MSc Project Notes

Opale

May 8, 2024

1 Memory model constructor cheatsheet

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

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	$(\mathrm{I} \xrightharpoonup{fin} \mathbb{S}.\Sigma) \times \mathcal{P}(I)^?$ *1	lift with index in-param	
List	Ensure continuous memory allocation	$(\mathbb{N} \xrightharpoonup{fin} \mathbb{S}.\Sigma) \times \mathbb{N}^?$ *2	lift with index in-param	
Freeable	The program only has one go at something (eg. freeing memory)	Exc($\mathbb{S}.\Sigma$) $\oplus$ Exc($\{\varnothing\}$)	`free`	

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

*2 Full definition: $\left\{(b, n^?) \in (\mathbb{N} \xrightharpoonup{fin} \tau) \times \mathbb{N}^? \mid \mathrm{dom}(b) \subseteq [0, n^?)\right\}$, 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, the given `vt list` being the in-params of the predicate, and the `vt list` of the returned `action_ret` being the out-params.
`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`	???

Name/Type	Description
`val can_fix` `err_t -> bool`	If an error is fixable (if missing)

`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 :`
$\mathcal{A} \to \Sigma \to$ *Val* `list` $\to (\mathcal{O} \times$ *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 :`
$\Sigma \to \Delta \to$ *Val* `list` $\to$ *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 $\vec{v_i}$)

4 Emptiness in state models

4.1 Current State

There exists a source of unsoundness in the current definition of state models, related to the representation of empty states, as there may exist multiple different observably empty states, leading to composition rules being unsound if not accounted for.

To demonstrate this, we may consider the Freeable state. It is defined as:

$$\mathrm{Freeable}(X) \stackrel{\text{def}}{=} \bot \mid \varnothing \mid \mathit{Val}\ X$$
$$\underline{\mathrm{Freeable}}(X) \stackrel{\text{def}}{=} \varnothing \mid \mathit{Val}\ X$$

With composition:

$$f \cdot \bot = f$$
$$\bot \cdot f = f$$
$$\mathit{Val}\ f_1 \cdot \mathit{Val}\ f_2 = \mathit{Val}\ (f_1 \cdot f_2)$$

This forms a valid partially commutative monoid (PCM). We may now look at an example of using this state model that proves to be unsound. Assume the state model PMap($\mathbb{N}$, Freeable(Exc)), with PMap representing a partial map, that is initially with no mappings, []. We omit the domain set for brevity.

State	Operation applied
$\bot$	Initial State
$0 \mapsto \mathit{Val}\ a$	Produce $\langle$`PointsTo`$\rangle\,(0; a)$
$0 \mapsto \mathit{Val}\ \bot$	Consume $\langle$`PointsTo`$\rangle\,(0; a)$
undefined!	Produce $\langle$`Free`$\rangle\,(0;)$

This is, of course, unsound, as the state at the end should indeed be $0 \mapsto \varnothing$. This is due to the fact what is effectively an empty state, $\mathit{Val}\ \bot$, is different from the Freeable empty state, $\bot$, rendering the composition $\mathit{Val}\ \bot \cdot \varnothing$ undefined. Note that consuming `PointsTo` doesn't result in a $0 \mapsto \bot$ state, as PMap must pass consumption down, and so must Freeable, as neither are "aware" of the meaning of `PointsTo`.

To fix this, Freeable would need to be able to know if its contents are observably empty after consuming a predicate, and if so become $\bot$. This seems to hint at a definition or property of our state models missing.

This error can also be shown to exist in the implementation itself, by writing a specification with `True` as the precondition, `False` as the postcondition.

```
spec free_cell(x)
  [[ (x == #x) * <points_to>(#x;#anything) ]]
  [[ (ret == null) * <freed>(#x;) ]]
  normal
proc free_cell(x) {
  n := [free](x);
  ret := null;
  return
};

spec test_unsoundness()
  [[ True ]]
  [[ False ]]
  normal
proc test_unsoundness() {
  x := [alloc]();
  x := l-nth(x, 0i);
  n := "free_cell"(x);
  ret := null;
  return
};
```

This GIL code is then succesfuly verified, with the following output.

```
Parsing and compiling...
Preprocessing...
Obtaining specs to verify...
Obtaining lemmas to verify...
Obtained 2 symbolic tests in total
Running symbolic tests: 0.002729
Verifying one spec of procedure free_cell... s Success
Verifying one spec of procedure test_unsoundness... Success
All specs succeeded: 0.004729
```

4.2 Proof of Unsoundness

First we may take a look at the rules consumers and producers must follow to be sound. Given a set of core predicates $(\Delta \ni \delta, \vDash)$:

$$\texttt{produce}\ \sigma\ \delta\ \vec{v_i}\ \vec{v_o} = \{\sigma \cdot \sigma_\delta \mid \sigma_\delta \vDash \langle\delta\rangle\,(\vec{v_i};\vec{v_o}),\ \sigma\#\sigma_\delta\}$$

$$\sigma.\texttt{consume}_\delta(\vec{v_i}) \to \texttt{Ok} : (\vec{v_o}, \sigma')$$
$$\implies \exists \sigma_\delta.\sigma = \sigma' \cdot \sigma_\delta \wedge \sigma_\delta \vDash \langle\delta\rangle\,(\vec{v_i};\vec{v_o})$$

The `produce` rule in particular states that the result of `produce` must be the given state extended by all disjoint resources that could be associated to a given core predicate.

Freeable(X) being a derivative of a sum state model (in the form of X $\oplus$ Freed), and similar unsoundness being present in the sum state model, we may look at what property sum breaks to exhibit such unsoundness. First, we may have take a look at how sum is originally defined.

The sum $\mathbb{S}.1 \oplus \mathbb{S}.2$ is defined as `type` Σ `=` $\perp_\oplus$ `| S1 of` $\mathbb{S}_1.\Sigma$ `| S2 of` $\mathbb{S}_2.\Sigma$, with the following composition rules and `produce` implementation (note we annotate the sum's $\perp$ element as $\perp_\oplus$ to later distinguish it from the $\perp$ of other state models):

$$\begin{gathered}
\sigma \cdot \perp_\oplus = \perp_\oplus \cdot\, \sigma = \sigma \\
(\mathtt{S}_1\ \sigma_1) \cdot (\mathtt{S}_1\ \sigma_1') = \mathtt{S}_1\ (\sigma_1 \cdot \sigma_1') \\
(\mathtt{S}_2\ \sigma_2) \cdot (\mathtt{S}_2\ \sigma_2') = \mathtt{S}_2\ (\sigma_2 \cdot \sigma_2') \\
\text{undefined otherwise}
\end{gathered}$$

```
produce σ δ v⃗ᵢ v⃗ₒ =
    match σ, δ with
    | S1 σ₁, P1 δ₁ ->
        let* δ₁' = produce σ₁ δ₁ v⃗ᵢ v⃗ₒ in
        S1 δ₁'
    | ⊥⊕, P1 δ₁ ->
            let* δ₁' = produce 𝕊₁.0 δ₁ v⃗ᵢ v⃗ₒ in
        S1 δ₁'
    | S2 σ₂, P2 δ₂ ->
        let* δ₂' = produce σ₂ δ₂ v⃗ᵢ v⃗ₒ in
        S2 δ₂'
    | ⊥⊕, P2 δ₂ ->
        let* δ₂' = produce 𝕊₂.0 δ₂ v⃗ᵢ v⃗ₒ in
        S2 δ₂'
    | _, _ -> vanish
```

This function simply dispatches the predicate to the corresponding state model, default to the relevant empty state if needed, and vanishes if a mismatch occurs (for either `S1` σ_1`, P2` δ_2 or `S2` σ_2`, P1` δ_1).

Now, let there be a state model A that defines a bottom element $\perp_A$, such that $\forall\sigma.\ \sigma \cdot \perp_A = \sigma$, and a second state model B. Given the state model $A \oplus B$, and the above implementation of `produce`, we get the following results:

$$\texttt{produce}\ \sigma\ \delta\ \vec{v_i}\ \vec{v_o} = \begin{cases} \texttt{produce}\ \sigma_A\ \delta_A\ \vec{v_i}\ \vec{v_o} & \text{if } \delta = \texttt{P1}\ \delta_A \land \sigma = \texttt{S1}\ \sigma_A \\ \texttt{produce}\ A.0\ \delta_A\ \vec{v_i}\ \vec{v_o} & \text{if } \delta = \texttt{P1}\ \delta_A \land \sigma = \bot_\oplus \\ \texttt{produce}\ \sigma_B\ \delta_B\ \vec{v_i}\ \vec{v_o} & \text{if } \delta = \texttt{P2}\ \delta_B \land \sigma = \texttt{S2}\ \sigma_B \\ \texttt{produce}\ B.0\ \delta_B\ \vec{v_i}\ \vec{v_o} & \text{if } \delta = \texttt{P2}\ \delta_B \land \sigma = \bot_\oplus \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

Let the current state be $\texttt{S1}\ \bot_A$. As $\bot_A$ is the empty state for the A memory model, it is disjoint from any other state. Given a core predicate δ_B and its ins and outs, there may exist a state σ_B such that $\sigma_B \vDash \langle\delta_B\rangle\,(\vec{v_i};\vec{v_o})$ and we know that $\bot_A \# \sigma_B$.

As such, we would expect $\texttt{produce S1}\ \bot_A\ \delta_B\ \vec{v_i}\ \vec{v_o} = \{\texttt{S2}\ \sigma_B \mid \sigma_B \in \texttt{produce}\ B.0\ \delta_B\ \vec{v_i}\ \vec{v_o}\}$, however that is not the case:

$$\begin{aligned} \texttt{produce}\ (\texttt{S1}\ \bot_A)\ \delta_B\ \vec{v_i}\ \vec{v_o} &= \{\texttt{S1}\ \bot_A \cdot \texttt{S2}\ \sigma_{\delta_B} \mid \sigma_{\delta_B} \vDash \langle\delta\rangle\,(\vec{v_i};\vec{v_o}), \bot_A \# \sigma_{\delta_B}\} && \text{from core predicates producers rule} \\ &= \emptyset && \text{as } \texttt{S1}\ \sigma_A \cdot \texttt{S2}\ \sigma_B \text{ is undefined} \end{aligned}$$

$\nexists \delta.\ \texttt{produce}\ \sigma\ \delta\ \vec{v_i}\ \vec{v_o} \not\subseteq \{\sigma\}$
ie. $\forall \delta.\ \texttt{produce}\ \sigma\ \delta\ \vec{v_i}\ \vec{v_o} \subset \{\sigma\}$

For the result of `produce` to match what's expected, new rules would need to be added to composition and produce to handle $\texttt{S1}\ \bot_A$ in the same way it handles $\bot_\oplus$. However there currently is no way of doing this, as the existence of an empty element in A or B is not exposed – while some state models (like Exc) may have one, others (like Ag) may not.

4.3 Sound Emptiness

A fix to this is to define a "global" emptiness, that replaces the different "local" empty states each state model may define. Consuming a predicate, or executing an action, may result in a new state *or* a global $\bot$. This forces state model transformers to handle such empty states, and ensures a state becoming empty deep within a now-observably-empty construction will naturally unwrap into a shallow empty. We also lift the composition operation defined by state models to handle $\bot$: $a \cdot \bot = \bot \cdot a = a$.

A side-effect of this is that a non-$\bot$ state is never considered observably-empty, as otherwise it would be $\bot$ – this allows us to remove the `is_empty` function that was used for some optimisations, as it is sufficient to compare a given state with $\bot$.

An advantage of this approach is that lifting a full state model to a complete state model doesn't require anything (aside from handling $\bot$ in `produce` and `consume`), as the empty compositional state is already added!

We now redefine some of the constructs used in the engine according to this new definition.

$$\begin{gathered} \Sigma^? \stackrel{\text{def}}{=} \Sigma \uplus \bot \\ \texttt{consume} : \Sigma \to \Delta \to \mathit{Val}\ \texttt{list} \to (\mathcal{O}_l^+ \times \mathit{Val}\ \texttt{list} \times \Sigma^?) \\ \texttt{produce} : \Sigma^? \to \Delta \to \mathit{Val}\ \texttt{list} \to \mathit{Val}\ \texttt{list} \to \Sigma\ \texttt{set} \\ \texttt{eval\_action} : \mathcal{A} \to \Sigma^? \to \mathit{Val}\ \texttt{list} \to (\mathcal{O}_l^+ \times \mathit{Val}\ \texttt{list} \times \Sigma^?) \end{gathered}$$

A consideration with this is that `consume` only works on non-$\bot$ elements, as no predicate is satisfied by the empty state (not sure about this). For produce however, a predicate can be produced from the absence of state.

We may now redefine some of the state models with this new idea.

4.3.1 State Sum

A state sum $\mathbb{S}_1 \oplus \mathbb{S}_2$ is now defined as $\texttt{type}\ \Sigma\ \texttt{= S1 of}\ \mathbb{S}_1.\Sigma\ \texttt{| S2 of}\ \mathbb{S}_2.\Sigma$. We define composition and `produce` as follows:

$$\begin{gathered} (\texttt{S1}\ \ \sigma) \cdot (\texttt{S1}\ \ \sigma') = \texttt{S1}\ \ (\sigma \cdot \sigma') \\ (\texttt{S2}\ \ \sigma) \cdot (\texttt{S2}\ \ \sigma') = \texttt{S2}\ \ (\sigma \cdot \sigma') \\ \text{undefined otherwise} \end{gathered}$$

```
produce σ? δ v⃗ᵢ v⃗ₒ =
    match σ?, δ with
    | S1 σ₁, P1 δ₁ ->
        let* δ₁' = produce σ₁ δ₁ v⃗ᵢ v⃗ₒ in
        S1 δ₁'
    | ⊥, P1 δ₁ ->
            let* δ₁' = produce 𝕊₁.0 δ₁ v⃗ᵢ v⃗ₒ in
        S1 δ₁'
    | S2 σ₂, P2 δ₂ ->
        let* δ₂' = produce σ₂ δ₂ v⃗ᵢ v⃗ₒ in
        S2 δ₂'
    | ⊥, P2 δ₂ ->
        let* δ₂' = produce 𝕊₂.0 δ₂ v⃗ᵢ v⃗ₒ in
        S2 δ₂'
    | _, _ -> vanish
```