MSc Project Notes

Opale

April 20, 2024

1 Memory model constructor cheatsheet

Note $X^? \stackrel{\text{def}}{=} X \uplus \perp$, $X^\emptyset \stackrel{\text{def}}{=} X \uplus \emptyset$

1.1 Examples per language

Language	Memory Model
WISL	PMap(*Loc*, OneShot(List(Exc(*Val*))))
JSIL	PMap(*Loc*, PMap(*Str*,Exc(*Val*$^\emptyset$)) $\otimes$ PMap(*Loc*,Ag(*Val*))

1.2 State Models

Base building blocks for later transformers. They store values of type τ, usually *Value* or something derived from it. They all define a `load` and `store` action.

Name	Purpose	Type	Predicates
Exc	Exclusive ownership of a specific resources	$\tau^?$	`PointsTo`
Ag	Multiple parties agree on the same value for a resource	τ	`Agree`
Frac	Allow partial (readonly) ownership of an object	$\tau \times (0, 1]$	`Frac`

1.3 State Model Transformers

State model transformers take one or more input state models $\mathbb{S}$ (and an auxiliary sort I in the case of PMap), and result in a new state model. Here the "Type" column only specifies the type of the resulting memory model, the inputs are inferred. $\mathbb{S}.\Sigma$ stands for the heap type of memory model $\mathbb{S}$.

Name	Purpose	Type	Actions	Predicates
Product ($\otimes$)	Two simultaneous states, each being updated separately (eg. List)	$\mathbb{S}_1.\Sigma \times \mathbb{S}_2.\Sigma$	lift with `A1`, `A2`	
Sum ($\oplus$)	Either of two states existing	$\mathbb{S}_1.\Sigma \uplus \mathbb{S}_2.\Sigma$	lift with `A1`, `A2`	
PMap	Define memory as a map of address (a sort I) to value	$(\mathrm{I} \stackrel{fin}{\rightharpoonup} \mathbb{S}.\Sigma) \times \mathcal{P}(I)^?$ *1	lift with index in-param	
List	Ensure continuous memory allocation	$(\mathbb{N} \stackrel{fin}{\rightharpoonup} \mathbb{S}.\Sigma) \times \mathbb{N}^?$ *2	lift with index in-param	
OneShot	The program only has one go at something (eg. freeing memory)	$\mathrm{Exc}(\mathbb{S}.\Sigma) \oplus \mathrm{Exc}(\{\varnothing\})$	`free`	

*1 Full definition: $\left\{(h, d) \in (\mathrm{I} \stackrel{fin}{\rightharpoonup} \tau) \times \mathcal{P}(I)^? \mid \mathrm{dom}(h)^? \subseteq d\right\}$, with the heap h and d the domain set indicating the non-missing indices.

*2 Full definition: $\left\{(b, n^?) \in (\mathbb{N} \stackrel{fin}{\rightharpoonup} \tau) \times \mathbb{N}^? \mid \mathrm{dom}(b) \subseteq [0, n^?)\right\}$, with b the block and n the size of the block if known.

2 `MonadicSMemory` Functions

Name/Type	Description
`type init_data`	Data needed to initialise the memory model (global context)
`type vt` `= SVal.M.t`	Type of GIL Values - always `SVal.M`
`type st` `= SVal.SESubst.t`	Type of substitutions
`type c_fix_t`	How to fix missing errors
`type err_t`	Errors encountered (missing, program errors, logical errors)
`type t`	State type
`type action_ret` `= (t * vt list, err_t) result`	Alias for return type of actions/consume
`val init` `init_data -> t`	Construct the state model, with `init_data` obtained from `ParserAndCompiler`
`val get_init_data` `t -> init_data`	Returns the `init_data` used to construct this memory model, to avoid having the engine keep track of it
`val clear` `t -> t`	Returns an "empty" copy of the state, ie. the state when it is constructed from `init_data`
`val execute_action` `action_name:string -> t -> vt list -> action_ret Delayed.t`	Executes a GIL action with given parameters, returns a symbolic outcome
`val consume` `core_pred:string -> t -> vt list -> action_ret Delayed.t`	Substract the state corresponding to the given core predicate, `vt list` being the in-params of the predicate
`val produce` `core_pred:string -> t -> vt list -> t Delayed.t`	Extend the state with the given core predicate – `vt list` are the in-params AND the out-params of the predicate
`val is_overlapping_asrt` `string -> bool`	Always false, to make GIllian handle overlapping equality stuff
`val copy` `t -> t`	Produces a copy of the state (in case it is mutable)
`val pp` `Format.formatter -> t -> unit`	Pretty print the state
`val substitution_in_place` `st -> t -> t Delayed.t`	Applies substitution to the state, replacing variables with their values. Not in place.
`val clean_up` `?keep:Expr.Set.t -> t -> Expr.Set.t * Expr.Set.t`	Ignore
`val lvars` `t -> Containers.SS.t`	Returns all logical values in the state to ensure that simplifications don't remove variables we need
`val alocs` `t -> Containers.SS.t`	Returns all the abstract locations in the state – ignore for now or return recursively
`val assertions` `?to_keep:Containers.SS.t -> t -> Asrt.t list`	Make a list of logical assertions from the state ($\star$, predicates, formulae, typing...). Note sure what `to_keep` is.
`val mem_constraints` `t -> Formula.t list`	Weird extra well-formedness assertions, that shouldn't matter because they should be handled in `produce` anyways.
`val pp_c_fix` `Format.formatter -> c_fix_t -> unit`	Pretty print fix value
`val get_recovery_tactic` `t -> err_t -> vt Recovery_tactic.t`	Given a state and error, returns two lists of values that should be folded and unfolded respectively
`val pp_err` `Format.formatter -> err_t -> unit`	Pretty print error
`val get_failing_constraint` `err_t -> Formula.t`	A formula that must be satisfied to avoid causing the given error (?)
`val get_fixes` `t -> PFS.t -> Type_env.t -> err_t -> (c_fix_t list * Formula.t list * (string * Type.t) list * Containers.SS.t) list`	???
`val can_fix` `err_t -> bool`	If an error is fixable (if missing)

Name/Type	Description

`val apply_fix`
`t -> c_fix_t`
`-> (t, err_t) result Delayed.t`
	Apply a given fix to a state, possibly resulting in a new error

`val pp_by_need`
`Containers.SS.t -> Format.formatter`
`-> t -> unit`
	Pretty print the state (?)

`val get_print_info`
`Containers.SS.t -> t`
`-> Containers.SS.t * Containers.SS.t`
	Given ? and a state, returns a tuple of ? and ? to print

`val sure_is_nonempty` `t -> bool`	If this state fragment is empty - can be over-approximated to always be `false`

`val split_further`
`t -> string -> vt list -> err_t`
`-> (vt list list * vt list) option`
	If an error occurred when trying to split a core predicate, offers a new way of splitting it, with a list of ins and ways of learning the outs. Related to wands. Can always return None

3 Mismatches

Differences between the theory and what is implemented in Gillian.

Theory	Gillian

`val eval_action :`
$\mathcal{A} \to \Sigma \to \mathit{Val}$ `list` $\to (\mathcal{O} \times \mathit{Val} \times \Sigma)$ `set`
	`val execute_action :`
	`string` $\to$ `t` $\to$ `vt list` $\to$ `action_ret Delayed.t`
	with `action_ret = (t * vt list, err_t) result`
	(note `vt list`, rather than `vt`)

`produce` $\sigma\ \delta\ \vec{v_i}\ \vec{v_o} = \{\sigma \cdot \sigma_\delta \mid \sigma_\delta \vDash \langle\delta\rangle\,(\vec{v_i}, \vec{v_o})\}$, ie.
`val produce :`
$\Sigma \to \Delta \to \mathit{Val}$ `list` $\to \mathit{Val}$ `list` $\to \Sigma$ `list`
	`val produce :`
	`core_pred:string` $\to$ `t` $\to$ `vt list` $\to$ `t Delayed.t`
	(note there is only one `vt list` input, for $\vec{v_i}$)