Definition and soundness proof for address-based RRA

July 6, 2021

1 Programming model

1.1 Static definitions

type	$\rightarrow$	`bool` | `uint` | `pointer` (of `uint`)
program	$\rightarrow$	decls instrs
decl	$\rightarrow$	type `id`, `id` $\in$ `V`
instr	$\rightarrow$	for `id` := `exp`... `exp` do instrs (for loop, `id` is typed `uint`)
	$\rightarrow$	assign `id` `exp` (variable assignment)
	$\rightarrow$	assignmem `exp` `exp` (memory location assignment)
	$\rightarrow$	assert `exp` (assert)
	$\rightarrow$	assume `exp` (assume)
	$\rightarrow$	malloc `id` `exp` (malloc "exp" sized uint space for pointer `id`)
	$\rightarrow$	skip
	$\rightarrow$	fail
exp	$\rightarrow$	`id` where `id` $\in$ `V` or `id` is a loop iterator.
	$\rightarrow$	`constant` where `constant` is a `bool` or `uint` value.
	$\rightarrow$	$*$`exp` where the pointer "exp" is dereferenced, i.e. `memory`[`exp.base`][`exp.offset`]
	$\rightarrow$	`exp` $\pm$ `exp` (both uint or pointer $\pm$ uint)
	$\rightarrow$	`exp` rel `exp` (relation operators such as $a <= b$)
	$\rightarrow$	`exp`?`exp` : `exp` (conditional expression)
	$\rightarrow$	`valid`(`exp`) (check whether a pointer points to a valid memory location)

type Our program model supports 3 types: `bool`, `uint` and `pointer`. Each pointer pointers to a location in the memory which can be dereferenced into an `uint` value. `pointer` type, together with memory of the program, will be talked about in the memory paragraph.

program Our program contains a set of declarations of variables, a list of instructions and a memory which is maintained as a dictionary at runtime. Note that function calls are not contained in our program model for simplicity of the proof. However, they can be emulated by our program model so it will not invalid our soundness proof.

decl In our program, a number of variables are declared and they can be accessed during execution.

instr The program consists a list of instructions. The instructions will be executed in order when we run the program. Executing an instruction will change the state s of a program. The state s is given by valuation of all variables, arrays in the memory and valuation of all entries in the memory.

exp Exps are expressions over variables/memorys that can be evaluated to a concrete value at a given state s of a program.

1.2 Runtime States

memory Memory maintains memory spaces allocated for `uint` arrays. At a specific state furing runtime, it is a list containing all uint arrays allocated upon that time. At the initial state of the program, memory is empty. A new array would be inserted to memory everytime we run the `malloc` instruction.

Each `pointer` p is a reference to a specific entry in one of the arrays in memory. It has two fields: $p.\texttt{base}$ and $p.\texttt{offset}$, meaning it is accessing the $p.\texttt{offset}$-th entry in the array $p.\texttt{base}$ in memory, which is represented as $\texttt{memory}[p.\texttt{base}][p.\texttt{offset}]$. Assigning $p1$ to $p2$ would make $p2$ have the same base and offset as $p1$. Addition/subtraction between a `pointer` and an `uint` would act as the corresponding operation on the `pointer`'s offset.

For example, say if memory is [A1 (array of `uint` with size 8), A2 (size 10)]. When we execute an instruction "`malloc` p 4", an new array would be inserted to memory, making the memory become [A1 (size 8), A2 (size 10), A3 (size 4)]. $p.\texttt{base}$ would become A3 and its offset would be set to 0. The expression $*(p+2)$ would access A3[2] in the memory.

The length of an array is represented as $\texttt{memory}[p.\texttt{base}][p.\texttt{offset}].\texttt{length}$.

state A state s at a certain time during program execution is determined by all valuation of all variables as well as the valuation of `memory` (arrays stored and content of arrays). We use the denotation $s\langle\texttt{id}\rangle$ to represent the valuation of the variable `id` at the state s. Similar to variables, the `memory` at state s is represented with $s\langle\texttt{memory}\rangle$.

To show changes to states, we write $s\{\texttt{id} \mapsto v\}$ to represent a state that is identical to s except that the variable `id` has a new value v. Similarly, we write $s\{\texttt{memory}[A][offset] \mapsto v\}$ to represent a state identical to s except that the $A[offset]$ element has a new value v. $s\{\texttt{memory}[A] \mapsto \texttt{array}(v)\}$ shows a state identical to s except that a new array A is initialized with size v.

1.3 Semantics

Semantics for evaluation of expressions We write $\langle\texttt{exp}, s\rangle \to_e^* v$ to indicate the expression `exp` is evaluated to the specific value v at state s.

$$\frac{s\langle\texttt{id}\rangle = v}{\langle\texttt{id}, s\rangle \to_e^* v}$$

$$\frac{}{\langle\texttt{constant}, s\rangle \to_e^* \texttt{constant}}$$

$$\frac{\langle\texttt{exp}, s\rangle \to_e^* ptr, s\langle\texttt{memory}\rangle[ptr.\texttt{base}][ptr.\texttt{offset}] = v}{\langle *\texttt{exp}, s\rangle \to_e^* v}$$

$$\frac{\langle\texttt{exp}_1, s\rangle \to_e^* a, \langle\texttt{exp}_2, s\rangle \to_e^* b, v = \texttt{pointer}(a.\texttt{base}, a.\texttt{offset} \pm b)}{\langle\texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \to_e^* v} \quad , \ \texttt{exp}_1 \text{ is a } \texttt{pointer}$$

$$\frac{\langle\texttt{exp}_1, s\rangle \to_e^* a, \langle\texttt{exp}_2, s\rangle \to_e^* b, v = a \pm b}{\langle\texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \to_e^* v} \quad , \ \texttt{exp}_1 \text{ is an } \texttt{uint}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \to_e^* v}{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \to_e^* v}, \quad \frac{\langle \texttt{exp}_1, s\rangle \to_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \to_e^* v}{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \to_e^* v}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle \texttt{memory}\rangle, ptr.\texttt{offset} < s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{true}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\langle \texttt{memory}\rangle}{\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{false}}, \quad \frac{\langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle \texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{false}}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* a, \langle \texttt{exp}_2, s\rangle \to_e^* b, v = a \text{ rel } b}{\langle \texttt{exp}_1 \text{ rel } \texttt{exp}_2, s\rangle \to_e^* v}$$

Information about valuation of subexpressions and the state can be inferred from valuation of expressions:

$$\frac{\langle \texttt{id}, s\rangle \to_e^* v}{s\langle \texttt{id}\rangle = v}$$

$$\frac{\langle *\texttt{exp}, s\rangle \to_e^* v}{\exists ptr, \langle \texttt{exp}, s\rangle \to_e^* ptr, s\langle \texttt{memory}\rangle[ptr.\texttt{base}][ptr.\texttt{offset}] = v}$$

$$\frac{\texttt{type}(\texttt{exp}_1) == \texttt{pointer}, \langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \to_e^* v}{\exists a, b \text{ s.t. } \langle \texttt{exp}_1, s\rangle \to_e^* a, \langle \texttt{exp}_2, s\rangle \to_e^* b, v = \texttt{pointer}(a.\texttt{base}, a.\texttt{offset} \pm b)}$$

$$\frac{\texttt{type}(\texttt{exp}_1) == \texttt{uint}, \langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \to_e^* v}{\exists a, b \text{ s.t. } \langle \texttt{exp}_1, s\rangle \to_e^* a, \langle \texttt{exp}_2, s\rangle \to_e^* b, v = a \pm b}$$

$$\frac{\langle \texttt{exp}_1 \text{ rel } \texttt{exp}_2, s\rangle \to_e^* v}{\exists a, b \text{ s.t. } \langle \texttt{exp}_1, s\rangle \to_e^* a, \langle \texttt{exp}_2, s\rangle \to_e^* b, v = a \text{ rel } b}$$

$$\frac{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \to_e^* v}{(\exists v, \langle \texttt{exp}_1, s\rangle \to_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \to_e^* v) \text{ or } (\exists v, \langle \texttt{exp}_1, s\rangle \to_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \to_e^* v)}$$

$$\frac{\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{true}}{\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle \texttt{memory}\rangle, ptr.\texttt{offset} < s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}$$

$$\frac{\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{false}}{(\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\langle \texttt{memory}\rangle) \text{ or } (\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle \texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length})}$$

Semantics for executing instructions We write $\langle \texttt{inst}_1, s_1\rangle \to \langle \texttt{inst}_2, s_2\rangle$ to show that executing $\texttt{inst}_1$ at state s_1 will lead to state s_2 with the next instruction to execute as $\texttt{inst}_2$.

Sequencing instructions:

$$\frac{\langle c_0, s\rangle \to \langle c_0', s'\rangle}{\langle c_0; c_1, s\rangle \to \langle c_0'; c_1, s'\rangle}$$

Skip/fail instructions:

$$\frac{}{\langle \text{skip}; c, s\rangle \to \langle c, s\rangle}$$

$$\frac{}{\langle \text{fail}; c, s\rangle \to \langle \text{fail}, s\rangle}$$

Assign instructions:

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* v}{\langle \text{assign } \texttt{id} \ \texttt{exp}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto v\}\rangle}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \mathit{ptr}, \langle \texttt{exp}_2, s\rangle \to_e^* v}{\langle \text{assignmem } \texttt{exp}_1 \ \texttt{exp}_2, s\rangle \to \langle \text{skip}, s\{\texttt{memory}[\mathit{ptr}.\texttt{base}][\mathit{ptr}.\texttt{offset}] \mapsto v\}\rangle}$$

Assert instructions:

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{false}}{\langle \text{assert } \texttt{exp}, s\rangle \to \langle \text{fail}, s\rangle}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{true}}{\langle \text{assert } \texttt{exp}, s\rangle \to \langle \text{skip}, s\rangle}$$

Assume instructions actually has the same semantic at program leval as asserts, but it tells solvers to only solve for some execution paths:

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{false}}{\langle \text{assume } \texttt{exp}, s\rangle \to \langle \text{fail}, s\rangle}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{true}}{\langle \text{assume } \texttt{exp}, s\rangle \to \langle \text{skip}, s\rangle}$$

Malloc instructions:

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{size}, l_{new} = \texttt{newloc}(s\langle \texttt{memory}\rangle)}{\langle \text{malloc } \texttt{id} \ \texttt{exp}, s\rangle \to \langle \text{skip}, s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\mathit{size}), \texttt{id} \mapsto \texttt{pointer}(l_{new}, 0)\}\rangle}$$

For loops:

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \mathit{start}, \langle \texttt{exp}_2, s\rangle \to_e^* \mathit{end}, \mathit{start} \le \mathit{end}}{\langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \langle \text{insts}; \text{for } \texttt{id} := \mathit{start} + 1 \ldots \mathit{end} \text{ do insts}, s\{\texttt{id} \mapsto \mathit{start}\}\rangle}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \mathit{start}, \langle \texttt{exp}_2, s\rangle \to_e^* \mathit{end}, \mathit{start} > \mathit{end}}{\langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto \mathit{start}\}\rangle}$$

Information about valuation of related expressions and the state can be inferred from state transitions cased by instructions:

$$\frac{\langle \text{assign } \texttt{id} \ \texttt{exp}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto v\}\rangle}{\langle \texttt{exp}, s\rangle \to_e^* v}$$

$$\frac{\langle \text{assignmem } \texttt{exp}_1\ \texttt{exp}_2, s\rangle \rightarrow \langle \text{skip}, s\{\texttt{memory}[base][offset] \mapsto v\}\rangle}{\exists ptr \text{ s.t. } \langle \texttt{exp}_1, s\rangle \rightarrow_e^* ptr, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* v, ptr.\texttt{base} = base, ptr.\texttt{offset} = offset}$$

$$\frac{\langle \text{malloc } \texttt{id}\ \texttt{exp}, s\rangle \rightarrow \langle \text{skip}, s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(size), \texttt{id} \mapsto \texttt{pointer}(l_{new}, 0)\}\rangle}{l_{new} = \texttt{newloc}(s\langle\texttt{memory}\rangle), \langle \texttt{exp}, s\rangle \rightarrow_e^* v}$$

$$\frac{\langle \text{assert } \texttt{exp}, s\rangle \rightarrow \langle \text{skip}, s\rangle}{\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{true}}, \quad \frac{\langle \text{assume } \texttt{exp}, s\rangle \rightarrow \langle \text{skip}, s\rangle}{\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{true}}$$

1.4 Failure semantics

Semantics for expression evaluation failures

$$\frac{\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}{\langle *\texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}, \quad \frac{\langle \texttt{valid}(\texttt{exp}), s\rangle \rightarrow_e^* \texttt{false}}{\langle *\texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}, \quad \frac{\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 \text{ rel } \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}, \quad \frac{\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 \text{ rel } \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}}, \quad \frac{\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}}, \quad \frac{\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}}$$

$$\frac{\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{valid}(\texttt{exp}), s\rangle \rightarrow_e^* \texttt{fail}}$$

Information about evaluations of subexpressions can be inferred:

$$\frac{\langle *\texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}{(\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{valid}(\texttt{exp}), s\rangle \rightarrow_e^* \texttt{false})}$$

$$\frac{\langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}{(\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail})}, \quad \frac{\langle \texttt{exp}_1 \text{ rel } \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}{(\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail})}$$

$$\frac{\langle \texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}}{(\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail})}$$

$$\frac{\langle \texttt{valid}(\texttt{exp}), s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}$$

Semantics for instruction failures because of expression failures

$$\frac{\langle \mathtt{exp}_1, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{for } \mathtt{id} := \mathtt{exp}_1 \ldots \mathtt{exp}_2 \text{ do insts}, s\rangle \rightarrow \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \mathtt{exp}_2, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{for } \mathtt{id} := \mathtt{exp}_1 \ldots \mathtt{exp}_2 \text{ do insts}, s\rangle \rightarrow \langle \text{fail}, s\rangle}$$

$$\frac{\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{assign } \mathtt{id}\ \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \mathtt{exp}_1, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{assignmem } \mathtt{exp}_1\ \mathtt{exp}_2, s\rangle \rightarrow \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \mathtt{exp}_2, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{assignmem } \mathtt{exp}_1\ \mathtt{exp}_2, s\rangle \rightarrow \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \mathtt{valid(exp}_1\mathtt{)}, s\rangle \rightarrow_e^* \mathtt{false}}{\langle \text{assignmem } \mathtt{exp}_1\ \mathtt{exp}_2, s\rangle \rightarrow \langle \text{fail}, s\rangle}$$

$$\frac{\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{assert } \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{assume } \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}$$

$$\frac{\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail}}{\langle \text{malloc } \mathtt{id}\ \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}$$

Information about expressions shown up in instructions can be inferred from instruction failures:

$$\frac{\langle \text{for } \mathtt{id} := \mathtt{exp}_1 \ldots \mathtt{exp}_2 \text{ do insts}, s\rangle \rightarrow \langle \text{fail}, s\rangle}{(\langle \mathtt{exp}_1, s\rangle \rightarrow_e^* \mathtt{fail}) \text{ or } (\langle \mathtt{exp}_2, s\rangle \rightarrow_e^* \mathtt{fail})}\ ,\ \frac{\langle \text{assign } \mathtt{id}\ \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}{\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail}}$$

$$\frac{\langle \text{assignmem } \mathtt{exp}_1\ \mathtt{exp}_2, s\rangle \rightarrow \langle \text{fail}, s\rangle}{(\langle \mathtt{exp}_1, s\rangle \rightarrow_e^* \mathtt{fail}) \text{ or } (\langle \mathtt{exp}_2, s\rangle \rightarrow_e^* \mathtt{fail}) \text{ or } (\langle \mathtt{valid(exp}_1\mathtt{)}, s\rangle \rightarrow_e^* \mathtt{false})}$$

$$\frac{\langle \text{malloc } \mathtt{id}\ \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}{\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail}}$$

$$\frac{\langle \text{assert } \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}{(\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{false}) \text{ or } (\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail})}\ ,\ \frac{\langle \text{assume } \mathtt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}{(\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{false}) \text{ or } (\langle \mathtt{exp}, s\rangle \rightarrow_e^* \mathtt{fail})}$$

2 RRA definition

2.1 Abstraction shape/abstraction functions

To start the abstraction process the user first has to specify the set of pointers along with the shapes to use. When allocating arrays in memory for these pointers, we will be allocating abstracted arrays. They also have to specify loop iterators ids used to iterate over such arrays. Denote these sets of variable identifiers by *PointersAbst* and *IndicesAbst* respectively.

The shape of the abstraction represents how we would like to map real concrete locations to abstract locations. Take the shape "*c*c*" as an example which keeps track of two locations $c_1 < c_2$. In general, the shape $*c*c*$ can also represent cases where $(c_1 = c_2 - 1)$, that is, the precisely tracked indices are adjacent. This would complicate the abstraction and concretization functions $\alpha_{(c_1,c_2)}$ and $\gamma_{(c_1,c_2)}$ described below. To keep the presentation simple we assume $c_1 < c_2 - 1$. Extension to other shapes is straightforward. The abstraction function $\alpha_{(c_1,c_2)}$ mapping concrete indices to abstract indices is parameterized by the values c_1, c_2 for the precise locations and is as follows:

$$\begin{aligned}
\alpha_{(c_1,c_2)}(v) &= 0 \text{ if } v < c_1 \\
&= 1 \text{ if } v = c_1 \\
&= 2 \text{ if } c_1 < v < c_2 \\
&= 3 \text{ if } v = c_2 \\
&= 4 \text{ if } v > c_2
\end{aligned}$$

There will be a corresponding concretization function as follows:

$$
\begin{array}{lcl}
\gamma_{(c_1,c_2)}(0) & = & \{v \mid \text{ where } 0 \le v < c_1\} \\
\gamma_{(c_1,c_2)}(1) & = & \{c_1\} \\
\gamma_{(c_1,c_2)}(2) & = & \{v \mid \text{ where } 0 \le c_1 < v < c_2\} \\
\gamma_{(c_1,c_2)}(3) & = & \{c_2\} \\
\gamma_{(c_1,c_2)}(4) & = & \{v \mid \text{ where } v > c_2\}
\end{array}
$$

In this document, we the concrete locations $c1$ and $c2$ are not important so we hide them when writing abstraction functions, namely α and γ. Note that for any v, $v \in \gamma(\alpha(v))$, which will be used to prove the soundness later.

$$\frac{}{v \in \gamma(\alpha(v))}$$

When we say we abstract an array in the memory, say A, we only keep track of values at concrete locations ('c's in the shape). Other locations are handled in an abstracted way - reading from abstract locations will result in a non-deterministic value and writing to abstract locations will be discarded. Thus, when allocating an array with size len that needs to be abstracted, we only need to allocate a space sized $\alpha(len)$

When a loop iterator `id` is abstracted, we only iterate each abstracted location ('*' in the shape) once. For example, if in the original program we `id` takes values from 0 to 100, then in the abstracted program with shape "*c*c*" where $c2 = 100$, the abstracted `id` will only take values from 0 to 3. This could lead to soundness issues if there are true dependences between iterations of this loop.

We also introduce a function to tell whether an index corresponds to a location that is precisely tracked:

$$
\begin{array}{lcl}
\beta_{(c_1,c_2)}(c) & = & \texttt{true} \text{ if } c == c_1 \text{ or } c == c_2 \\
 & = & \texttt{false} \text{ otherwise}
\end{array}
$$

2.2 Program states in abstracted programs

In the abstracted program, we use $s\$abst$ (with a $\$abst$ suffix) to represent its state at a given time. Note that it is just a suffix and does not mean it is obtained by transformation over s. In addition to information that is also available in the original programs' states, a state $s\$abst$ in the abstracted program also maintains a set `abstmem` identifying which arrays in `memory` are kept in an abstract way. Memory arrays in `abstmem` are abstracted arrays of arrays in the original program. I.e. an memory array $A \in$ `abstmem` which is originally sized len in the original program becomes one with size $\alpha(len)$. We use $s\$abst\langle\texttt{abstmem}\rangle$ to represent the set `abstmem` at state $s\$abst$, and $s\$abst\{\texttt{abstmem.add}(A)\}$ to represnet the state that is identical to $s\$abst$ except that a new array A is added to `abstmem`.

For each array in the memory, we also use a new field `conc_length` to maintain its concrete length which corresponds to its length in the original program. For an non-abstracted array $A \notin s\$abst\langle\texttt{abstmem}\rangle$, its allocated size equals to its concrete length, i.e. $s\$abst\langle\texttt{memory}\rangle[A].\texttt{length} = s\$abst\langle\texttt{memory}\rangle[A].\texttt{conc\_length}$.

Since we now keep track of both `length` and `conc_length`, when initializing an array A, we need to present both pieces of information:

$$s\$abst\{\texttt{memory}[A] \mapsto \texttt{array}(len, conc\_len)\}$$

2.3 New expressions/instructions introduced in abstracted programs

Two new instructions (abstract for loop & abstract a memory location) are introduced. Several expressions are introduced to handle pointers in an abstract way.

instr	$\rightarrow$	abstfor `id`:=exp. . . exp `do` instrs (for loop, `id` is typed `uint`)
	$\rightarrow$	abstmalloc `id` `exp` (allocate an abstract memory space and assign to `id`)
exp	$\rightarrow$	`ndbool` (it can be evaulated `true` or `false`)
	$\rightarrow$	`nduint` (it can be evaluated to any uint number)
	$\rightarrow$	`in_abstmem(exp)` (return whether the pointer `exp` points to memory in `abstmem`)
	$\rightarrow$	`is_prec(exp)` (return whether the pointer `exp` points to a precise location)
	$\rightarrow$	`abst_valid(exp)` (check whether an access to a pointer whose base is in `abstmem` is valid)
	$\rightarrow$	`abst_ptr(exp)` (transform the pointer `exp`'s offset to abstracted offset)
	$\rightarrow$	`abst_deref(exp)` (access value for a pointer `exp` whose base represents an abstracted array, i.e. in `abstmem`)
	$\rightarrow$	`concretize(exp)` (concretize `exp` (abstract index space) to a concrete index value; this is non-deterministic)
	$\rightarrow$	`abstract(exp)` (abstract `exp` (concrete index space) to an abstract index value)

2.3.1 Semantics

Now we define semantics for those newly introduced things.

When `id` is at precise locations, the behavior of abstract for loop is the same as regular for loops. However, when `id` is at abstract locations, we non-deterministically move the iterator forward.

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \mathit{abst\_s}, \langle \texttt{exp}_2, s\rangle \to_e^* \mathit{abst\_e}, \mathit{conc\_s} \in \gamma(\mathit{abst\_s}), \mathit{conc\_e} \in \gamma(\mathit{abst\_e}), \mathit{conc\_s} > \mathit{conc\_e}}{\langle \text{abstfor } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto \mathit{abst\_s}\}\rangle}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \mathit{abst\_s}, \langle \texttt{exp}_2, s\rangle \to_e^* \mathit{abst\_e}, \mathit{conc\_s} \in \gamma(\mathit{abst\_s}), \mathit{conc\_e} \in \gamma(\mathit{abst\_e}), \mathit{conc\_s} \leq \mathit{conc\_e}}{\begin{array}{c}\langle \text{abstfor } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \\ \langle \text{insts; abstfor } \texttt{id} := \texttt{abstract}(\mathit{conc\_s} + 1) \ldots \texttt{abstract}(\mathit{conc_e}) \text{ do insts}, s\{\texttt{id} \mapsto \mathit{abst\_s}\}\rangle\end{array}}$$

Semantics for the abstract malloc:

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{size}, l_{new} = \texttt{newloc}(s\langle\texttt{memory}\rangle)}{\langle \text{mallocabst } \texttt{id exp}, s\rangle \to \langle \text{skip}, s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\alpha(\mathit{size}), \mathit{size}), \texttt{id} \mapsto \texttt{pointer}(l_{new}, 0), \texttt{abstmem.add}(l_{new})\}\rangle}$$

Note that the semantic for regular mallocs also slightly changes because we have a new field `conc_length` for arrays.

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{size}, l_{new} = \texttt{newloc}(s\langle\texttt{memory}\rangle)}{\langle \text{malloc } \texttt{id exp}, s\rangle \to \langle \text{skip}, s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\mathit{size}, \mathit{size}), \texttt{id} \mapsto \texttt{pointer}(l_{new}, 0)\}\rangle}$$

Semantics for the newly introduced expressions:

$$\frac{}{\langle \texttt{ndbool}, s\rangle \to_e^* \texttt{true}} \quad \frac{}{\langle \texttt{ndbool}, s\rangle \to_e^* \texttt{false}}$$

$$\frac{}{\langle \texttt{nduint}, s\rangle \to_e^* v}, v \text{ is an uint}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* v, v.\texttt{base} \in s\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem(exp)}, s\rangle \to_e^* \texttt{true}} \quad \frac{\langle \texttt{exp}, s\rangle \to_e^* v, v.\texttt{base} \notin s\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem(exp)}, s\rangle \to_e^* \texttt{false}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \beta(\mathit{ptr}.\texttt{offset}) = \texttt{false}}{\langle \texttt{is\_prec(exp)}, s\rangle \to_e^* \texttt{false}}, \frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \beta(\mathit{ptr}.\texttt{offset}) = \texttt{true}}{\langle \texttt{is\_prec(exp)}, s\rangle \to_e^* \texttt{true}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \mathit{ptr}.\texttt{base} \in s\langle\texttt{memory}\rangle, \mathit{ptr}.\texttt{offset} < s\langle\texttt{memory}\rangle[\mathit{ptr}.\texttt{base}].\texttt{conc\_length}}{\langle \texttt{abst\_valid(exp)}, s\rangle \to_e^* \texttt{true}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \mathit{ptr}.\texttt{base} \notin s\langle\texttt{memory}\rangle}{\langle \texttt{abst\_valid(exp)}, s\rangle \to_e^* \texttt{false}}, \frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \mathit{ptr}.\texttt{base} \in s\langle\texttt{memory}\rangle, \mathit{ptr}.\texttt{offset} \geq s\langle\texttt{memory}\rangle[\mathit{ptr}.\texttt{base}].\texttt{conc\_length}}{\langle \texttt{abst\_valid(exp)}, s\rangle \to_e^* \texttt{false}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}}{\langle \texttt{abst\_ptr(exp)}, s\rangle \to_e^* \texttt{pointer}(\mathit{ptr}.\texttt{base}, \alpha(\mathit{ptr}.\texttt{offset}))}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \beta(\mathit{ptr}.\texttt{offset}) = \texttt{true}, s\langle\texttt{memory}\rangle[\mathit{ptr}.\texttt{base}][\alpha(\mathit{ptr}.\texttt{offset})] = v}{\langle \texttt{abst\_deref(exp)}, s\rangle \to_e^* v}, \frac{\langle \texttt{exp}, s\rangle \to_e^* \mathit{ptr}, \beta(\mathit{ptr}.\texttt{offset}) = \texttt{false}}{\langle \texttt{abst\_deref(exp)}, s\rangle \to_e^* v} \text{ (for any uint } v)$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* v, c \in \gamma(v)}{\langle \texttt{concretize(exp)}, s\rangle \to_e^* c}, \frac{\langle \texttt{exp}, s\rangle \to_e^* v}{\langle \texttt{abstract(exp)}, s\rangle \to_e^* \alpha(v)}$$

2.3.2 Failure semantics

Semantics for expression evaluation failures

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{in\_abstmem(exp)}, s\rangle \to_e^* \texttt{fail}}\ ,\ \frac{\langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\langle \texttt{memory}\rangle}{\langle \texttt{in\_abstmem(exp)}, s\rangle \to_e^* \texttt{fail}}\ ,\ \frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{is\_prec(exp)}, s\rangle \to_e^* \texttt{fail}}\ ,\ \frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{abst\_valid(exp)}, s\rangle \to_e^* \texttt{fail}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{abst\_ptr(exp)}, s\rangle \to_e^* \texttt{fail}}\ ,\ \frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{concretize(exp)}, s\rangle \to_e^* \texttt{fail}}\ ,\ \frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{abstract(exp)}, s\rangle \to_e^* \texttt{fail}}$$

$$\frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \texttt{abst\_deref(exp)}, s\rangle \to_e^* \texttt{fail}}\ ,\ \frac{\langle \texttt{abst\_valid(exp)}, s\rangle \to_e^* \texttt{false}}{\langle \texttt{abst\_deref(exp)}, s\rangle \to_e^* \texttt{fail}}$$

Semantics for instruction failures because of expression failures

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \texttt{fail}}{\langle \text{abstfor } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \texttt{exp}_2, s\rangle \to_e^* \texttt{fail}}{\langle \text{abstfor } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}}{\langle \text{mallocabst } \texttt{id}\ \texttt{exp}, s\rangle \to \langle \text{fail}, s\rangle}$$

Note that we also re-define failure semantics for the "assignmem" instruction to reflect abstracted memory accesses.

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* \texttt{fail}}{\langle \text{assignmem } \texttt{exp}_1\ \texttt{exp}_2, s\rangle \to \langle \text{fail}, s\rangle}\ ,\ \frac{\langle \texttt{exp}_2, s\rangle \to_e^* \texttt{fail}}{\langle \text{assignmem } \texttt{exp}_1\ \texttt{exp}_2, s\rangle \to \langle \text{fail}, s\rangle}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\langle \texttt{abstmem}\rangle, ptr.\texttt{offset} \geq s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle \text{assignmem } \texttt{exp}_1\ \texttt{exp}_2, s\rangle \to \langle \text{fail}, s\rangle}$$

$$\frac{\langle \texttt{exp}_1, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle \texttt{abstmem}\rangle, c\_off \in \gamma(ptr.\texttt{offset}), c\_off \geq s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}}{\langle \text{assignmem } \texttt{exp}_1\ \texttt{exp}_2, s\rangle \to \langle \text{fail}, s\rangle}$$

2.4 Program transformation

We translate the program so that arrays allocated to $PointersAbst$ become abstract and loops with iterators in $IndicesAbst$ are iterated in an abstracted way. We will rewrite all instructions in the programs to make this happen. During runtime, we also keep track of a set of arrays, `abstmem`, in the memory that are kept abstract. Such information also becomes part of the program's state.

Tr(for `id` := `exp1` ... `exp2` do instrs)	
→for `id` :=Tr_{read}(`exp1`) ... Tr_{read}(`exp2`) do Tr(instrs)	if `id`$\notin$ $IndicesAbst$
→abstfor `id` :=`abstract`(Tr_{read}(`exp1`)) ... `abstract`(Tr_{read}(`exp2`)) do Tr(instrs)	if `id`$\in$ $IndicesAbst$
Tr(assign `id` `exp`)	
→assign `id` Tr_{read}(`exp`)	(we don't allow writing to $IndicesAbst$, `id` $\notin$ $IndicesAbst$)
Tr(assignmem $\texttt{exp}_1$ $\texttt{exp}_2$)	
→assignmem $Tr_{write}(\texttt{exp}_1)$ $Tr_{read}(\texttt{exp}_2)$	
Tr(malloc `id` `exp`)	
→malloc `id` Tr_{read}(`exp`)	if `id` $\notin$ $PointersAbst$
→abstmalloc `id` Tr_{read}(`exp`)	if `id` $\in$ $PointersAbst$
Tr(assert `exp`)	
→assert Tr_{read}(`exp`)	
Tr(assume `exp`)	
→assume Tr_{read}(`exp`)	

We also define transformation for expressions on the left-hand side (write) and right-hand side (read). For right-hand side expressions:

$Tr_{read}(\texttt{id})$
$\rightarrow$`concretize` (`id`) if `id`$\in$ *IndicesAbst*
$\rightarrow$`id` if `id`$\notin$ *IndicesAbst*

$Tr_{read}(\texttt{constant})$
$\rightarrow$`constant`

$Tr_{read}(^*\texttt{exp})$
$\rightarrow\texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_deref}(Tr_{read}(\texttt{exp})) : *Tr_{read}(\texttt{exp})$

$Tr_{read}(\texttt{exp}_1 \pm \texttt{exp}_2)$
$\rightarrow Tr_{read}(\texttt{exp}_1) \pm Tr_{read}(\texttt{exp}_2)$

$Tr_{read}(\texttt{exp}_1 \text{ rel } \texttt{exp}_2)$
$\rightarrow Tr_{read}(\texttt{exp}_1) \text{ rel } Tr_{read}(\texttt{exp}_2)$

$Tr_{read}(\texttt{exp}_1 ? \texttt{exp}_2 : \texttt{exp}_3)$
$\rightarrow Tr_{read}(\texttt{exp}_1) ? Tr_{read}(\texttt{exp}_2) : Tr_{read}(\texttt{exp}_3)$

$Tr_{read}(\texttt{valid}(\texttt{exp}))$
$\rightarrow\texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})) : \texttt{valid}(Tr_{read}(\texttt{exp}))$

We also define pointer transformation when writing to them:

$Tr_{write}(\texttt{exp})$
$\rightarrow\texttt{in\_abstmem}(Tr_{read}(\texttt{exp})) \ ? \ \texttt{abst\_ptr}(Tr_{read}(\texttt{exp})) \ : \ Tr_{read}(\texttt{exp})$

The program P (decls instrs) is abstracted into $P\$abst$. Where every instruction is abstracted using Tr. A state s for program P consists of all valuation of variables and memory arrays at a given time.

3 State Abstraction

We say $s\$abst$ is an abstraction of s if it satisfies the following criteria:

- $\forall \texttt{id} \notin IndicesAbst$, $s\$abst\langle \texttt{id} \rangle = s\langle \texttt{id} \rangle$
- $\forall \texttt{id} \in IndicesAbst$, $s\$abst\langle \texttt{id} \rangle = \alpha(s\langle \texttt{id} \rangle)$
- $\forall A \in s\langle \texttt{memory} \rangle$, $A \in s\$abst\langle \texttt{memory} \rangle$ and
$\begin{cases} s\$abst\langle \texttt{memory} \rangle[A].\texttt{length} = \alpha\left(s\langle \texttt{memory} \rangle[A].\texttt{length}\right); s\$abst\langle \texttt{memory} \rangle[A] \text{ is an abstraction of } s\langle \texttt{memory} \rangle[A] & \text{if } A \in s\$abst\langle \texttt{abstmem} \rangle \\ s\$abst\langle \texttt{memory} \rangle[A].\texttt{length} = s\langle \texttt{memory} \rangle[A].\texttt{length}; \forall i, s\$abst\langle \texttt{memory} \rangle[A][i] = s\langle \texttt{memory} \rangle[A][i] & \text{if } A \notin s\$abst\langle \texttt{abstmem} \rangle \end{cases}$,
$s\$abst\langle \texttt{memory} \rangle[A].\texttt{conc\_length} = s\langle \texttt{memory} \rangle[A].\texttt{length}$
- $\forall A \notin s\langle \texttt{memory} \rangle$, $A \notin s\$abst\langle \texttt{memory} \rangle$
- $\forall A \notin s\$abst\langle \texttt{memory} \rangle$, $A \notin s\langle \texttt{memory} \rangle$

$s\$abst\langle \texttt{memory} \rangle[A]$ is an abstraction of $s\langle \texttt{memory} \rangle[A]$ if and only if the following statement is true:

$$\forall i, \beta(i) \implies s\$abst\langle \texttt{memory} \rangle[A][\alpha(i)] = s\langle \texttt{memory} \rangle[A][i]$$

We write $s\$abst|s$ to indicate $s\$abst$ is an abstraction of s. Reusing the same symbol, we write $s\$abst\langle \texttt{memory} \rangle[A]|s\langle \texttt{memory} \rangle[A]$ for "$s\$abst\langle \texttt{memory} \rangle[A]$ is an abstraction of $s\langle \texttt{memory} \rangle[A]$".

Some simple conclusions can be inferred from the definition of state abstraction, which can be useful in our proof:

- If $s\$abst|s$ and $\texttt{id} \notin \mathit{IndicesAbst}$, $s\$abst\langle\texttt{id}\rangle = s\langle\texttt{id}\rangle$
- If $s\$abst|s$ and $\texttt{id} \in \mathit{IndicesAbst}$, $s\$abst\langle\texttt{id}\rangle = \alpha(s\langle\texttt{id}\rangle)$
- If $s\$abst|s$ and $base \notin s\$abst\langle\texttt{abstmem}\rangle$, $\forall offset : s\$abst\langle\texttt{memory}\rangle[base][offset] = s\langle\texttt{memory}\rangle[base][offset]$
- If $s\$abst|s$ and $base \in s\$abst\langle\texttt{abstmem}\rangle$, $\forall offset : \beta(offset) \implies s\$abst\langle\texttt{memory}\rangle[base][offset] = s\langle\texttt{memory}\rangle[base][offset]$
- If $s\$abst|s$ and $base \notin s\$abst\langle\texttt{abstmem}\rangle$, $s\$abst\langle\texttt{memory}\rangle[base].\texttt{length} = s\langle\texttt{memory}\rangle[base].\texttt{length}$
- If $s\$abst|s$ and $base \in s\$abst\langle\texttt{abstmem}\rangle$, $s\$abst\langle\texttt{memory}\rangle[base].\texttt{length} = \alpha\left(s\langle\texttt{memory}\rangle[base].\texttt{length}\right)$
- If $s\$abst|s$, $\forall base, s\$abst\langle\texttt{memory}\rangle[base].\texttt{conc\_length} = s\langle\texttt{memory}\rangle[base].\texttt{length}$
- If $s\$abst|s$ and $\texttt{id} \notin \mathit{IndicesAbst}$, $\forall v : s\$abst\{\texttt{id} \mapsto v\}|s\{\texttt{id} \mapsto v\}$
- If $s\$abst|s$ and $\texttt{id} \in \mathit{IndicesAbst}$, $\forall v : s\$abst\{\texttt{id} \mapsto \alpha(v)\}|s\{\texttt{id} \mapsto v\}$
- If $s\$abst|s$ and $base \notin s\$abst\langle\texttt{abstmem}\rangle$, $\forall offset, v : s\$abst\{\texttt{memory}[base][offset] \mapsto v\}|s\{\texttt{memory}[base][offset] \mapsto v\}$
- If $s\$abst|s$ and $base \in s\$abst\langle\texttt{abstmem}\rangle$, $\forall offset, v : s\$abst\{\texttt{memory}[base][\alpha(offset)] \mapsto v\}|s\{\texttt{memory}[base][offset] \mapsto v\}$
- If $s\$abst|s$, for any l_{new} s.t. $l_{new} = \texttt{newloc}(s\langle\texttt{memory}\rangle)$, $\forall size, s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(size, size)\}|s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(size)\}$
- If $s\$abst|s$, for any l_{new} s.t. $l_{new} = \texttt{newloc}(s\langle\texttt{memory}\rangle)$, $\forall size, s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\alpha(size), size), \texttt{abstmem.add}(l_{new})\}|s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(size)\}$

Note that the initial state of $P\$abst$ ($init\$abst$) is also an abstraction of the initial state of P ($init$) because their `memory`s are both empty, $init\$abst\langle\texttt{abstmem}\rangle$ is empty and all `id`s are at initial state $\perp$.

4 Proof

We will prove two lemmas and use the them to prove the soundness statement.

Lemma 1 *If* $\langle\texttt{exp}, s\rangle \rightarrow_e^* v$ *and* $s\$abst$ *is an abstraction of* s*,* $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* v$*.*

Lemma 2 *If* $\langle \mathrm{inst}_1, s_1\rangle \rightarrow \langle \mathrm{inst}_2, s_2\rangle$ *and* $s_1\$abst$ *is an abstraction of* s_1*,* $\exists s_2\$abst$ *such that* $s_2\$abst$ *is an abstraction of* s_2 *and* $\langle Tr(\mathrm{inst}_1), s_1\$abst\rangle \rightarrow^* \langle Tr(\mathrm{inst}_2), s_2\$abst\rangle$*.*

Theorem 1 (Soundness) *If* `assert(exp)` *fails in* `P` *then* `assert(`Tr_{read}`(exp))` *can fail in* `P$abst`

Lemma 1 ensures that every expression shown up in the original program can be evaluated to the same value after transformation. Lemma 2 guarantees that there exists a refinement mapping between the abstracted program and the original program, or in another way, the abstracted program is a simulation of the original one.

4.1 Lemma 1

The lemma can be proved inductively since `exp`s are defined in this way.

4.1.1 Without valuation falures

Variable First of all, we prove the lemma in the cases where `exp` is a variable.

$$\frac{\langle \mathtt{id}, s\rangle \to_e^* v}{v = s\langle \mathtt{id}\rangle}$$

When $\mathtt{id} \in \mathit{IndicesAbst}$, $Tr_{read}(\mathtt{id}) \to \mathtt{concretize}(\mathtt{id})$. Given $s\$abst|s$ and $\mathtt{id} \in \mathit{IndicesAbst}$, $s\$abst\langle \mathtt{id}\rangle = \alpha(s\langle \mathtt{id}\rangle) = \alpha(v)$:

$$\frac{\dfrac{s\$abst\langle \mathtt{id}\rangle = \alpha(v)}{\langle \mathtt{id}, s\$abst\rangle \to_e^* \alpha(v)}, \dfrac{}{v \in \gamma(\alpha(v))}}{\langle \mathtt{concretize}(\mathtt{id}), s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(\mathtt{id}), s\$abst\rangle \to_e^* v}$$

When $\mathtt{id} \notin \mathit{IndicesAbst}$, $Tr_{read}(\mathtt{id}) \to \mathtt{id}$. Given $s\$abst|s$ and $\mathtt{id} \notin \mathit{IndicesAbst}$, we have $s\$abst\langle \mathtt{id}\rangle = s\langle \mathtt{id}\rangle = v$. Then:

$$\frac{s\$abst\langle \mathtt{id}\rangle = v}{\langle \mathtt{id}, s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(\mathtt{id}), s\$abst\rangle \to_e^* v}$$

Constant Next we prove the lamma for the case where `exp` is `constant`:

$$\frac{\langle \mathtt{constant}, s\rangle \to_e^* v}{v = \mathtt{constant}}$$

Note that $Tr_{read}(\mathtt{constant}) \to \mathtt{constant}$,

$$\frac{v = \mathtt{constant}}{\langle \mathtt{constant}, s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(\mathtt{id}), s\$abst\rangle \to_e^* v}$$

Pointer dereference For cases with pointer dereference, we have:

$$\frac{\langle *\mathtt{exp}, s\rangle \to_e^* v}{\exists ptr, \langle \mathtt{exp}, s\rangle \to_e^* ptr, s\langle \mathtt{memory}\rangle[ptr.\mathtt{base}][ptr.\mathtt{offset}] = v}$$

Given that $s\$abst|s$, we have $\langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr$. We prove the lemma in 3 cases. The target expression is

$$Tr_{read}(*\mathtt{exp}) \to \mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp}))?\mathtt{abst\_deref}(Tr_{read}(\mathtt{exp})) : *Tr_{read}(\mathtt{exp})$$

Case 1 ($ptr.\mathtt{base} \notin s\$abst\langle \mathtt{abstmem}\rangle$): since $s\$abst|s$, we have

$$s\$abst\langle \mathtt{memory}\rangle[ptr.\mathtt{base}][ptr.\mathtt{offset}] = s\langle \mathtt{memory}\rangle[ptr.\mathtt{base}][ptr.\mathtt{offset}] = v$$

Then:

$$\frac{\dfrac{\langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\mathtt{base} \notin s\$abst\langle \mathtt{abstmem}\rangle}{\langle \mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp})), s\$abst\rangle \to_e^* \mathtt{false},}, \dfrac{\langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr, s\$abst\langle \mathtt{memory}\rangle[ptr.\mathtt{base}][ptr.\mathtt{offset}] = v}{\langle *Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* v}}{\langle \mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp}))?\mathtt{abst\_deref}(Tr_{read}(\mathtt{exp})) : *Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(*\mathtt{exp}), s\$abst\rangle \to_e^* v}$$

Case 2 ($ptr.\mathtt{base} \in s\$abst\langle \mathtt{abstmem}\rangle$ and $\beta(ptr.\mathtt{offset}) = \mathtt{false}$):

$$\frac{\dfrac{\langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\mathtt{base} \in s\$abst\langle \mathtt{abstmem}\rangle}{\langle \mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp})), s\$abst\rangle \to_e^* \mathtt{true},}, \dfrac{\langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr, \beta(ptr.\mathtt{offset}) = \mathtt{false}}{\langle \mathtt{abst\_deref}(Tr_{read}(\mathtt{exp})), s\$abst\rangle \to_e^* v}}{\langle \mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp}))?\mathtt{abst\_deref}(Tr_{read}(\mathtt{exp})) : *Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(*\mathtt{exp}), s\$abst\rangle \to_e^* v}$$

Case 3 ($ptr.\mathtt{base} \in s\$abst\langle\mathtt{abstmem}\rangle$ and $\beta(ptr.\mathtt{offset}) = \mathtt{true}$): since $s\$abst|s$ and $ptr.\mathtt{base} \in s\$abst\langle\mathtt{abstmem}\rangle$ and $\beta(ptr.\mathtt{offset}) = \mathtt{true}$, we have

$$s\$abst\langle\mathtt{memory}\rangle[ptr.\mathtt{base}][\alpha(ptr.\mathtt{offset})] = s\langle\mathtt{memory}\rangle[ptr.\mathtt{base}][ptr.\mathtt{offset}] = v$$

Then,

$$\frac{\dfrac{\langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\mathtt{base} \in s\$abst\langle\mathtt{abstmem}\rangle}{\langle\mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp})), s\$abst\rangle \to_e^* \mathtt{true},} , \dfrac{\beta(ptr.\mathtt{offset}) = \mathtt{true}, \langle Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* ptr, s\$abst\langle\mathtt{memory}\rangle[ptr.\mathtt{base}][\alpha(ptr.\mathtt{offset})] = v}{\langle\mathtt{abst\_deref}(Tr_{read}(\mathtt{exp})), s\$abst\rangle \to_e^* v}}{\langle\mathtt{in\_abstmem}(Tr_{read}(\mathtt{exp}))?\mathtt{abst\_deref}(Tr_{read}(\mathtt{exp})) : *Tr_{read}(\mathtt{exp}), s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(*\mathtt{exp}), s\$abst\rangle \to_e^* v}$$

Plus/minus We want to prove the lemma for $\mathtt{exp}_1 \pm \mathtt{exp}_2$ separately for the case "$\mathtt{pointer} \pm \mathtt{uint}$" and the case"$\mathtt{uint} \pm \mathtt{uint}$". The target expression is

$$Tr_{read}(\mathtt{exp}_1 \pm \mathtt{exp}_2) \to Tr_{read}(\mathtt{exp}_1) \pm Tr_{read}(\mathtt{exp}_2)$$

When $\mathtt{exp}_1$ is a $\mathtt{pointer}$ (inferring $Tr_{read}(\mathtt{exp}_1)$ is also a $\mathtt{pointer}$):

$$\frac{\langle\mathtt{exp}_1 \pm \mathtt{exp}_2, s\rangle \to_e^* v}{\exists ptr, v_2 \text{ s.t. } \langle\mathtt{exp}_1, s\rangle \to_e^* ptr, \langle\mathtt{exp}_2, s\rangle \to_e^* v_2, v = \mathtt{pointer}(ptr.\mathtt{base}, ptr.\mathtt{offset} \pm v_2)}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\mathtt{exp}_1), s\$abst\rangle \to_e^* ptr$ and $\langle Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v_2$. Then,

$$\frac{\langle Tr_{read}(\mathtt{exp}_1), s\$abst\rangle \to_e^* ptr, \langle Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v_2, v = \mathtt{pointer}(ptr.\mathtt{base}, ptr.\mathtt{offset} \pm v_2)}{\langle Tr_{read}(\mathtt{exp}_1) \pm Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v, \text{ i.e. } \langle Tr_{read}(\mathtt{exp}_1 \pm \mathtt{exp}_2), s\$abst\rangle \to_e^* v} \quad , Tr_{read}(\mathtt{exp}_1) \text{ is a } \mathtt{pointer}$$

When $\mathtt{exp}_1$ is an $\mathtt{uint}$ (inferring $Tr_{read}(\mathtt{exp}_1)$ is also an $\mathtt{uint}$):

$$\frac{\langle\mathtt{exp}_1 \pm \mathtt{exp}_2, s\rangle \to_e^* v}{\exists v_1, v_2 \text{ s.t. } \langle\mathtt{exp}_1, s\rangle \to_e^* v_1, \langle\mathtt{exp}_2, s\rangle \to_e^* v_2, v = v_1 \pm v_2)}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\mathtt{exp}_1), s\$abst\rangle \to_e^* v_1$ and $\langle Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v_2$. Then,

$$\frac{\langle Tr_{read}(\mathtt{exp}_1), s\$abst\rangle \to_e^* v_1, \langle Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v_2, v = v_1 \pm v_2}{\langle Tr_{read}(\mathtt{exp}_1) \pm Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v, \text{ i.e. } \langle Tr_{read}(\mathtt{exp}_1 \pm \mathtt{exp}_2), s\$abst\rangle \to_e^* v} \quad , Tr_{read}(\mathtt{exp}_1) \text{ is an } \mathtt{uint}$$

Relation operators Similar to plus/minus, $Tr_{read}(\mathtt{exp}_1 \text{ rel } \mathtt{exp}_2) \to Tr_{read}(\mathtt{exp}_1) \text{ rel } Tr_{read}(\mathtt{exp}_2)$.

$$\frac{\langle\mathtt{exp}_1 \text{ rel } \mathtt{exp}_2, s\rangle \to_e^* v}{\exists v_1, v_2 \text{ s.t.} \langle\mathtt{exp}_1, s\rangle \to_e^* v_1, \langle\mathtt{exp}_2, s\rangle \to_e^* v_2, v = v_1 \text{ rel } v_2}$$

Given $s\$abst|s$, we have $\langle Tr_{read}(\mathtt{exp}_1), s\$abst\rangle \to_e^* v_1$ and $\langle Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v_2$. Then,

$$\frac{\langle Tr_{read}(\mathtt{exp}_1), s\$abst\rangle \to_e^* v_1, \langle Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v_2, v = v_1 \text{ rel } v_2}{\langle Tr_{read}(\mathtt{exp}_1) \text{ rel } Tr_{read}(\mathtt{exp}_2), s\$abst\rangle \to_e^* v, \text{i.e.} \langle Tr_{read}(\mathtt{exp}_1 \text{ rel } \mathtt{exp}_2), s\$abst\rangle \to_e^* v}$$

Conditional expression Conditional expression are translated to

$$Tr_{read}(\texttt{exp}_1?\texttt{exp}_2:\texttt{exp}_3) \rightarrow Tr_{read}(\texttt{exp}_1)?Tr_{read}(\texttt{exp}_2):Tr_{read}(\texttt{exp}_3)$$

We know that:

$$\frac{\langle \texttt{exp}_1?\texttt{exp}_2:\texttt{exp}_3, s\rangle \rightarrow_e^* v}{(\exists v, \langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* v) \text{ or } (\exists v, \langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \rightarrow_e^* v)}$$

We separately prove the lemma for two cases. For the first case where $\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* v$, given $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{true}$ and $\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* v$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{true}, \langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* v}{\langle Tr_{read}(\texttt{exp}_1)?Tr_{read}(\texttt{exp}_2):Tr_{read}(\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* v, \text{ i.e. } \langle Tr_{read}(\texttt{exp}_1?\texttt{exp}_2:\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* v}$$

For the second case ($\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \rightarrow_e^* v$), with $s\$abst|s$, we got $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{false}$ and $\langle Tr_{read}(\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* v$. Therefore,

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{false}, \langle Tr_{read}(\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* v}{\langle Tr_{read}(\texttt{exp}_1)?Tr_{read}(\texttt{exp}_2):Tr_{read}(\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* v, \text{ i.e. } \langle Tr_{read}(\texttt{exp}_1?\texttt{exp}_2:\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* v}$$

Pointer validation checkers Pointer checkers are translated to:

$$Tr_{read}(\texttt{valid(exp)}) \rightarrow \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})):\texttt{valid}(Tr_{read}(\texttt{exp}))$$

We will prove lemma 1 for the true case and the false case. In the true case:

$$\frac{\langle \texttt{valid(exp)}, s\rangle \rightarrow_e^* \texttt{true}}{\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\langle\texttt{memory}\rangle, ptr.\texttt{offset} < s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr$, $ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle$, and $ptr.\texttt{offset} < s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}$.

When $ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$, we got

$$ptr.\texttt{offset} < s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length} = s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length} = s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$$

Then,

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{false}}, \dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle, ptr.\texttt{offset} < s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{true}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})):\texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{true} \text{ i.e. } \langle Tr_{read}(\texttt{valid(exp)}), s\$abst\rangle \rightarrow_e^* \texttt{true}}$$

When $ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$, we got

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{true}}, \dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle, ptr.\texttt{offset} < s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}}{\langle \texttt{abst\_valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{true}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})):\texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{true} \text{ i.e. } \langle Tr_{read}(\texttt{valid(exp)}), s\$abst\rangle \rightarrow_e^* \texttt{true}}$$

Next, we prove the false case. We know that:

$$\frac{\langle \texttt{valid(exp)}, s\rangle \rightarrow_e^* \texttt{false}}{(\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\langle\texttt{memory}\rangle) \text{ or } (\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length})}$$

In the first case where "$\exists ptr$ s.t. $\langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\langle\texttt{memory}\rangle$", since $s\$abst|s$, we got $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr$ and $ptr.\texttt{base} \notin s\$abst\langle\texttt{memory}\rangle$. When $ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$,

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}} \quad , \quad \dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{memory}\rangle}{\langle \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})) : \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false} \text{ i.e. } \langle Tr_{read}(\texttt{valid}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}$$

When $ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{true}} \quad , \quad \dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{memory}\rangle}{\langle \texttt{abst\_valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})) : \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false} \text{ i.e. } \langle Tr_{read}(\texttt{valid}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}$$

In the second case where "$\exists ptr$ s.t. $\langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$", since $s\$abst|s$, we got $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr$, $ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle$ and $ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}$. When $ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$, we got

$$ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length} = s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$$

Then,

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}} \quad , \quad \dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})) : \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false} \text{ i.e. } \langle Tr_{read}(\texttt{valid}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}$$

When $ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{true}} \quad , \quad \dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}}{\langle \texttt{abst\_valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_valid}(Tr_{read}(\texttt{exp})) : \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false} \text{ i.e. } \langle Tr_{read}(\texttt{valid}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{false}}$$

4.1.2 With valuation failures

Pointer dereference From a pointer dereference in the original program, we know that:

$$\frac{\langle *\texttt{exp}, s\rangle \to_e^* \texttt{fail}}{(\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}) \text{ or } (\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{false})}$$

In the first case where $\langle \texttt{exp}, s\rangle \to_e^* \texttt{fail}$, since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* \texttt{fail}$. Then,

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* \texttt{fail}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \to_e^* \texttt{fail}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_deref}(Tr_{read}(\texttt{exp})) : *Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* \texttt{fail}, \text{i.e.} \langle Tr_{read}(*\texttt{exp}), s\$abst\rangle \to_e^* \texttt{fail}}$$

In the second case where $\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{false}$,

$$\frac{\langle \texttt{valid}(\texttt{exp}), s\rangle \to_e^* \texttt{false}}{(\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\langle\texttt{memory}\rangle) \text{ or } (\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \to_e^* ptr, ptr.\texttt{base} \in s\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length})}$$

When "$\langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\langle\texttt{memory}\rangle$", since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr$ and $ptr.\texttt{base} \notin s\$abst\langle\texttt{memory}\rangle$. Then,

$$\cfrac{\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{memory}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_deref}(Tr_{read}(\texttt{exp})) : *Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}, \text{i.e.} \langle Tr_{read}(*\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}}$$

When "$\langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$", since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr$, $ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle$ and $ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}$. To prove this case, we have 2 subcases.

In subcase 1 where $ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$, given $s\$abst|s$, we got:

$$ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length} = s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$$

Then,

$$\cfrac{\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{false}}, \quad \cfrac{\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{false}}}{\langle *Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_deref}(Tr_{read}(\texttt{exp})) : *Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}, \text{i.e.} \langle Tr_{read}(*\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}}$$

In subcase 2 where $ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$,

$$\cfrac{\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{true}}, \quad \cfrac{\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}}{\langle \texttt{abst\_valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{false}}}{\langle \texttt{abst\_deref}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}))?\texttt{abst\_deref}(Tr_{read}(\texttt{exp})) : *Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}, \text{i.e.} \langle Tr_{read}(*\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}}$$

Plus/minus Given $\langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}$,

$$\frac{\langle \texttt{exp}_1 \pm \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}}{(\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail})}$$

When $\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}$, given $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle Tr_{read}(\texttt{exp}_1) \pm Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* \texttt{fail} \text{ i.e. } \langle Tr_{read}(\texttt{exp}_1 \pm \texttt{exp}_2)\rangle \rightarrow_e^* \texttt{fail}}$$

Proof is similar when $\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}$.

Relation operators Proof is similar to plus/minus.

Conditional expressions Given $\langle \texttt{exp}_1?\texttt{exp}_2 : \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}$,

$$\frac{\langle \texttt{exp}_1?\texttt{exp}_2 : \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail}}{(\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{false}, \langle \texttt{exp}_3, s\rangle \rightarrow_e^* \texttt{fail})}$$

When $\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}$, since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle Tr_{read}(\texttt{exp}_1)?Tr_{read}(\texttt{exp}_2) : Tr_{read}(\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* \texttt{fail} \text{ i.e. } \langle Tr_{read}(\texttt{exp}_1?\texttt{exp}_2 : \texttt{exp}_3), s\$abst\rangle \rightarrow_e^* \texttt{fail}}$$

When $\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{true}, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{true}$ and $\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* \texttt{fail}$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{true}, \langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle Tr_{read}(\texttt{exp}_1)?Tr_{read}(\texttt{exp}_2) : Tr_{read}(\texttt{exp}_3), s\$abst\rangle \rightarrow_e^* \texttt{fail} \text{ i.e. } \langle Tr_{read}(\texttt{exp}_1?\texttt{exp}_2 : \texttt{exp}_3), s\$abst\rangle \rightarrow_e^* \texttt{fail}}$$

The third case is similar to the second case.

Pointer validation checks Given $\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}$,

$$\frac{\langle \texttt{valid}(\texttt{exp}), s\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{valid}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{fail} \text{ i.e. } \langle Tr_{read}(\texttt{valid}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{fail}}$$

4.2 Lemma 2

The following lemma (Lemma 2) ensures trace containment when we abstract a program:

If $\langle \text{inst}_1, s_1\rangle \rightarrow \langle \text{inst}_2, s_2\rangle$ and $s_1\$abst|s_1$, $\exists s_2\$abst$ such that $s_2\$abst|s_2$ and $\langle Tr(\text{inst}_1), s_1\$abst\rangle \rightarrow^* \langle Tr(\text{inst}_2), s_2\$abst\rangle$.

We prove this lemma first in cases where no failure appears (except for assertion/assumptions) and then in cases where instruction fails.

4.2.1 Without failures

For loops The original for loop instruction is "for $\texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2$ do instrs". We split into 2 cases: $\texttt{id} \notin IndicesAbst$ and $\texttt{id} \in IndicesAbst$.

Case 1 When $\texttt{id} \notin IndicesAbst$, the abstracted version is:

$$Tr(\text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}) \rightarrow \text{for } \texttt{id} := Tr_{read}(\texttt{exp}_1) \ldots Tr_{read}(\texttt{exp}_2) \text{ do } Tr(\text{instrs})$$

Two possible transition can happen in the original program:

$$\begin{array}{ll} \langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}, s\rangle \rightarrow \langle \text{instrs}; \text{for } \texttt{id} := start + 1 \ldots end \text{ do instrs}, s\{\texttt{id} \mapsto start\}\rangle & \text{loop continues} \\ \langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}, s\rangle \rightarrow \langle \text{skip}, s\{\texttt{id} \mapsto start\}\rangle & \text{loop finishes} \end{array}$$

When the loop continues, we got:

$$\frac{\langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}, s\rangle \rightarrow \langle \text{insts}; \text{for } \texttt{id} := start + 1 \ldots end \text{ do insts}, s\{\texttt{id} \mapsto start\}\rangle}{\langle \texttt{exp}_1, s\rangle \rightarrow_e^* start, \langle \texttt{exp}_2, s\rangle \rightarrow_e^* end, start \leq end}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* start$ and $\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* end$. Then:

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* start, \langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* end, start \leq end}{\langle \text{for } \texttt{id} := Tr_{read}(\texttt{exp}_1) \ldots Tr_{read}(\texttt{exp}_2) \text{ do instrs}, s\$abst\rangle \rightarrow \langle Tr(\text{instrs}); \text{for } \texttt{id} := start + 1 \ldots end \text{ do } Tr(\text{instrs}), s\$abst\{\texttt{id} \mapsto start\}\rangle}$$

Note that $start+1$ and end are constants, so $Tr_{read}(start+1) \to start+1$ and $Tr_{read}(end) \to end$. Given $\texttt{id} \notin IndicesAbst$ and $s\$abst|s$, we also have $s\$abst\{\texttt{id} \mapsto start\}|s\{\texttt{id} \mapsto start\}$. Therefore we proved the lemma when loop continues.

When the loop finishes, we got:

$$\frac{\langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto start\}\rangle}{\langle \texttt{exp}_1, s\rangle \to_e^* start, \langle \texttt{exp}_2, s\rangle \to_e^* end, start > end}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* start$ and $\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* end$. Then:

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* start, \langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* end, start > end}{\langle \text{for } \texttt{id} := Tr_{read}(\texttt{exp}_1) \ldots Tr_{read}(\texttt{exp}_2) \text{ do instrs}, s\$abst\rangle \to \langle \text{skip}, s\$abst\{\texttt{id} \mapsto start\}\rangle}$$

We got $Tr(\text{skip}) \to skip$. Since $s\$abst|s$ and $\texttt{id} \notin IndicesAbst$, we also have $s\$abst\{\texttt{id} \mapsto start\}|s\{\texttt{id} \mapsto start\}$. Therefore the lemma is proved when the loop finishes in this case.

Case 2 When $\texttt{id} \in IndicesAbst$, the abstracted version is:

$$Tr(\text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}) \to \text{abstfor } \texttt{id} := \texttt{abstract}(Tr_{read}(\texttt{exp}_1)) \ldots \texttt{abstract}(Tr_{read}(\texttt{exp}_2)) \text{ do } Tr(\text{instrs})$$

When the loop continues, we got:

$$\frac{\langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}, s\rangle \to \langle \text{instrs}; \text{for } \texttt{id} := start+1 \ldots end \text{ do insts}, s\{\texttt{id} \mapsto start\}\rangle}{\langle \texttt{exp}_1, s\rangle \to_e^* start, \langle \texttt{exp}_2, s\rangle \to_e^* end, start \leq end}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* start$ and $\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* end$. Then:

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* start}{\langle \texttt{abstract}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \to_e^* \alpha(start)}, \dfrac{\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* end}{\langle \texttt{abstract}(Tr_{read}(\texttt{exp}_2))\rangle \to_e^* \alpha(end)}, \dfrac{}{start \in \gamma(\alpha(start))}, \dfrac{}{end \in \gamma(\alpha(end))}, start \leq end}{\begin{array}{l}\langle \text{abstfor } \texttt{id} : \texttt{abstract}(Tr_{read}(\texttt{exp}_1)) \ldots \texttt{abstract}(Tr_{read}(\texttt{exp}_2)) \text{ do } Tr(\text{instrs}), s\$abst\rangle \to \\ \langle Tr(\text{instrs}); \text{abstfor } \texttt{id} := \texttt{abstract}(start+1) \ldots \texttt{abstract}(end) \text{ do } Tr(\text{instrs}), s\$abst\{\texttt{id} \mapsto \alpha(start)\}\rangle\end{array}}$$

Because $Tr_{read}(start+1) \to start+1$ and $Tr_{read}(end) \to end$, we know that

$$Tr(\text{instrs}; \text{for } \texttt{id} := start+1 \ldots end \text{ do insts}) \to \langle Tr(\text{instrs}); \text{abstfor } \texttt{id} := \texttt{abstract}(start+1) \ldots \texttt{abstract}(end) \text{ do } Tr(\text{instrs})$$

Moreover, given $s\$abst|s$ and $\texttt{id} \in IndicesAbst$, $s\$abst\{\texttt{id} \mapsto \alpha(start)\}|s\{\texttt{id} \mapsto start\}$. We proved the lemma in this case.

When the loop finishes, we got:

$$\frac{\langle \text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do instrs}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto start\}\rangle}{\langle \texttt{exp}_1, s\rangle \to_e^* start, \langle \texttt{exp}_2, s\rangle \to_e^* end, start > end}$$

Since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* start$ and $\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* end$. Then:

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* start}{\langle \texttt{abstract}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \to_e^* \alpha(start)}, \dfrac{\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* end}{\langle \texttt{abstract}(Tr_{read}(\texttt{exp}_2))\rangle \to_e^* \alpha(end)}, \dfrac{}{start \in \gamma(\alpha(start))}, \dfrac{}{end \in \gamma(\alpha(end))}, start > end}{\begin{array}{l}\langle \text{abstfor } \texttt{id} : \texttt{abstract}(Tr_{read}(\texttt{exp}_1)) \ldots \texttt{abstract}(Tr_{read}(\texttt{exp}_2)) \text{ do } Tr(\text{instrs}), s\$abst\rangle \to \\ \langle \text{skip}, s\$abst\{\texttt{id} \mapsto \alpha(start)\}\rangle\end{array}}$$

Similar to the case where loop continues, we have $s\$abst\{\texttt{id} \mapsto \alpha(start)\}|s\{\texttt{id} \mapsto start\}$. Therefore we proved the lemma in this case.

Variable assigns We have:

$$\frac{\langle \text{assign } \texttt{id}\ \texttt{exp}, s\rangle \to \langle \text{skip}, s\{\texttt{id} \mapsto v\}\rangle}{\langle \texttt{exp}, s\rangle \to_e^* v}$$

Note that we do not allow writing to loop indices, i.e. $\texttt{id} \notin \mathit{IndicesAbst}$. Given $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* v$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* v}{\langle \text{assign } \texttt{id}\ Tr_{read}(\texttt{exp}), s\$abst\rangle \to \langle \text{skip}, s\$abst\{\texttt{id} \mapsto v\}\rangle, \text{i.e.} \langle Tr(\texttt{id} := \texttt{exp}), s\$abst\rangle \to \langle Tr(\text{skip}), s\$abst\{\texttt{id} \mapsto v\}\rangle}$$

We showed in earlier sections that $s\$abst\{\texttt{id} \mapsto v\}|s\{\texttt{id} \mapsto v\}$ if $\texttt{id} \notin \mathit{IndicesAbst}$. Therefore we proved the lemma.

Memory assigns For assignment in the form assignmem $\texttt{exp}_1$ $\texttt{exp}_2$, first of all, we have:

$$\frac{\langle \text{assignmem } \texttt{exp}_1\ \texttt{exp}_2, s\rangle \to \langle \text{skip}, s\{\texttt{memory}[base][offset] \mapsto v\}\rangle}{\exists ptr \text{ s.t. } \langle \texttt{exp}_1, s\rangle \to_e^* ptr, \langle \texttt{exp}_2, s\rangle \to_e^* v, ptr.\texttt{base} = base, ptr.\texttt{offset} = offset}$$

Given that $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\rangle \to_e^* ptr$,$\langle Tr_{read}(\texttt{exp}_2), s\rangle \to_e^* v$. Going from there, we want to split into 2 cases and prove the lemma for each case.

Case 1 ($ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$):

$$\frac{\dfrac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \to_e^* \texttt{false}} \quad , \langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* ptr}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp}_1)) : Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* ptr} \quad , \langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* v}{\langle \text{assignmem } \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp}_1)) : Tr_{read}(\texttt{exp}_1)\ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to \langle \text{skip}, s\$abst\{\texttt{memory}[ptr.\texttt{base}][ptr.\texttt{offset}] \mapsto v\}\rangle}$$

Since $ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$, $s\$abst\{\texttt{memory}[ptr.\texttt{base}][ptr.\texttt{offset}] \mapsto v\}$ is an abstraction of $s\{\texttt{memory}[ptr.\texttt{base}][ptr.\texttt{offset}] \mapsto v\}$ given $s\$abst|s$. We proved the lemma in this case.

Case 2 ($ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$):

$$\frac{\dfrac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \to_e^* \texttt{true}} \quad , \dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* ptr}{\langle \texttt{abst\_ptr}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \to_e^* \texttt{pointer}(ptr.\texttt{base}, \alpha(ptr.\texttt{offset}))}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp}_1)) : Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* \texttt{pointer}(ptr.\texttt{base}, \alpha(ptr.\texttt{offset}))} \quad , \langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to_e^* v}{\langle \text{assignmem } \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp}_1)) : Tr_{read}(\texttt{exp}_1)\ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \to \langle \text{skip}, s\$abst\{\texttt{memory}[ptr.\texttt{base}][\alpha(ptr.\texttt{offset})] \mapsto v\}\rangle}$$

Since $ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$m we show in the previous session that $s\$abst\{\texttt{memory}[ptr.\texttt{base}][\alpha(ptr.\texttt{offset})] \mapsto v\}$ is an abstraction of $s\{\texttt{memory}[ptr.\texttt{base}][ptr.\texttt{offset}] \mapsto v\}$ given $s\$abst|s$. We proved the lemma in this case.

Mallocs For malloc, we have:

$$\frac{\langle \text{malloc } \texttt{id}\ \texttt{exp}, s\rangle \to \langle \text{skip}, s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v), id \mapsto \texttt{pointer}(l_{new}, 0)\}\rangle}{\text{s.t. } l_{new} = \texttt{newloc}(s\langle\texttt{memory}\rangle), \langle \texttt{exp}, s\rangle \to_e^* v}$$

Since $s\$abst|s$, we have $l_{new} = \texttt{newloc}(s\$abst\langle\texttt{memory}\rangle)$,$\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* v$. Then we divide the proof into two cases. One for $\texttt{id} \notin \mathit{PointersAbst}$ and another for $\texttt{id} \in \mathit{PointersAbst}$.

Case 1 ($\texttt{id} \notin \mathit{PointersAbst}$):

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* v, l_{new} = \texttt{newloc}(s\$abst\langle\texttt{memory}\rangle)}{\langle \text{malloc } \texttt{id}\ Tr_{read}(\texttt{exp}), s\$abst\rangle \to \langle \text{skip}, s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v, v), \texttt{id} \mapsto \texttt{pointer}(l_{new}, 0)\}\rangle}$$

Since $s\$abst|s$, we have $s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v,v)\}|s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v)\}$. $\texttt{id}$ is a $\texttt{pointer}$ which means $\texttt{id} \notin \mathit{IndicesAbst}$. Therefore $s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v,v), \texttt{id} \mapsto \texttt{pointer}(l_{new},0)\}$ is also an abstraction of $s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v), \texttt{id} \mapsto \texttt{pointer}(l_{new},0)\}$. We proved the lemma in this case.

Case 2 ($\texttt{id} \in \mathit{PointersAbst}$):

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* v, l_{new} = (newloc)(s\$abst\langle\texttt{memory}\rangle)}{\langle\text{abstmalloc } \texttt{id}\ Tr_{read}(\texttt{exp}), s\$abst\rangle \to \langle\text{skip}, s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\alpha(v),v), \texttt{id} \mapsto \texttt{pointer}(l_{new},0), \texttt{abstmem.add}(l_{new})\}\rangle}$$

Given our condition $s\$abst|s$, $s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\alpha(v),v), \texttt{abstmem.add}(l_{new})\}$ is an abstraction of $s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v)\}$. Since $\texttt{id}$ is a $\texttt{pointer}$ ($\texttt{id} \notin \mathit{IndicesAbst}$), $s\$abst\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(\alpha(v),v), \texttt{id} \mapsto \texttt{pointer}(l_{new},0), \texttt{abstmem.add}(l_{new})\}$ is an abstraction of $s\{\texttt{memory}[l_{new}] \mapsto \texttt{array}(v), \texttt{id} \mapsto \texttt{pointer}(l_{new},0)\}$. We proved the lemma in this case.

Asserts For assertinos, in the original program, we have two cases. In one case the test goes through and the other case it fails. We will talk about the two cases and prove the lemma separately.

For the first case where it goes through:

$$\frac{\langle\text{assert } \texttt{exp}, s\rangle \to \langle\text{skip}, s\rangle}{\langle\texttt{exp}, s\rangle \to_e^* \texttt{true}}$$

Since $s\$abst|s$, in the abstracted program, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* \texttt{true}$. Then:

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \to_e^* \texttt{true}}{\langle\text{assert } Tr_{read}(\texttt{exp}), s\$abst\rangle \to \langle\text{skip}, s\$abst\rangle}$$

Since $s\$abst|s$, we proved the lemma in this case. The other case where the assertion fail, it might be failure of expression valuation or assertion failure. We will prove this case in next section.

Assumes Proofs for assumptions are similar to ones for assertions since they have the same semantics at program model level.

Skips Skip will be transformed to skip and it does not change the state. Therefore proof of this case is trivial.

4.2.2 With failures

For most types of instructions failure comes from failed evaluations of expressions shown up in the instruction. Proof for lemma 2 is similar for those instructions. The special cases are the "assignmem" instruction (need to consider whether the target location is valid), "assert/assume" (failure also from false valuation). We will only prove one of the regular ones.

For loops When a for loop fails, we know:

$$\frac{\langle\text{for } \texttt{id} := \texttt{exp}_1 \ldots \texttt{exp}_2 \text{ do insts}, s\rangle \to \langle\text{fail}, s\rangle}{(\langle\texttt{exp}_1, s\rangle \to_e^* \texttt{fail}) \text{ or } (\langle\texttt{exp}_2, s\rangle \to_e^* \texttt{fail})}$$

In the first case where $\langle\texttt{exp}_1, s\rangle \to_e^* \texttt{fail}$, since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* \texttt{fail}$. Then, when $\texttt{id} \notin \mathit{IndicesAbst}$

$$\frac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* \texttt{fail}}{\langle\text{for } \texttt{id} := Tr_{read}(\texttt{exp}_1) \ldots Tr_{read}(\texttt{exp}_2) \text{ do } Tr(\text{insts}), s\$abst\rangle \to \langle\text{fail}, s\$abst\rangle}$$

When $\texttt{id} \in \mathit{IndicesAbst}$,

$$\frac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \to_e^* \texttt{fail}}{\langle\texttt{abstract}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \to_e^* \texttt{fail}}}{\langle\text{abstfor } \texttt{id} := \texttt{abstract}(Tr_{read}(\texttt{exp}_1)) \ldots \texttt{abstract}(Tr_{read}(\texttt{exp}_2)) \text{ do } Tr(\text{insts}), s\$abst\rangle \to \langle\text{fail}, s\$abst\rangle}$$

The other case ($\langle\texttt{exp}_2, s\rangle \to_e^* \texttt{fail}$) can be proved similarly.

Mallocs, assigns Similar to the proof for loops.

Assertions

$$\frac{\langle \text{assert } \texttt{exp}, s\rangle \rightarrow \langle \text{fail}, s\rangle}{(\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{false}) \text{ or } (\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail})}$$

In the first case ($\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{false}$), since $s\$abst|s$, in the abstracted program, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{false}$. Then:

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{false}}{\langle \text{assert } Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow \langle \text{fail}, s\$abst\rangle}$$

In the second case ($\langle \texttt{exp}, s\rangle \rightarrow_e^* \texttt{fail}$), since $s\$abst|s$, in the abstracted program, we have $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}$. Then:

$$\frac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle \text{assert } Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow \langle \text{fail}, s\$abst\rangle}$$

Assumptions Proof is similar to the one for assertion.

Assign to memorys For assignmem failures,

$$\frac{\langle \text{assignmem } \texttt{exp}_1 \ \texttt{exp}_2, s\rangle \rightarrow \langle \text{fail}, s\rangle}{(\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}) \text{ or } (\langle \texttt{valid}(\texttt{exp}_1), s\rangle \rightarrow_e^* \texttt{false})}$$

In the first case ($\langle \texttt{exp}_1, s\rangle \rightarrow_e^* \texttt{fail}$), since $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_1)\rangle \rightarrow_e^* \texttt{fail}$. Then,

$$\frac{\dfrac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp})) : Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail} \text{ i.e. } \langle Tr_{write}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \text{assignmem } Tr_{write}(\texttt{exp}_1) \ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow \langle \text{fail}, s\$abst\rangle}$$

In the second case ($\langle \texttt{exp}_2, s\rangle \rightarrow_e^* \texttt{fail}$), given $s\$abst|s$, we have $\langle Tr_{read}(\texttt{exp}_2)\rangle \rightarrow_e^* \texttt{fail}$. Then,

$$\frac{\langle Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow_e^* \texttt{fail}}{\langle \text{assignmem } Tr_{write}(\texttt{exp}_1) \ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow \langle \text{fail}, s\$abst\rangle}$$

In the third case where $\langle \texttt{valid}(\texttt{exp}_1), s\rangle \rightarrow_e^* \texttt{false}$,

$$\frac{\texttt{valid}(\texttt{exp}_1), s\rangle \rightarrow_e^* \texttt{false}}{(\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\langle \texttt{memory}\rangle) \text{ or } (\exists ptr \text{ s.t. } \langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\langle \texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle \texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length})}$$

When $\langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\langle \texttt{memory}\rangle$, since $s\$abst|s$, we have:

$$\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle \texttt{memory}\rangle$$

Then:

$$\frac{\dfrac{\dfrac{\langle Tr_{read}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \notin s\$abst\langle \texttt{memory}\rangle}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp})) : Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{fail} \text{ i.e. } \langle Tr_{write}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{fail}}}{\langle \text{assignmem } Tr_{write}(\texttt{exp}_1) \ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow \langle \text{fail}, s\$abst\rangle}$$

When $\langle \texttt{exp}, s\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$, since $s\$abst|s$, we have

$$\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{memory}\rangle, ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length}$$

We split into 2 subcases. In the first subcase ($ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle$), we have $ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_length} = s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}$. We first prove that

$$\cfrac{\cfrac{ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle}{\langle\texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \rightarrow_e^* \texttt{false}}, \langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr}{\langle\texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp})) : Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr \text{ i.e. } \langle Tr_{write}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* ptr}$$

Then,

$$\cfrac{\cfrac{ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle, \langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr}{\langle Tr_{write}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* ptr}, ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle, ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{length}}{\langle\text{assignmem } Tr_{write}(\texttt{exp}_1)\ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow \langle\text{fail}, s\$abst\rangle}$$

In the second subcase where $ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle$, First, we have:

$$\cfrac{\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle}{\langle\texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1)), s\$abst\rangle \rightarrow_e^* \texttt{true}}, \cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr}{\langle\texttt{abst\_ptr}(Tr_{read}(\texttt{exp})), s\$abst\rangle \rightarrow_e^* \texttt{pointer}(ptr.\texttt{base}, \alpha(ptr.\texttt{offset}))}}{\langle\texttt{in\_abstmem}(Tr_{read}(\texttt{exp}_1))?\texttt{abst\_ptr}(Tr_{read}(\texttt{exp})) : Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{pointer}(ptr.\texttt{base}, \alpha(ptr.\texttt{offset})) \text{ i.e. } \langle Tr_{write}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{pointer}(ptr.\texttt{base}, \alpha(ptr.\texttt{offse}}$$

Then, we got

$$\cfrac{\cfrac{ptr.\texttt{base} \notin s\$abst\langle\texttt{abstmem}\rangle, \langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* ptr}{\langle Tr_{write}(\texttt{exp}_1), s\$abst\rangle \rightarrow_e^* \texttt{pointer}(ptr.\texttt{base}, \alpha(ptr.\texttt{offset}))}, ptr.\texttt{base} \in s\$abst\langle\texttt{abstmem}\rangle, \cfrac{}{ptr.\texttt{offset} \in \gamma(\alpha(ptr.\texttt{offset}))}, ptr.\texttt{offset} \geq s\$abst\langle\texttt{memory}\rangle[ptr.\texttt{base}].\texttt{conc\_lengt}}{\langle\text{assignmem } Tr_{write}(\texttt{exp}_1)\ Tr_{read}(\texttt{exp}_2), s\$abst\rangle \rightarrow \langle\text{fail}, s\$abst\rangle}$$

4.3 Theorem 1

The soundness theorem says that

> If `assert(exp)` fails in `P` then `assert(`Tr_{read}`(exp))` can fail in `P$abst`

Based on Lemma 1 and 2, Consider the execution before the assertion failure is

$$\langle\text{instr}_0, init\rangle \rightarrow \langle\text{instr}_1, s_1\rangle \rightarrow \cdots \rightarrow \langle\text{instr}_n, s_n\rangle \rightarrow \langle\text{assert } \texttt{exp}, s\rangle$$

Given $init\$abst|init$, because of Lemma 2, we can find a path in the abstracted program:

$$\langle Tr(\text{instr}_0), init\$abst\rangle \rightarrow \langle Tr(\text{instr}_1), s_1\$abst\rangle \rightarrow \cdots \rightarrow \langle Tr(\text{instr}_n), s_n\$abst\rangle \rightarrow \langle\text{assert } Tr_{read}(\texttt{exp}), s\$abst\rangle$$

where $s\$abst|s$.

Note that $\langle\texttt{exp}, s\rangle \rightarrow_e^* \texttt{false}$. Based on Lemma 1, $\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{false}$. Therefore,

$$\cfrac{\langle Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow_e^* \texttt{false}}{\langle\text{assert } Tr_{read}(\texttt{exp}), s\$abst\rangle \rightarrow \langle\text{fail}, s\$abst\rangle}$$