} \times (\mathrm{State} \to (\mathrm{Wor} \overset{mon,\, ne}{\to} \mathrm{UPred}(\mathrm{HeapSegment}))))$$

Where State is a set of states with all the ones we use in this paper.

$$\mathrm{Rel} = \{R \in \mathcal{P}(\mathrm{State}^2) \mid R \text{ is reflexive and transitive}\}$$

This cannot be solved with sets, so we use preordered complete ordered families of equivalences where it is possible to solve such an equation that ressembles the above one, namely it is possible to find an isomorphism ξ and preordered c.o.f.e. W such that

$$\xi : \mathrm{Wor} \cong \blacktriangleright(\mathbb{N} \xrightarrow{fin} (\mathrm{State} \times \mathrm{Rel} \times (\mathrm{State} \to (\mathrm{Wor} \overset{mon,\, ne}{\to} \mathrm{UPred}(\mathrm{HeapSegment})))))$$

Definition 1 (o.f.e's)**.** *An ordered family of equivalences (o.f.e.) is a set and a family of equivalences,* $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$*. The familly of equivalences have to satisfy the following properties*

- $\overset{0}{=}$ *is a total relation on* X
- $\forall n.\, \forall x, y \in S.\, x \overset{n+1}{=} y \Rightarrow x \overset{n}{=} y$
- $\forall x, y.\, (\forall n.\, x \overset{n}{=} y) \Rightarrow x = y$

■

DD: I suppose you're using a standard ultrametric metric to make an o.f.e. a metric space?

Definition 2 (c.o.f.e.'s)**.** *A complete orderede family of equivalences is an o.f.e.* $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ *where all Cauchy sequences in* X *have a limit in* X*.* ■

Definition 3 (Preordered c.o.f.e.'s)**.** *A preordered c.o.f.e. is a c.o.f.e. equiped with a preorder on* X*,* $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$*.*

- *The ordering preserves limits. That is, for Cauchy chains* $\{a_n\}_n$ *and* $\{b_n\}_n$ *in* X *if* $\{a_n\}_n \sqsupseteq \{b_n\}_n$*, then* $\lim\{a_n\}_n \sqsupseteq \lim\{b_n\}_n$*.*

■

Definition 4 (Preordered c.o.f.e. construction: Finite-partial function)**.** *Given a set* S *and preordered c.o.f.e.* X*,* $S \xrightarrow{fin} X$ *is a preordered c.o.f.e. with the ordering*

$$f \sqsupseteq g$$
$$\textit{iff}$$
$$\mathrm{dom}(f) \supseteq \mathrm{dom}(g) \quad \textit{and} \quad \forall n \in S.\, f(n) \sqsupseteq g(n)$$

■

We need the following constructions to create the preordered c.o.f.e. needed to solve the recursive domain equation. DD: this sentence doesn't parse :)

Definition 5 (Preordered c.o.f.e. construction: Function)**.** *Given a set S and c.o.f.e. HP, $S \to HP$ is a preordered c.o.f.e. with the ordering*

$$f \sqsupseteq g$$
$$\textit{iff}$$
$$\forall s \in \mathrm{dom}(f).\, f(s) \sqsupseteq g(s)$$

■

Definition 6 (Preordered c.o.f.e. construction: Monotone, non-expansive function)**.** *Given a preordered c.o.f.e. W and preordered c.o.f.e. U, $W \overset{mon,\ ne}{\to} U$ is a preordered c.o.f.e. with the ordering*

$$f \sqsupseteq g$$
$$\textit{iff}$$
$$\forall s \in \mathrm{dom}(f).\, f(s) \sqsupseteq g(s)$$

■

The above are standard constructions, so they are used here without showing they are in fact well-defined as shown in Birkedal and Bizjak [2014].

Definition 7 (Preordered c.o.f.e. construction: Region)**.** *Given a c.o.f.e. H, the tuple*

$$(\mathrm{State} \times \mathrm{Rel} \times H)$$

is a preordered c.o.f.e. with the ordering

$$(s_2, \phi_2, H_2) \sqsupseteq (s_1, \phi_1, H_2)$$
$$\textit{iff}$$
$$H_2 = H_1 \ \textit{and}\ \phi_2 = \phi_1 \ \textit{and}\ (s_1, s_2) \in \phi_2$$

■

Lemma 2 (Region definition well-defined)**.** *The construction in Definition 7 is a preordered c.o.f.e.. That is*

- *It is a c.o.f.e. (this is a standard construction)*
- *$\sqsupseteq$ is a transitive and reflexive relation.*
- *$\sqsupseteq$ preserves limits.*
  That is for Cauchy chains $\{a_n\}_n$ and $\{b_n\}_n$ if

  $$\{a_n\}_n \geq \{b_n\}_n,$$

  then

  $$\lim\{a_n\}_n \sqsupseteq \lim\{b_n\}_n$$

■

The category of c.o.f.e.'s is the category whith c.o.f.e.'s as objects and non-expansive functions as morphisms. We denote this category $\mathbb{C}$. The category of preordered c.o.f.e.'s has preordered c.o.f.e.'s as objects and monotone and non-expansive functions as morphisms. We denote this category $\mathbb{P}$.

Define functors K, R, and G as follows:

$$K : \mathbb{P} \to \mathbb{P}$$
$$K(R) = \mathbb{N} \xrightarrow{fin} R$$
$$K(f) = \lambda\phi.\, \lambda n.\, f(\phi(n))$$

$$R : \mathbb{C} \to \mathbb{P}$$
$$R(H) = \text{State} \times \text{Rel} \times H$$
$$R(h) = \lambda(s, \Phi, H).\, (s, \Phi, h(H))$$

$$G : \mathbb{P}^{op} \to \mathbb{C}$$
$$G(W) = \text{State} \xrightarrow{ne} W \xrightarrow{mon,\, ne} \text{UPred}(HS)$$
$$G(g) = \lambda H.\, \lambda st.\, \lambda x.\, H(st)(g(x))$$

We first show that K, R, and G are well-defined mappings.

Lemma 3 (World finite partial mapping)**.** *For all f and ϕ, $K(f)(\phi)$ is a finite partial mapping.* ■

Lemma 4 (Heap segment predicate monotone)**.** *For all g, H, and st*

$$G(g)(H)(st)$$

is non-expansive. ■

Lemma 5 (Heap segment predicate non-expansive)**.** *For all g, H, and st*

$$G(g)(H)(st)$$

is monotone. ■

Next we show that K, R, and G are in fact functors:

Lemma 6 (K functorial)**.**

1. $K(f) : K(X) \to K(Y)$ *is monotone and non-expansive for* $f : X \xrightarrow{mon,\, ne} Y$
2. $K(f \circ g) = K(f) \circ K(g)$ *for* $f : Z \xrightarrow{ne} Y$ *and* $g : X \xrightarrow{ne} Z$
3. $K(id) = id$

■

Lemma 7 (R functorial)**.**

1. $R(f) : R(X) \to R(Y)$ *is non-expansive and monotone for* $f : X \overset{ne}{\to} Y$
2. $R(f \circ g) = R(f) \circ R(g)$ *for* $f : Z \overset{ne}{\to} Y$ *and* $g : X \overset{ne}{\to} Z$
3. $R(id) = id$

■

Lemma 8 (G functorial)**.**

1. $G(f) : G(Y) \to G(X)$ *is non-expansive for* $f : X \overset{mon.\ ne}{\to} Y$
2. $G(f \circ g) = G(g) \circ G(f)$ *for* $f : Z \overset{ne}{\to} Y$ *and* $g : Y \overset{ne}{\to} Z$
3. $G(id) = id$

■

We now compose the above functors into the functor we actually want to use: $F = K \circ R \circ G$, $F : \mathbb{P}^{op} \to \mathbb{P}$.

Lemma 9 (F functorial)**.**

1. $F(f) : F(Y) \to F(X)$ *is monotone and non-expansive for* $f : X \overset{mon.\ ne}{\to} Y$
2. $F(f \circ g) = F(g) \circ F(f)$ *for* $f : Z \overset{ne}{\to} Y$ *and* $g : Y \overset{ne}{\to} Z$
3. $F(id) = id$

■

Lemma 10 (F locally non-expansive)**.** *For all* $f, g : X \to Y$*, if* $f \overset{n}{=} g$*, then* $F(f) \overset{n}{=} F(g)$*.* ■

With F being locally-non-expansive, we can pre- or post-compose with later ($\blacktriangleright$) to get a locally contractive function. In this case we construct F' by post-copmosition of $\blacktriangleright$:

$$F'(\text{Wor}) = \blacktriangleright(F(\text{Wor}))$$

We have a theorem that gives us a solution to the recurisve domain equation

$$\text{Wor} \cong F'(\text{Wor}) = \blacktriangleright(\mathbb{N} \overset{fin}{\rightharpoonup} (\text{State} \times \text{Rel} \times (\text{State} \to \text{Wor} \overset{mon.\ ne}{\to} \text{UPred}(\text{HeapSegment}))))$$

The solution to the recursice domain equations is presented by Birkedal et al. [2010]. They solve it in pre-ordered, non-empty, complete, 1-bounded ultrametric spaces, but they have a simple correspondence to pre-ordered c.o.f.e.'s.

2.2 Worlds

Assume preordered c.o.f.e. Wor and isomorphism ξ such that:

$$\xi : \mathrm{Wor} \cong \blacktriangleright(\mathbb{N} \xrightarrow{fin} (\mathrm{State} \times \mathrm{Rel} \times (\mathrm{State} \to (\mathrm{Wor} \overset{mon,\, ne}{\to} \mathrm{UPred}(\mathrm{HeapSegment})))))$$

ξ is a morphism in the category $\mathbb{P}$ (the category of preordered c.o.f.e.'s), so it is non-expansive and monotone.

We now define regions as

$$\mathrm{Region} \stackrel{def}{=} (\mathrm{State} \times \mathrm{Rel} \times (\mathrm{State} \to (\mathrm{Wor} \overset{mon,\, ne}{\to} \mathrm{UPred}(\mathrm{HeapSegment}))))$$

define region names to be natural numbers, i.e.,

$$\mathrm{RegionName} \stackrel{def}{=} \mathbb{N}$$

and define worlds as

$$\mathrm{World} \stackrel{def}{=} \mathrm{RegionName} \xrightarrow{fin} \mathrm{Region}$$

To define future worlds and regions, We use the ordering inherited from the preordered c.o.f.e.'s.

Definition 8 (Future regions)**.** *For regions* $(s_2, \phi_2, H_2), (s_1, \phi_1, H_1) \in \mathrm{Region}$

$$(s_2, \phi_2, H_2) \sqsupseteq (s_1, \phi_1, H_1) \qquad \textit{iff} \qquad (\phi_1, H_1) = (\phi_2, H_2) \ \textit{and} \ (s_1, s_2) \in \phi_2$$

■

Definition 9 (Future worlds)**.** *For* $W, W' \in \mathrm{World}$

$$W' \sqsupseteq W \qquad \textit{iff} \qquad \begin{array}{c} \mathrm{dom}(W') \supseteq \mathrm{dom}(W) \\ \textit{and} \\ \forall r \in \mathrm{dom}(W).\, W'(r) \sqsupseteq W(r) \end{array}$$

■

Lemma 11 (Future worlds transitive)**.**

$$\begin{array}{l} \forall W, W', W''. \\ \quad W'' \sqsupseteq W' \wedge W' \sqsupseteq W \Rightarrow W'' \sqsupseteq W \end{array}$$

■

Lemma 12.

$$\begin{array}{l} \forall W_1, W_2, W_1'. \\ \quad W_1 \stackrel{n}{=} W_2 \wedge W_1' \sqsupseteq W_1 \Rightarrow \\ \qquad \exists W_2'.\, W_2' \sqsupseteq W_2 \wedge W_1' \stackrel{n}{=} W_2' \end{array}$$

■

Definition 10 (n-subset for regions)**.** *For regions* $(s_1, \phi_1, H_1), (s_2, \phi_2, H_2) \in$ Region

$$(s_1, \phi_1, H_1) \overset{n}{\subseteq} (s_2, \phi_2, H_2) \qquad \textit{iff} \qquad \begin{array}{c} (s_1, \phi_1) = (s_2, \phi_2) \\ \textit{and} \\ \forall W \in \text{Wor}.\, H_1\ s_1\ W \overset{n}{\subseteq} H_2\ s_2\ W \end{array}$$

■

Definition 11 (Heap satisfaction/erasure)**.**

$$\begin{array}{l} hs :_n W \\ \quad \textit{iff} \\ \exists R : \text{dom}(W) \to \text{HeapSegment}. \\ \quad hs = \biguplus_{r \in \text{dom}(W)} R(r) \\ \quad \textit{and} \\ \quad \forall r \in \text{dom}(W).\, \forall n' < n.\, (n', R(r)) \in W(r).H(W(r).s)(\xi^{-1}(W)) \end{array}$$

■

Lemma 13 (Heap satisfaction uniformity)**.**

$$\begin{array}{l} \forall n, n', hs, W. \\ \quad hs :_n W \wedge n' \leq n \Rightarrow hs :_{n'} W \end{array}$$

■

Lemma 14 (Heap satisfaction non-expansive)**.**

$$\begin{array}{l} \forall n, hs, W, W'. \\ \quad hs :_n W \wedge W \overset{n}{=} W' \Rightarrow hs :_n W' \end{array}$$

■

2.3 Logical Relation Setup

Our logical relation is defined using multiple recursive definitions, so the definitions in the following subsections are defined simultaneously. We want to define the value relation as the fixed-point given by Banach's fixed-point theorem, so all our definitions will be parameterized with the value relation.

2.3.1 Observation Relation

In order to define the expression relation, we define an observation relation.

$$\begin{aligned}\mathcal{O} \quad &: \quad \text{World} \xrightarrow{ne} \text{UPred}(\text{Reg} \times \text{HeapSegment}) \\ \mathcal{O}(W) \stackrel{def}{=} &\{(n, (reg, hs)) \mid \\ &(\forall heap_f, heap', i \leq n.\, (reg, hs \uplus heap_f) \rightarrow_i (\mathit{halted}, heap') \\ &\Rightarrow \exists W' \sqsupseteq W.\, \exists hs'.\, heap' = hs' \uplus heap_f \wedge hs' :_{n-i} W'\end{aligned}$$

A pair of a register and a heap segment is "good" if we can put it together with a frame heap, so we can execute it. The execution should then end up in a heap where the frame remains the same and the remaining heap segment satisfies the world.

Note that the operational semantic is total, so we cannot get stuck. If the execution ends up in a *failed* configuration, then we do not care about the heap and the registers. This is why, we only have requirements on the result when we end up in a *halted* configuration.

The following lemmas show that the observation relation is well-defined.

Lemma 15 (Observation relation uniformity)**.**

$$\begin{aligned}&\forall n' < n.\, \forall W.\, \forall reg.\, \forall hs. \\ &\quad (n, (reg, hs)) \in \mathcal{O}(W) \Rightarrow (n', (reg, hs)) \in \mathcal{O}(W)\end{aligned}$$

■

Lemma 16 (Observation relation non-expansive in worlds)**.**

$$\begin{aligned}&\forall W, W', n. \\ &\quad W \stackrel{n}{=} W' \Rightarrow \mathcal{O}(W) \stackrel{n}{=} \mathcal{O}(W')\end{aligned}$$

■

2.3.2 Regiser-File Relation

This relation is used in the definition of the continuation relation as well as the expression relation.

$$\begin{aligned}\mathcal{R} \quad &: \quad (\text{World} \xrightarrow{mon,\, ne} \text{UPred}(\text{Word})) \xrightarrow{ne} \text{World} \xrightarrow{mon,\, ne} \text{UPred}(\text{Reg}) \\ \mathcal{R} \stackrel{def}{=} &\lambda \mathcal{V}.\, \lambda W.\, \{(n, reg) \mid \forall r \in \text{RegisterName} \setminus \{\text{pc}\}. \\ &\qquad\qquad (n, reg(r)) \in \mathcal{V}(W)\}\end{aligned}$$

Well-formedness lemmas for this definition:

Lemma 17 (Register relation uniformity)**.**

$$\begin{aligned}&\forall \mathcal{V}, n' \leq n.\, \forall W.\, \forall reg. \\ &\quad (n, reg) \in \mathcal{R}(\mathcal{V})(W) \Rightarrow (n', reg) \in \mathcal{R}(\mathcal{V})(W)\end{aligned}$$

■

Lemma 18 (Register relation montone in worlds)**.**

$$\forall \mathcal{V}, n.\, \forall W' \sqsupseteq W.\, \forall reg.\\ (n, reg) \in \mathcal{R}(\mathcal{V})(W) \Rightarrow (n, reg) \in \mathcal{R}(\mathcal{V})(W')$$

■

Lemma 19 (Register relation non-expansive in worlds)**.**

$$\forall \mathcal{V}, n.\, \forall W, W'.\\ W \overset{n}{=} W'\\ \Rightarrow \mathcal{R}(\mathcal{V})(W) \overset{n}{=} \mathcal{R}(\mathcal{V})(W')$$

■

Lemma 20 (Register relation non-expansive in value relation)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.\, \mathcal{V} \overset{n}{=} \mathcal{V}' \Rightarrow \mathcal{R}(\mathcal{V}) \overset{n}{=} \mathcal{R}(\mathcal{V}')$$

■

2.3.3 Continuation Relation

The continuation relation is used in the definition of the expression relation. The continuation relation ensures that if you continue execution through a continuation, then it will result in a good result according to the world.

$$\mathcal{K} \quad : \quad (\text{World} \overset{mon,\, ne}{\to} \text{UPred(Word)}) \overset{ne}{\to} \text{World} \overset{mon,\, ne}{\to} \text{UPred(Word)}$$

$$\mathcal{K} \overset{def}{=} \lambda \mathcal{V}.\, \lambda W.\, \{(n, c) \mid\ (n, c) \in \mathcal{V}(W) \wedge\\ \forall W' \sqsupseteq W, n' < n.\, \forall hs :_{n'} W'.\, \forall reg, (n', reg) \in \mathcal{R}(\mathcal{V})(W').\\ (n', (reg[\text{pc} \mapsto \mathit{updatePcPerm}(c)], hs)) \in \mathcal{O}(W')\}$$

We require the continuation to be in the value relation, because it will have to be in the registers when it is invoked, so to argue that the register file is in the register-file relation, we need to know that the continuation is in the value relation.

Normally, it would just be required for the return value to be in the value relation, so the continuation is invoked with this value. Here everything that is in the register file is available when the continuation is invoked, so we do not special case on r_1 (which is the register that in the calling convention we use should contain the return value).

Well-definedness lemmas:

Lemma 21 (Continuation relation uniformity)**.**

$$\forall \mathcal{V}.\, \forall n' \leq n.\, \forall W.\, \forall c.\\ (n, c) \in \mathcal{K}(\mathcal{V})(W) \Rightarrow (n', c) \in \mathcal{K}(\mathcal{V})(W)$$

■

Lemma 22 (Continuation relation monotone in worlds)**.**

$$\begin{aligned}&\forall \mathcal{V}.\, \forall n.\, \forall W' \sqsupseteq W.\, \forall c.\\ &\quad (n, c) \in \mathcal{K}(\mathcal{V})(W) \Rightarrow (n, c) \in \mathcal{K}(\mathcal{V})(W')\end{aligned}$$

■

Lemma 23 (Continuation relation non-expansive in worlds)**.**

$$\begin{aligned}&\forall \mathcal{V}, n.\, \forall W, W'.\\ &\quad W \stackrel{n}{=} W'\\ &\qquad \Rightarrow \mathcal{K}(\mathcal{V})(W) \stackrel{n}{=} \mathcal{K}(\mathcal{V})(W')\end{aligned}$$

■

Lemma 24 (Continuation relation non-expansive in value relation)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.\, \mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow \mathcal{K}(\mathcal{V}) \stackrel{n}{=} \mathcal{K}(\mathcal{V}')$$

■

2.3.4 Expression Relation

The expression relation is defined as follows:

$$\begin{aligned}\mathcal{E} \quad &: \quad (\text{World} \stackrel{mon, ne}{\to} \text{UPred(Word)}) \stackrel{ne}{\to} \text{World} \stackrel{ne}{\to} \text{UPred(Word)}\\ \mathcal{E} &\stackrel{def}{=} \lambda \mathcal{V}.\, \lambda W.\, \{(n, pc) \mid \forall n' \leq n.\\ &\qquad \forall\, (n', reg) \in \mathcal{R}(\mathcal{V})(W).\\ &\qquad\quad \forall\, (n', c) \in \mathcal{K}(\mathcal{V})(W).\\ &\qquad\qquad \forall hs :_{n'} W.\\ &\qquad\qquad\quad (n', (reg[r_0 \mapsto c][\text{pc} \mapsto pc], hs)) \in \mathcal{O}(W)\}\end{aligned}$$

Well-definedness lemmas:

Lemma 25 (Expression relation uniformity)**.**

$$\begin{aligned}&\forall \mathcal{V}.\, \forall n' \leq n.\, \forall W.\, \forall pc.\\ &\quad (n, pc) \in \mathcal{E}(\mathcal{V})(W) \Rightarrow (n', pc) \in \mathcal{E}(\mathcal{V})(W)\end{aligned}$$

■

Lemma 26 (Expression relation non-expansive in world)**.**

$$\forall \mathcal{V}.\, \forall W_1 \stackrel{n}{=} W_2.\, \mathcal{E}(\mathcal{V})(W_1) \stackrel{n}{=} \mathcal{E}(\mathcal{V})(W_2)$$

■

Lemma 27 (Expression relation non-expansive in value relation)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.\, \mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow \mathcal{E}(\mathcal{V}) \stackrel{n}{=} \mathcal{E}(\mathcal{V}')$$

■

2.3.5 Standard Region

The following standard region is used in the definition of the value relation. Specifically, it is used in the *readCondition* and the *readWriteCondition* (to be defined next)

$$\iota_{start,end} : (\text{World} \overset{mon,\, ne}{\to} \text{UPred(Word)}) \overset{ne}{\to} \text{Region}$$
$$\iota_{base,end} \overset{def}{=} \lambda\, \mathcal{V}.\, ((base, end), =, H_{std}(\mathcal{V}))$$

$$H_{std} : (\text{World} \overset{mon,\, ne}{\to} \text{UPred(Word)}) \overset{ne}{\to} \text{State} \overset{ne}{\to} \text{Wor} \overset{mon,\, ne}{\to} \text{UPred(HeapSegment)}$$
$$H_{std}\ \mathcal{V}\ (base, end)\ \hat{W} \overset{def}{=} \left\{ (n, hs) \,\middle|\, \begin{array}{l} \text{dom}(hs) = [base, end] \wedge \\ \forall a \in [base, end].\, (n-1, hs(a)) \in \mathcal{V}(\xi\ \hat{W}) \end{array} \right\}$$

As mentioned previously, the set of states contains the "necessary" states. For the above to make sense, the set of states contains pairs of natural numbers $(base, end)$.

The well-definedness lemmas for the above is:

Lemma 28 (H_{std} is monotone in the worlds)**.**

$$\forall \mathcal{V}.\, \forall base, end.\, \forall W' \sqsupseteq W.$$
$$H_{std}\ \mathcal{V}\ (base, end)\ W' \supseteq H_{std}\ \mathcal{V}\ (base, end)\ W$$

■

Lemma 29 (H_{std} is non-expansive in the worlds)**.**

$$\forall \mathcal{V}.\, \forall base, end.\, \forall n.\, \forall W_1 \overset{n}{=} W_2.$$
$$H_{std}\ \mathcal{V}\ (base, end)\ W_1 \overset{n}{=} H_{std}\ \mathcal{V}\ (base, end)\ W$$

■

Lemma 30 (H_{std} is non-expansive in the value relation)**.**

$$\forall \mathcal{V}, \mathcal{V}'.\, \forall n.\, \mathcal{V} \overset{n}{=} \mathcal{V}' \Rightarrow$$
$$H_{std}\ \mathcal{V} \overset{n}{=} H_{std}\ \mathcal{V}'$$

■

Lemma 31 (H_{std} is uniform)**.**

$$\forall \mathcal{V}, s = (base, end), \hat{W}, n', n, hs.$$
$$n' \leq n \wedge (n, hs) \in H_{std}\ \mathcal{V}\ s\ \hat{W}$$
$$\Rightarrow (n', hs) \in H_{std}\ \mathcal{V}\ s\ \hat{W}$$

■

Lemma 32 (H_{std} is non-expansive in state)**.**

$$\begin{aligned}&\forall \mathcal{V}, (base, end), (base', end'), n.\\ &\quad (base, end) \stackrel{n}{=} (base', end')\\ &\qquad \Rightarrow H_{std}\ \mathcal{V}\ (base, end) \stackrel{n}{=} H_{std}\ \mathcal{V}\ (base', end')\end{aligned}$$

■

Lemma 33 ($\iota_{base,end}$ is non-expansive in the value relation)**.**

$$\begin{aligned}&\forall base, end.\, \forall \mathcal{V}, \mathcal{V}'.\, \forall n.\, \mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow\\ &\quad \iota_{base,end}\ \mathcal{V} \stackrel{n}{=} \iota_{base,end}\ \mathcal{V}'\end{aligned}$$

■

2.3.6 Capability Conditions

The definition of the value relation has the same conditions several times, so to define it consisely, we define the following conditions.

$$\begin{aligned}&\mathit{readCondition} : (\text{World} \stackrel{mon,ne}{\to} \text{UPred(Word)}) \stackrel{ne}{\to} (\text{Addr}^2 \times \text{World}) \stackrel{mon,ne}{\to} \text{P}^{\downarrow}(\mathbb{N})\\ &\mathit{readCondition}(\mathcal{V})(base, end, W) = \{n \mid \exists r \in \text{RegionName}.\\ &\qquad \exists [base', end'] \supseteq [base, end].\\ &\qquad\quad W(r) \stackrel{n-1}{\subseteq} \iota_{base', end'}(\mathcal{V})\}\end{aligned}$$

$$\begin{aligned}&\mathit{readWriteCondition} : (\text{World} \stackrel{mon,ne}{\to} \text{UPred(Word)}) \stackrel{ne}{\to} (\text{Addr}^2 \times \text{World}) \stackrel{mon,ne}{\to} \text{P}^{\downarrow}(\mathbb{N})\\ &\mathit{readWriteCondition}(\mathcal{V})(base, end, W) = \{n \mid \exists r \in \text{RegionName}.\\ &\qquad \exists [base', end'] \supseteq [base, end].\\ &\qquad\quad W(r) \stackrel{n-1}{=} \iota_{base', end'}(\mathcal{V})\}\end{aligned}$$

$$\begin{aligned}&\mathit{executeCondition} : (\text{World} \stackrel{mon,ne}{\to} \text{UPred(Word)}) \stackrel{ne}{\to} (\text{Addr}^2 \times \text{Perm} \times \text{World}) \stackrel{mon,ne}{\to} \text{P}^{\downarrow}(\mathbb{N})\\ &\mathit{executeCondition}(\mathcal{V})(base, end, perm, W) = \{n \mid \forall n' < n.\, \forall W' \sqsupseteq W.\\ &\qquad \forall a \in [base, end].\\ &\qquad\quad (n', (perm, base, end, a)) \in \mathcal{E}(\mathcal{V})(W')\}\end{aligned}$$

$$\begin{aligned}&\mathit{entryCondition} : (\text{World} \stackrel{mon,ne}{\to} \text{UPred(Word)}) \stackrel{ne}{\to} (\text{Addr}^3 \times \text{World}) \stackrel{mon,ne}{\to} \text{P}^{\downarrow}(\mathbb{N})\\ &\mathit{entryCondition}(\mathcal{V})(base, end, a, W) = \{n \mid \forall n' < n.\, \forall W' \sqsupseteq W.\\ &\qquad (n', (\text{rx}, base, end, a)) \in \mathcal{E}(\mathcal{V})(W')\}\end{aligned}$$

The following lemmas show that the above conditions are well-defined:

Lemma 34 (Read condition downwards-closed)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, n', W, base, end.\\ &\quad n \in readCondition(\mathcal{V})(base, end, W) \wedge\\ &\quad n' \leq n\\ &\qquad \Rightarrow n' \in readCondition(\mathcal{V})(base, end, W)\end{aligned}$$

■

Lemma 35 (Read condition monotone in world)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, W, W', base, end, base', end'.\\ &\quad (base', end', W') \sqsupseteq (base, end, W)\\ &\qquad \Rightarrow readCondition(\mathcal{V})(base', end', W') \supseteq readCondition(\mathcal{V})(base, end, W)\end{aligned}$$

■

Lemma 36 (Read condition non-expansive in worlds)**.**

$$\begin{aligned}&\forall \mathcal{V}, base, end, base', end', n, W, W'.\\ &\quad (base, end, W) \stackrel{n}{=} (base', end', W') \Rightarrow\\ &\qquad readCondition(\mathcal{V})(base, end, W) \stackrel{n}{=} readCondition(\mathcal{V})(base', end', W')\end{aligned}$$

■

Lemma 37 (Read-write condition uniformity)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, n', W, base, end.\\ &\quad n \in readWriteCondition(\mathcal{V})(base, end, W) \wedge\\ &\quad n' \leq n\\ &\qquad \Rightarrow n' \in readWriteCondition(\mathcal{V})(base, end, W)\end{aligned}$$

■

Lemma 38 (Read-write condition monotone in world)**.**

$$\begin{aligned}&\forall \mathcal{V}, W, W', base, end, base', end'.\\ &\quad (base', end', W') \sqsupseteq (base, end, W)\\ &\qquad \Rightarrow readWriteCondition(\mathcal{V})(base', end', W') \supseteq readWriteCondition(\mathcal{V})(base, end, W)\end{aligned}$$

■

Lemma 39 (Read-write condition non-expansive in world)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, W, W', base, end, base', end'.\\ &\quad (base', end', W') \stackrel{n}{=} (base, end, W)\\ &\qquad \Rightarrow readWriteCondition(\mathcal{V})(base, end, W) \stackrel{n}{=} readWriteCondition(\mathcal{V})(base', end', W')\end{aligned}$$

■

Lemma 40 (Execute condition downwards-closed)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, n', W, base, end, perm.\\ &\quad n \in executeCondition(\mathcal{V})(base, end, perm, W) \wedge\\ &\quad n' \leq n\\ &\qquad \Rightarrow n' \in executeCondition(\mathcal{V})(base, end, perm, W)\end{aligned}$$

■

Lemma 41 (Execute condition monotone in world)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, W, W', base, end, perm, base', end', perm'.\\ &\quad (base', end', perm', W') \sqsupseteq (base, end, perm, W)\\ &\qquad \Rightarrow executeCondition(\mathcal{V})(base', end', perm', W') \supseteq executeCondition(\mathcal{V})(base, end, perm, W)\end{aligned}$$

■

Lemma 42 (Execute condition non-expansive in worlds)**.**

$$\begin{aligned}&\forall \mathcal{V}, W, W', n, base, end, perm, base', end', perm'.\\ &\quad (base, end, perm, W) \stackrel{n}{=} (base', end', perm', W') \Rightarrow\\ &\qquad \Rightarrow executeCondition(\mathcal{V})(base, end, perm, W) \stackrel{n}{=} executeCondition(\mathcal{V})(base', end', perm', W')\end{aligned}$$

■

Lemma 43 (Entry condition downwards-closed)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, n', W, base, end, a.\\ &\quad n \in entryCondition(\mathcal{V})(base, end, a, W) \wedge\\ &\quad n' \leq n\\ &\qquad \Rightarrow n' \in entryCondition(\mathcal{V})(base, end, a, W)\end{aligned}$$

■

Lemma 44 (Entry condition monotone in world)**.**

$$\begin{aligned}&\forall \mathcal{V}, W, W', base, end, a, base', end', a'.\\ &\quad (base', end', a', W') \sqsupseteq (base, end, a, W)\\ &\qquad \Rightarrow entryCondition(\mathcal{V})(base', end', perm', W') \supseteq entryCondition(\mathcal{V})(base, end, perm, W)\end{aligned}$$

■

Lemma 45 (Entry condition non-expansive in worlds)**.**

$$\begin{aligned}&\forall \mathcal{V}, n, W, W', base, end, a, base', end', a'.\\ &\quad (base, end, perm, W) \stackrel{n}{=} (base', end', perm', W') \Rightarrow\\ &\qquad \Rightarrow executeCondition(\mathcal{V})(base, end, a, W) \stackrel{n}{=} executeCondition(\mathcal{V})(base', end', a', W')\end{aligned}$$

■

Finally, we need to show that all the conditions are non-expansive, but we later want to use Banach's fixed-point theorem to define the value relation. For this we will need that the above conditions are contractive, and if they are contractive, then they are also non-expansive, so we show that each of the conditions are contractive:

Lemma 46 (Read condition contractive)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.$$
$$\mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow \mathit{readCondition}(\mathcal{V}) \stackrel{n+1}{=} \mathit{readCondition}(\mathcal{V}')$$

■

Lemma 47 (Write condition contractive)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.$$
$$\mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow \mathit{readWriteCondition}(\mathcal{V}) \stackrel{n+1}{=} \mathit{readWriteCondition}(\mathcal{V}')$$

■

Lemma 48 (Execute condition contractive)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.$$
$$\mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow \mathit{executeCondition}(\mathcal{V}) \stackrel{n+1}{=} \mathit{executeCondition}(\mathcal{V}')$$

■

Lemma 49 (Entry condition contractive)**.**

$$\forall \mathcal{V}, \mathcal{V}', n.$$
$$\mathcal{V} \stackrel{n}{=} \mathcal{V}' \Rightarrow \mathit{entryCondition}(\mathcal{V}) \stackrel{n+1}{=} \mathit{entryCondition}(\mathcal{V}')$$

■

Lemma 50 (Write condition implies read condition)**.**

$$\forall n, W, \mathit{base}, \mathit{end}.$$
$$\mathit{readWriteCondition}(n, W, \mathit{base}, \mathit{end}) \Rightarrow \mathit{readCondition}(n, W, \mathit{base}, \mathit{end})$$

■

2.3.7 Value Relation

The value relation, is defined as follows:

$$\mathcal{V} : (\text{World} \overset{mon,\, ne}{\to} \text{UPred}(Words)) \overset{ne}{\to} \text{World} \overset{mon,\, ne}{\to} \text{UPred}(\text{Word})$$

$$\begin{aligned}
\mathcal{V} \overset{def}{=} \lambda\, V.\, \lambda\, W.\ & \{(n, i) \mid i \in \mathbb{Z}\} \cup \\
& \{(n, (\text{o}, \mathit{base}, \mathit{end}, a))\} \cup \\
& \{(n, (\text{ro}, \mathit{base}, \mathit{end}, a)) \mid n \in \mathit{readCondition}(V)(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(n, (\text{rw}, \mathit{base}, \mathit{end}, a)) \mid n \in \mathit{readWriteCondition}(V)(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(n, (\text{rx}, \mathit{base}, \mathit{end}, a)) \mid \\
& \quad n \in \mathit{readCondition}(V)(\mathit{base}, \mathit{end}, W) \wedge \\
& \quad n \in \mathit{executeCondition}(V)(\mathit{base}, \mathit{end}, \text{rx}, W)\} \cup \\
& \{(n, (\text{e}, \mathit{base}, \mathit{end}, a)) \mid n \in \mathit{entryCondition}(V)(\mathit{base}, \mathit{end}, a, W)\} \cup \\
& \{(n, (\text{rwx}, \mathit{base}, \mathit{end}, a)) \mid \\
& \quad n \in \mathit{readWriteCondition}(V)(\mathit{base}, \mathit{end}, W) \wedge \\
& \quad n \in \mathit{executeCondition}(V)(\mathit{base}, \mathit{end}, \text{rx}, W) \wedge \\
& \quad n \in \mathit{executeCondition}(V)(\mathit{base}, \mathit{end}, \text{rwx}, W)\}
\end{aligned}$$

Lemma 51 (Value relation uniformity)**.**

$$\begin{aligned}
& \forall V, W, n, n'. \\
& \quad n' \leq n \wedge n \in \mathcal{V}(V)(W) \\
& \qquad \Rightarrow n' \in \mathcal{V}(V)(W)
\end{aligned}$$

■

Lemma 52 (Value relation non-expansive in V)**.**

$$\begin{aligned}
& \forall V, V', n. \\
& \quad V \overset{n}{=} V' \Rightarrow \mathcal{V}(V) \overset{n}{=} \mathcal{V}(V')
\end{aligned}$$

■

Lemma 53 (Value relation monotone in worlds)**.**

$$\begin{aligned}
& \forall V, W, W'. \\
& W' \sqsupseteq W \Rightarrow \mathcal{V}(V)(W') \supseteq \mathcal{V}(V)(W)
\end{aligned}$$

■

Lemma 54 (Value relation non-expansive in worlds)**.**

$$\begin{aligned}
& \forall V, n, W, W'. \\
& W \overset{n}{=} W' \Rightarrow \mathcal{V}(V)(W) \overset{n}{=} \mathcal{V}(V)(W')
\end{aligned}$$

■

Lemma 55 (Value relation contractive)**.**

$$\forall V, V', n.$$
$$V \overset{n}{=} V' \Rightarrow \mathcal{V}(V) \overset{n+1}{=} \mathcal{V}(V')$$

■

Seeing as Lemma 55 gives us that $\mathcal{V}$ is contractive, the Banach fixed-point theorem gives us a fixed-point for $\mathcal{V}$. We will use this fixed point in the next section to define the logical relation we use in practice.

2.4 Logical Relation Definitions

LS: This section introduces the definitions, we started working with, so it is here so we don't have to rewrite everything.

Here we state the definitions, we will use for pratical purposes. Assume V is the fixed point given by Banach's fixed point theorem as described in Section 2.3.7. In general the definitions here will be the same as in Section 2.3, but but applied to V when applicable. As they are essentially the same, we will reuse the names.

$$\mathcal{R} \overset{def}{=} \mathcal{R}\ V$$
$$\mathcal{K} \overset{def}{=} \mathcal{K}\ V$$
$$\mathcal{E} \overset{def}{=} \mathcal{E}\ V$$
$$\mathcal{V} \overset{def}{=} \mathcal{V}\ V$$

$$\iota_{base,end} \overset{def}{=} \iota_{base,end}\ V$$
$$H_{std} \overset{def}{=} H_{std}\ V$$

For natural number n, adresses $base$, end, and a, permission $perm$, and world W, we define the capability conditions as follows

$$readCondition(n, W, base, end) \text{ iff } n \in readCondition(V)(base, end, W)$$
$$readWriteCondition(n, W, base, end) \text{ iff } n \in readWriteCondition(V)(base, end, W)$$
$$executeCondition(n, W, base, end, perm) \text{ iff } n \in executeCondition(V)(base, end, p, W)$$
$$entryCondition(n, W, base, end, a) \text{ iff } n \in entryCondition(V)(base, end, a, W)$$

The well-formedness lemmas of course still hold for this new presentation.

2.5 Further Standard Regions

We have already defined the standard region $\iota_{base,end}$ which ensures that all the words in the region are in the value relation. See Section 2.3.5. We here define

another usefull region, namely the one that requires all the words in the region to be the constant 0.

$$\iota^0_{base,end} \ : \ \text{Region}$$

$$\iota^0_{base,end} \stackrel{def}{=} ((base, end), =, H^0_{std})$$

$$H^0_{std}\ (base, end)\ W \stackrel{def}{=} \left\{ (n, hs) \middle| \begin{array}{l} \mathrm{dom}(hs) = [base, end] \wedge \\ \forall a \in \mathrm{dom}(hs).\ hs(a) = 0 \end{array} \right\}$$

Lemma 56 ($\iota^0_{base,end}$ is well-defined)**.**

- H^0_{std} *is non-expansive.*
- *For all adresses base and end,* $H^0_{std}\ (base, end)$ *is monotone and non-expansive.*
- $=$ *is a reflexive and transitive relation.*

■

2.6 Fundamental Theorem of Logical Relations

Lemma 57 (Fundamental theorem of logical relations (FTLR))**.**
For any $n \in \mathbb{N}$, $W \in$ World, $p \in$ Perm, *and addresses base, end, and a, if* $perm = \mathrm{rwx}$ *and* $readWriteCondition(n, W, base, end)$ *or* $perm = \mathrm{rx}$ *and* $readCondition(n, W, base, end)$*, then* $(n, (perm, base, end, a)) \in \mathcal{E}(W)$. ■

LS: With the current definition of the expression relation, an entry pointer will never be in the expression relation, but I guess we do not want that restriction. Say the adv capability in the ticket dispenser lemma, we want that to be an entry capability.
DD: I don't understand the above remark: why should an entry pointer be in the expression relation, since it shouldn't be able to reach the pc?
LS: In the ticket dispenser lemma, we need to make some assumptions about the adversary to be able to use the FTLR. What should these assumptions be? So far I have the assumption that it is all code (no capabilities), but I do not see how I conclude the write condition from this.
DD: Something I find interesting here, is that the FTLR can be interpreted to say that x permissions have no real security value: if you have an arbitrary r or rw capability that is valid, then the FTLR says that it remains valid if we change it to rx or rwx (respectively).

Proof. The proof is by induction on n.

If $n = 0$, then it all boils down to an assumption where $(reg, hs \uplus heap_f) \rightarrow_0 (halted, heap')$ for some reg, hs, $heap_f$, and $heap'$ which is not possible in our operational semantics.

If $n > 0$, then we need to consider a lot of cases, but first we make a lot of preliminary assumptions:
Let the world W, permission p, and addresses b, e, and a be given. Assume

$$p = \text{rwx and } \mathit{readWriteCondition}(n, W, b, e) \text{ or} \tag{1}$$
$$p = \text{ro and } \mathit{readCondition}(n, W, b, e) \tag{2}$$

We need to show

$$(n, (p, b, e, a)) \in \mathcal{E}(W)$$

Let n', reg, c, hs be given where

$$n' \leq n \tag{3}$$
$$(n', \mathit{reg}) \in \mathcal{R}(W) \tag{4}$$
$$(n', c) \in \mathcal{K}(W) \tag{5}$$
$$\mathit{hs} :_{n'} W \tag{6}$$

For $\mathit{pc} = (p, b, e, a)$ we need to show

$$(n', (\mathit{reg}[r_0 \mapsto c, \mathrm{pc} \mapsto \mathit{pc}], \mathit{hs})) \in \mathcal{O}(W)$$

For convenience, we will define $\mathit{reg}' = \mathit{reg}[r_0 \mapsto c, \mathrm{pc} \mapsto \mathit{pc}]$. To show the above, let h_f, h', and i be given where $i \leq n'$ and assume

$$\Phi \rightarrow_i (\mathit{halted}, h') \tag{7}$$

for $\Phi = (\mathit{reg}', \mathit{hs} \uplus h_f)$ then we need to show

$$\exists W' \sqsupseteq W.\, \exists \mathit{hs}'.$$
$$h' = \mathit{hs}' \uplus h_f \text{ and} \tag{8}$$
$$\mathit{hs}' :_{n'-i} W' \tag{9}$$

If $i = 0$, then we are trivially done as execution 7 is not possible. If $i > 0$, then execution 7 takes at least one step, consider the first step of this execution. By the operational semantics, the first step is:

$$\Phi \rightarrow [\![\mathit{decode}(h(a))]\!](\Phi)$$

where $h = \mathit{hs} \uplus h_f$. We may further assume

$$\mathit{reg}'(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, a)$$
$$\mathit{base} \leq a \leq \mathit{end}$$
$$\mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}$$

For convenience define $\mathit{instr} = \mathit{decode}(h(a))$. We proceed by case on instr.

Case $\mathit{instr} = \texttt{fail}$:
This case is trivially true as it is not possible to step to a halting state from a

failed step in any number of steps. By assumption 7, the execution ends in a halting state.

Case $instr = \texttt{halt}$:

Here we have:

$$\Phi \rightarrow (halted, (reg', h))$$

Pick $hs' = hs$ and $W' = W$. $h' = hs \uplus h_f$, so condition 8 is satisfied. Using $hs :_{n'} W$ and the uniformity of heap satisfaction, we get $hs :_{n'-i} W'$ as $n' > n' - i$.

Case $instr = \texttt{jmp}\ r$:

Here the first execution step is:

$$\Phi \rightarrow (reg[r_0 \mapsto c, \mathrm{pc} \mapsto updatePcPerm(reg'(r))], h)$$

We now consider four cases

Case $reg'(r) = (\mathrm{rwx}, b', e', a')$**:** In this case, we want to appeal to the induction hypothesis. To do so, we need to argue that we have $readWriteCondition(n' - 1, b', e', a')$. If $r = \mathrm{pc}$, then we have this by assumption 1 and uniformity. If $r \neq \mathrm{pc}$, then if $r = r_0$, then by assumption 5, we have $(n', c) \in \mathcal{K}(W)$ which entails $(n', c) \in \mathcal{V}(W)$ which together with uniformity gives the *readWriteCondition* due to the permission. Otherwise, the same folloes from assumption 4 and uniformity.

By IH we get

$$(n' - 1, (\mathrm{rwx}, b', e', a')) \in \mathcal{E}(W)$$

using $n' - 1$, reg, c, and hs and assumptions 3 - 6 and the appropriate uniformity lemmas, we can use the above to get

$$(n' - 1, reg[r_0 \mapsto c, \mathrm{pc} \mapsto (\mathrm{rwx}, b', e', a')]) \in \mathcal{O}(W)$$

This is the register file of the configuration after the first step of execution 7, so we know:

$$(reg[r_0 \mapsto c, \mathrm{pc} \mapsto (\mathrm{rwx}, b', e', a')], h) \rightarrow_{i-1} (halted, h')$$

using this and that the register file is in the observation relation, we get a $W' \sqsupseteq W$ and hs' such that

$$h' = hs' \uplus h_f$$
$$hs' :_{n'-1-(i-1)} W'$$

W' and hs' satisfy exactly the properties, we need, so we are done.

Case $reg'(r) = (\mathrm{rx}, b', e', a')$**:** This case goes just like the one above with the exception that we need $readCondition(n' - 1, b', e', a')$ which we get in a similar fashion.

Case $reg'(r) = (\mathrm{e}, b', e', a')$**:** In this case, we do not appeal to the induction hypothesis, but rather to the *entryCondition*. If $r = \mathrm{pc}$, then we have a contradiction with either 1 or 2, so it is safe to assume $r \neq \mathrm{pc}$. If $r = r_0$, then we rely on assumption 5 to get $(n', \mathrm{e}, b', e', a') \in \mathcal{V}(W)$. In the case where $r \neq r_0$, we rely on assumption 4 to get $(n', \mathrm{e}, b', e', a') \in \mathcal{V}(W)$, so we will have $\mathit{entryCondition}(n', W, b', e', a')$. Using the *entryCondition*, we get $(n' - 1, (\mathrm{rx}, b', e', a')) \in \mathcal{E}(W)$. From here on, the argument follows like the cases, we have seen. One notable thing is that in this case is that $\mathit{updatePcPerm}(reg[r_0 \mapsto c, \mathrm{pc} \mapsto pc](r))$ updates the permission to an execute permission rx which makes everything align up nicely as in the previous cases.

Otherwise in this case, we consider the next execution step which goes to *failed* as the pc is not a well-formed execution. We have, however, assumed that a halting configuration is reached eventually, so a contradiction is reached.

Case $instr = \texttt{jnz}$:

LS: TODO, I have not checked the details as I expect it to be like the `jmp` case.

Case $instr = \texttt{load}\ r_1\ r_2$:

Assume for some p', b', e', a', and w that

$$\begin{aligned} &reg'(r_1) = (p', b', e', a') \\ &\mathit{readAllowed}(p') \\ &\mathit{withinBounds}((p', b', e', a')) \\ &w = h(a) \end{aligned} \tag{10}$$

if the above where not the case, then by the semantics, we would step to a fail configuration which by assumption is not possible. We consider two cases for r_1.

$r_1 = \mathrm{pc}$ The first step of the execution is

$$\Phi \rightarrow \mathit{updatePc}(\Phi[\mathrm{reg.pc} \mapsto w])$$

If w is not a capability, then

$$\mathit{updatePc}(\Phi[\mathrm{reg.pc} \mapsto w]) = \mathit{failed}$$

and we are done. Assume $w = (p', b', e', a')$, then

$$\mathit{updatePc}(\Phi[\mathrm{reg.pc} \mapsto w]) = \Phi[\mathrm{reg.pc} \mapsto (p', b', e', a' + 1)]$$

Using execution 7, we can conclude that $p' \in \{\mathrm{rwx}, \mathrm{rx}\}$ as the above configuration otherwise would step to *failed*. From 10

$r_1 \neq \mathrm{pc}$

Case $instr =$ `store` :

Case $instr =$ `move` :

Case $instr =$ `lea` :

Case $instr =$ `restrict` :

Case $instr =$ `plus` :

□

3 Other examples and applications

This section contains some ideas about other examples and applications than the ticket dispenser example.

3.1 Stack and return pointer handling without OS involvement using local capabilities

The idea of this example would be to work out and prove a calling convention that enforces well-bracketed control flow and encapsulation of local variables using CHERI's local capabilities.

When one function invokes another function, the essential idea is that:

- Stack pointer is passed as a local and store-local capability.
- Return pointer is passed as a local capability.

Since local pointers cannot leave the registers except into regions for which a store-local capability is available, this basic idea seems to enforce a number of useful properties: well-bracketedness of control flow and encapsulation of private state stored on the stack. On the other hand, it also seems to validate the standard C treatment of the stack: the stack can be reused after a function returns, even between distrusting parties. However, safety/security of this design is very non-trivial and seems to rely on some non-trivial reasoning:

Only stack is store-local? A critical assumption is that adversary code has no way to *store* local capabilities except on the stack. The reason that it is fine to store local capabilities on the stack is that the adversary only has a *local* capability to the stack and cannot usefully store that capability anywhere. However, this means that we need to rely on the runtime system of our programming language to be careful when handing out store-local capabilities: only the libc startup code should initialize the stack as store-local and malloc should *not* produce them. This basically means that the libc initialization code (or whatever component produces the initial stack pointer) is part of our TCB.

Requirement for clearing the stack Imagine the following trusted C function:

```
void myfunction(){
  advfunction1();
  advfunction2();
}
```

where advfunction1() and advfunction2() are adversary functions. In the standard C treatment of the stack, advfunction2() would get the same stack pointer as advfunction1(). This is supposed to be safe since advfunction1() cannot have kept capabilities for the stack after its execution. But what if we require that the two functions have no way of communicating with each other? Concretely, advfunction1() has access to some secrets that must not be leaked to advfunction2(). How can we prevent advfunction1() from storing the secret somewhere on the stack and relying on advfunction2() from receiving the same stack pointer where it can read the secret? The most obvious solution seems to be that we should fully clear the stack (overwrite it with zeros) after the return of any adversary function, but this could cause an important overhead. Perhaps the processor should accomodate this with a special instruction that can zero the entire array that a capability points to?

What do return pointers look like? An important question is what return pointers look like? Since we want to protect the caller from the callee, it's important that the return pointer is opaque, i.e. an entry pointer. The entry pointer will point to a closure that contains the next instruction to execute, as well as the previous stack pointer. But since stack pointers are local, this means that the return pointer closure should be stored in a region of memory for which we have store-local permission, i.e. on the stack. This means we need the following in our calling convention: before invoking a function, we push the stack pointer and the instruction pointer after invocation on the stack, we construct a return pointer by copying the stack pointer, limiting it to these two entries and making it an entry pointer. Then we shrink the stack pointer to the unused part of the stack and jump.

Only one-way protection in higher-order settings? Another important point is that, in a sense, local capabilities provide only one-way protection: the caller is protected from the callee but not vice-versa. Concretely: when invoking a function with some arguments marked as local, the caller is guaranteed that the callee will not have been able to store the capabilities anywhere (except perhaps on the stack, see above). However, the callee seems to have more limited guarantees: Particularly, the caller may have kept its own stack capability and this stack capability may (and typically will) also cover the part of the stack that is "owned" by the callee. In this sense, the guarantees are more limited than in a linear language.

So what does this mean? In a first-order language, this is all fine, but what if we are in a higher-order language. Imagine the following (in some ML-like language):

```
let f = fun callback =>
          let ... in
          let ret = callback() in
          ...
//adversary top function
let advtop = f( (fun y => ...) )
```

Our trusted function f is invoked by the adversary (from function advtop()) and wants to invoke an untrusted callback received from the adversary. When invoking the closure, we don't want it to be able to access f's local variables which it has stored on the stack. To achieve this, we only give it a stack pointer that covers the part of the stack that is unused by f. However, the callback may be implemented as an entry pointer that carries capabilities, particularly the capability to advtop's stack pointer, which includes the part of the stack that is now used by f and contains f's local variables.

So how do we deal with this? Perhaps we should use the fact that this is only possible when f's callback argument is allocated to some part of the memory to which advtop has store-local permissions (since the callback contains a reference to the stack to which advtop only has a local capability). I see basically three ways to do this, all based on the idea of enforcing that the callback should be constructed in a part of memory for which no store-local permissions are available:

- One way to exclude the scenario is to require that callbacks are provided as non-local capabilities. The downside of this is that local callbacks can be useful for the caller to prevent the callee from storing them.
- Another way to exclude the scenario is to require that the stack is allocated in a fixed part of the address space and to check that callbacks point outside of this region before invoking them.
- Perhaps we should require that store-local permissions cannot be removed from a capability and simply require that callback pointers do not have store-local set. Perhaps we can allow store-local permissions to be given up, but only if the corresponding part of memory is fully zeroed in the process (or at least all local capabilities stored in the region).

3.2 A result to prove...

The simplest thing that comes to mind as a formal result for all of the above is to look at a concrete program that clearly relies on properties like well-bracketed control flow and encapsulation of local variables and prove it correct. As a concrete example: we might show an assembly program that corresponds to the

following (a higher-order program that crosses trust boundaries and relies on local variable encapsulation and well-bracketed control flow):

```
let trustedCode = fun adversary =>
                    let x = ref 0 in
                    let callback = fun adv2 =>
                      x := !x + 1;
                      let y = ref (!x) in
                      adv2 unit;
                      assert (!x == !y);
                      x := !x - 1)
                    let _ = adversary callback
                    assert (!x == 0)
```

LS: I have inserted some line breaks for readability. I am not sure what is going on here (the parenthesis after y is unmatched.)

4 Related reading

This is a list of related work that might be interesting to read in the context of this project.

4.1 Capability machines

4.1.1 M-Machine

More than 20 years ago, Carter et al. [1994] have described the use of capabilities in the M-Machine. They do seem to have a reference for the instruction set after all [Dally et al., 1995]; it seems like the server was just temporarily down when we were looking for this the first time...

4.1.2 CHERI

The CHERI processor is a much more recent capability machine, described by Woodruff et al. [2014], Watson et al. [2015].

Another result of this project is also CheriBSD: an adaptation of FreeBSD to the CHERI processor.[2] It is not separately described in a published paper, but mentioned in the papers cited above and in some tech reports (see url). This work includes a pure-capability ABI that could provide some interesting examples.

The CHERI team also has a webpage with all of their CHERI-related publications (including TRs and such)[3].

[2] http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html
[3] http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

4.2 Logical Relations

Some papers on logical relations that are relevant for this work are the following:

Hur and Dreyer [2011] describe a logical relation between ML and a (standard) assembly language for expressing compiler correctness. Relevant because they target an assembly language, and they use biorthogonality.

Dreyer et al. [2010] describe a logical relation for a ML-like language and use public/private transitions to reason about well-bracketed control flow. Relevant because we are considering to cover an example of enforcing well-bracketed control flow in a capability machine.

Devriese et al. [2016] describe a logical relation for a JavaScript-like language with object capabilities. Relevant because it treats object capabilities, albeit in a JavaScript-like lambda calculus.

References

Lars Birkedal and Ale Bizjak. A Taste of Categorical Logic tutorial notes. `http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf`, 2014.

Lars Birkedal, Kristian Stvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411(47):4102 – 4122, 2010. ISSN 0304-3975.

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. ACM. ISBN 0-89791-660-3. doi: 10.1145/195473.195579. URL `http://doi.acm.org/10.1145/195473.195579`.

William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The m-machine instruction set reference manual v1.55. Technical Report Memo 59, CVA, Stanford, 1995. URL `http://cva.stanford.edu/publications/1997/isa-1.55.ps.Z`.

Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities using logical relations and effect parametricity. In *IEEE European Symposium on Security and Privacy*. IEEE, 2016.

Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 143–156, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863566. URL `http://doi.acm.org/10.1145/1863543.1863566`.

Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ml and assembly. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146. ACM, 2011. doi: 10.1145/1926385.1926402.

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9.

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *International Symposium on Computer Architecuture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

A Proofs of Lemmas in Section 2

A.1 Definitions and Proofs related to Section 2.1

Definition 12 (Ordering of Cauchy chains)**.** *For Cauchy chains* $\{a_n\}_n$ *and* $\{b_n\}_n$*,*

$$\{a_n\}_n \geq \{b_n\}_n \qquad \textit{iff} \qquad \forall n.\, a_n \geq b_n$$

■

Definition 13 (Limit of Cauchy chains)**.** *For a Cauchy chain* $\{a_n\}_n$ *in the c.o.f.e.* $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$*, we say that* a *is a limit (denoted* $\lim\{a_n\}_n$*) of this chain iff it satisfies*

$$\forall n.\, \exists N.\, \forall m \geq N.\, a \overset{n}{=} a_m$$

■

Definition 14 (Set c.o.f.e. Construction)**.** *Given a set* X*, then* $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ *is a c.o.f.e. where all the equivalences are equiality.* ■

Definition 15 (Function c.o.f.e. Construction)**.** *Given c.o.f.e.'s* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ *and* $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ *define the c.o.f.e.* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right) \to \left(X, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ *as*

$$\left(\{f : X \to Y \mid f \textit{ non-expansive}\}, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$$

where the equivalence is defined as

$$f \overset{n}{=} g \qquad \textit{iff} \qquad \forall x \in X.\, f(x) \overset{n}{=}_Y g(x)$$

■

Proof of Lemma 2. The c.o.f.e. is a standard construction for tuples, so we won't show that it in fact is a c.o.f.e. here. That $\sqsupseteq$ is reflexive and transitive follows trivially from Φ being reflexive and transitive. It remains to show that $\sqsupseteq$ preserves limits.

Assume $\{a_n\}_n \geq \{b_n\}_n$.From this it follows that for all $m \in \mathbb{N}$:

$$\begin{aligned} \Phi_{b_m} &= \Phi_{a_m} \\ H_{b_m} &= H_{a_m} \\ (s_{a_m}, s_{b_m}) &\in \Phi_{b_m} \end{aligned}$$

Say $\lim a_{nn} = (s_a, \Phi_a, H_a)$ and $\lim b_{nn} = (s_b, \Phi_b, H_b)$. We need to show

$$\begin{aligned} \Phi_b &= \Phi_a \\ H_b &= H_a \\ (s_a, s_b) &\in \Phi_b \end{aligned}$$

To show $H_b = H_a$ by one of the c.o.f.e. properties SFTS $\forall n.\, H_b \overset{n}{=} H_a$. Given n use that a is a limit to get N s.t.

$$\forall m \geq N.\, H_{a_m} \overset{n}{=} H_a$$

Likewise for b we get a similar M. Take some $m \geq \max\{M, N\}$, using $H_{b_m} = H_{a_m}$, we get:

$$H_a = H_{a_m} = H_{b_m} = H_b$$

For $\Phi_b = \Phi_a$ we do something similar except, we just pick $n = 1$ as the n-equality in this c.o.f.e. is equality. Same approach is used for $(s_a, s_b) \in \Phi_b$ $\square$

Proof of Lemma 3.

$$K(f)(\phi) = \lambda n.\, f(\phi(n))$$

The above is defined exactly on the $n \in \mathbb{N}$ where ϕ is defined. ϕ is a finite partial function, so the above is as well. $\square$

Proof of Lemma 4. Given g, H, st, w and w'. Assume $w' \sqsupseteq w$. Show

$$\begin{aligned} (G(g))(\phi)(st)(w') &\sqsupseteq (G(g))(\phi)(st)(w) \\ &\Uparrow \qquad (11) \\ H(st)(g(w')) &\sqsupseteq H(st)(g(w)) \\ &\Uparrow \qquad (12) \\ g(w') &\sqsupseteq g(w) \\ &\Uparrow \qquad (13) \\ w' &\sqsupseteq w \end{aligned}$$

(11) expands the definition of G. (12) is due to $H(st)$ being monotone. (13) is due to g being a morphism in $\mathbb{C}^{op}$ and they are monotone function. $\square$

Proof of Lemma 5. Given n, g, H, st, w and w'. Assume $w' \overset{n}{=} w$. Show

$$\begin{array}{c}(G(g))(\phi)(st)(w') \overset{n}{=} (G(g))(\phi)(st)(w) \\ \Uparrow \\ H(st)(g(w')) \overset{n}{=} H(st)(g(w))\end{array} \tag{14}$$

We use that $H = H$ implies $H \overset{n}{=} H$ which means

$$\forall st, st'.\, st \overset{n}{=} st' \Rightarrow H(st) \overset{n}{=} H(st')$$

This in turn implies

$$\forall y, y' . . y \overset{n}{=} y' \Rightarrow H(st)(y) \overset{n}{=} H(st)(y')$$

Using our assumption, about the worlds, we are done. □

Proof of Lemma 6. $K(f) : K(X) \to K(Y)$ is *monotone* for $f : X \overset{mon,\, ne}{\to} Y$: Assume $\phi \sqsupseteq \phi'$. Show

$$(K(f))(\phi) \sqsupseteq (K(f))(\phi')$$

By def on ordering for functions SFTS

$$\forall n.\, (K(f))(\phi)(n) \sqsupseteq (K(f))(\phi')(n)$$

let n be givenm show

$$f(\phi'(n)) \sqsupseteq f(\phi(n))$$

f mon, so SFTS

$$\phi'(n) \sqsupseteq \phi(n)$$

which follows from the assumption $\phi' \sqsupseteq \phi$.

$K(f) : K(X) \to K(Y)$ is *non-expansive* for $f : X \overset{mon,\, ne}{\to} Y$: Assume $\phi \overset{n}{=} \phi'$. Show

$$(K(f))(\phi) \overset{n}{=} (K(f))(\phi')$$

Let k be given, SFTS

$$(K(f))(\phi)(k) \overset{n}{=} (K(f))(\phi')(k)$$

which unfolds to

$$f(\phi(k)) \overset{n}{=} f(\phi'(k))$$

as f is non-expansive SFTS

$$\phi(k) \overset{n}{=} \phi'(k)$$

which follows from assumption.

K preserves composition, show

$$K(f \circ g) = K(f) \circ K(g)$$

Given ϕ and n show

$$\begin{aligned}(K(f) \circ K(g))(\phi)(n) &= K(f)(K(g)(\phi))(n) \\ &= f(K(g)(\phi)(n)) \\ &= f(g(\phi(n))) \\ &= (f \circ g)(\phi(n)) \\ &= K(f \circ g)(\phi)(n)\end{aligned}$$

K preserves identity, show

$$K(id) = id$$

Let ϕ and n be given.

$$\begin{aligned}K(id) &= \lambda\phi.\, \lambda n.\, id(\phi(n)) \\ &= \lambda\phi.\, \lambda n.\, \phi(n) \\ &= \lambda\phi.\, \phi \\ &= id\end{aligned}$$

□

Proof of Lemma 7. $R(f) : R(X) \to R(Y)$ is *monotone* for $f : X \overset{ne}{\to} Y$. Given aformentioned f. Further let $(s_2, \Phi_2, H_2) \sqsupseteq (s_1, \Phi_1, H_1)$ be given.

$$\begin{array}{c} R(f)(s_2, \Phi_2, H_2) = (s_2, \Phi_2, H_2(f)) \sqsupseteq (s_1, \Phi_1, H_1(f)) = R(f)(s_1, \Phi_1, H_1) \\ \Uparrow \\ \Phi_2 = \Phi_1 \text{ and } H_1(f) = H_2(f) \text{ and } (s_1, s_2) \in \Phi_2 \end{array}$$

From our assumption, we have $H_2 = H_1$ from which it follows $H_1(f) = H_2(f)$. The remaining conditions also follow directly from the assumption.

$R(f) : R(X) \to R(Y)$ is *non-expansive* for $f : X \overset{ne}{\to} Y$. Let $(s_2, \Phi_2, H_2) \overset{n}{=} (s_1, \Phi_1, H_1)$ be given

$$R(f)(s_2, \Phi_2, H_2) = (s_2, \Phi_2, H_2(f)) \overset{n}{=} (s_1, \Phi_1, H_1(f)) = R(f)(s_1, \Phi_1, H_1)$$

$H_1(f) \overset{n}{=} H_2(f)$ follows from the definition of n-equality on functions as well as the fact that $f = f$ implies $f \overset{n}{=} f$ for any n.

Composition distributes over R, $R(f \circ g) = R(f) \circ R(g)$.

$$\begin{aligned}R(f) \circ R(g)(s, \Phi, H) &= R(f)(s, \Phi, g(H)) \\ &= (s, \Phi, f(g(H))) \\ &= R(f \circ g)\end{aligned}$$

Idenntity distributes over R, $R(id) = id$.

$$R(id) = \lambda(s, \Phi, H).\, (s, \Phi, id(H)) = \lambda(s, \Phi, H).\, (s, \Phi, H) = id$$

□

Proof of Lemma 8. $G(f) : G(Y) \to G(X)$ is *non-expansive* for $f : X \overset{mon.\ ne}{\to} Y$. For $H \overset{n}{=} H'$ show

$$G(f)(H) \overset{n}{=} G(f)(H')$$

which expands to

$$\lambda st.\, \lambda w.\, H(st)(f(w)) \overset{n}{=} \lambda st.\, \lambda w.\, H'(st)(f(w))$$

Given $st \overset{n}{=} st'$ and $w \overset{n}{=} n'$ SFTS

$$.\, H(st)(f(w)) \overset{n}{=} H'(st')(f(w'))$$

As f is non-expansive, we get $f(w) \overset{n}{=} f(w')$. As $H \overset{n}{=} H'$ we get $H(st) \overset{n}{=} H(st')$. Combining the two, we get the desired result. □

Proof of Lemma 9. Follows from Lemmas 6, 7, and 8. □

Proof of Lemma 10. Let k, f and g be given and assume $f \overset{k}{=} g$. Show

$$F(f) \overset{k}{=} F(g).$$

SFTS

$$\forall \phi.\, (F(f))(\phi) \overset{k}{=} (F(g))(\phi)$$

Let ϕ be given. We need to show $\mathrm{dom}((F(f))(\phi)) = \mathrm{dom}((F(g))(\phi))$. Both sides are defined exactly when ϕ, so they have the same domain. We also need to show

$$\forall n \in \mathrm{dom}((F(f))(\phi)).\, (F(f))(\phi)(n) \overset{k}{=} (F(g))(\phi)(n)$$

Let $n \in \mathrm{dom}((F(f))(\phi))$ be given. Define $H = \pi_3(\phi(n))$. The above unfolds to two triples, so SFTS each component is k-equal. That is

$$\pi_1(\phi(n)) \overset{k}{=} \pi_1(\phi(n))$$

$$\pi_2(\phi(n)) \overset{k}{=} \pi_2(\phi(n))$$

$$\lambda st'.\, (H(st') \circ f) \overset{k}{=} \lambda st'.\, (H(st') \circ g)$$

The two first are trivially satisfied. For the third, we need to show that for all st if we apply st to each side, then they are still k-equal. That is given st show

$$(H(st) \circ f) \overset{k}{=}. (H(st) \circ g)$$

SFTS

$$\forall w.\, (H(st) \circ f)(w) \overset{k}{=}. (H(st) \circ g)(w)$$

Let w be given. From $f \overset{k}{=} g$, we get $f(w) \overset{k}{=} g(w)$. From this and $H(st)$ being non-expansive, we get

$$H(st)(f(w)) \overset{k}{=} H(st)(g(w))$$

which is what we neede to show.

□

A.2 Proofs of Lemmas in Section 2.2

Proof of lemma 12. Assume $W_1 \stackrel{n}{=} W_2$ and $W_1' \sqsupseteq W_2'$. Define W_2' to have $\mathrm{dom}(W_2') = \mathrm{dom}(W_1')$ where

$$W_2'\ r = \begin{cases} (s_1', \phi_2, H_2) & \text{for } j \in \mathrm{dom}(W_2) \text{ and } W_2\ j = (s_2, \phi_2, H_2) \\ & \quad \text{and } W_1'\ j = (s_1', \phi', H_1') \\ W_1'\ r & \text{otherwise} \end{cases}$$

First we show $W_1' \stackrel{n}{=} W_2'$. The domains are defined to be equal, so that is satsfied. Let $r \in \mathrm{dom}(W_1')$ and show $W_1'\ r \stackrel{n}{=} W_2'\ r$. First consider the case where $r \in \mathrm{dom}(W_2)$. Assume $W_1'\ r = (s_1', \phi_1', H_1')$, $W_1 = (s_1, \phi_1, H_1)$, and $W_2 = (s_2, \phi_2, H_2)$, then $W_2'\ r = (s_1', \phi_2, H_2)$. We need to show

$$s_1' \stackrel{n}{=} s_1'$$
$$\phi_1' \stackrel{n}{=} \phi_2$$
$$H_1' \stackrel{n}{=} H_2$$

The first one is trivial. For the second one we use that we know $\phi_1 \stackrel{n}{=} \phi_2$ and $\phi_1 = \phi_1'$ respectively from the n-equality assumption and future world assumption. Similarly, the second one follows from $H_1 \stackrel{n}{=} H_2$ and $H_1 = H_1'$ - again respectively from the n-equality assumption and the future world assumption. For the case where $r \notin \mathrm{dom}(W_2)$, we need to show $W_1'\ r \stackrel{n}{=} W_1'\ r$ which follows trivially from reflexivity.

Next we show $W_2' \sqsupseteq W_2$. We know that $\mathrm{dom}(W_2') = \mathrm{dom}(W_1')$, $\mathrm{dom}(W_1') \supseteq \mathrm{dom}(W_1)$ (from future world assumption), and $\mathrm{dom}(W_1) = \mathrm{dom}(W_2)$ (from n-equality assumption). All this can easily be put together to get $\mathrm{dom}(W_2') \supseteq \mathrm{dom}(W_2)$. Now let $r \in \mathrm{dom}(W_2)$ and assume Assume $W_1'\ r = (s_1', \phi_1', H_1')$, $W_1 = (s_1, \phi_1, H_1)$, and $W_2 = (s_2, \phi_2, H_2)$, then $W_2'\ r = (s_1', \phi_2, H_2)$, then we need to show the following:

$$(s_2, s_1') \in \phi_2$$
$$\phi_2 = \phi_2$$
$$H_2 = H_2$$

Here the two last are trivial. The first follows from the assumptions $s_1 = s_2$, $\phi_1 = \phi_2$, and $(s_1, s_1') \in \phi_1$, where the first two are from the n-equality assumption (notive that n-equality between c.o.f.e.'s over sets is just equality) and the last one follows from the future world assumption. □

Proof of lemma 34. Follows directly from definition: Let $\mathcal{V}$, n, n', *base*, and *end* be given. Assume $n \in \mathit{readCondition}(\mathcal{V})(\mathit{base}, \mathit{end}, W)$ and $n' \leq n$. If $n = n'$, then the result follows trivially, as it is exactly our assumption. If $n' < n$, then by assumption we have a region name r, interval $[\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]$, and the assumption $W(r) \stackrel{n-1}{\subseteq} \iota_{\mathit{base}', \mathit{end}'}$ given by the read condition assumption.

Picking r and $[base', end']$, we need to show $W(r) \overset{n'-1}{\subseteq} \iota_{base',end'}$ which follows from downwards closure of $\overset{n-1}{\subseteq}$. □

A.3 Proofs of Lemmas in Section 2.3

A.3.1 Proofs of Lemmas in Section 2.3.1

Proof of lemma 15. Let $n' < n$, W, reg, and hs be given. Assume $(n, (reg, hs)) \in \mathcal{O}(W)$. Let $heap_f$, $heap'$, $i \leq n'$ be given and assume $(reg, hs \uplus heap_f) \rightarrow_i (halted, heap')$. By assumption, we have a $W' \sqsupseteq W$ and hs' such that

$$heap' = hs' \uplus heap_f \tag{15}$$

$$hs' :_{n-i} W' \tag{16}$$

. Using W' and hs' as existential witnesses, we already have Equation 15 as the first necessary condition and from the above heap satisfaction along with heap satisfaction being uniform in n, we get $hs' :_{n'-i} W'$. These are the two conditions necessary to get $(n', (reg, hs)) \in \mathcal{O}(W)$. □

Proof of lemma 16. □

A.3.2 Proofs of Lemmas in Section 2.3.2

Proof of lemma 17. □

Proof of lemma 18. □

Proof of lemma 20. □

A.3.3 Proofs of Lemmas in Section 2.3.3

Proof of lemma 21. □

Proof of lemma 22. □

Proof of lemma 24. □

A.3.4 Proofs of Lemmas in Section 2.3.4

Proof of lemma 25. □

Proof of lemma 26. □

Proof of lemma 27. □

A.3.5 Proofs of Lemmas in Section 2.3.5

Proof of Lemma 28. □

Proof of Lemma 29. □

Proof of Lemma 30. □

Proof of Lemma 33. □

A.3.6 Proofs of Lemmas in Section 2.3.6

Proof of lemma 35. □

Proof of lemma 36. □

Proof of lemma 37. □

Proof of lemma 38. □

Proof of lemma 40. □

Proof of lemma 41. □

Proof of lemma 42. □

Proof of lemma 43. Like the *executeCondition* uniformity proof. □

Proof of lemma 44. Follows directly from definition. □

Proof of lemma 45. This proof goes like the proof of *executeCondition* being non-expansive in the second argument (lemma 42). □

Proof of lemma 46. □

Proof of lemma 47. □

Proof of lemma 48. □

Proof of lemma 49. □

Proof of lemma 50. Follows directly from the definition. □

A.3.7 Proofs of Lemmas in Section 2.3.7

Proof of lemma 51. Follows from the uniformity of *readCondition*, *readWriteCondition*, *executeCondition*, and *entryCondition*. □

Proof of lemma 52. □

Proof of lemma 53. Follows from monotonicity of *readCondition*, *readWriteCondition*, *executeCondition*, and *entryCondition* in the worlds. That is Lemma 35, 38, 41, and 44. □

Proof of lemma 54. □

Proof of lemma 55. □

A.4 Proofs of Lemmas in Section 2.5

Proof of Lemma 56. □