IMPERIAL

Project Notes

Background & Progress Report

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

August 8, 2024

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing (Software Engineering)

Contents

Chapter 1

Resource Algebras for a CSE

1.1 Current State

1.1.1 CSE

CSE and Sacha's thesis do the traditional choice of using Partial Commutative Monoids (PCMs) to model state. They are defined as the tuple $(M, (\cdot) : M \times M \rightharpoonup M, 0)$. They are further equipped with a set of actions $\mathcal{A}$, an `execute_action` function, a set of core predicates Δ and a pair of `consume` and `produce` functions.

These additions are necessary for the engine to be parametric on the state model, as it provides an interface for interaction with the state.

The usage of PCMs comes with issues: the requirement of a single 0 for each state model means that state models such as the sum state model $\mathbb{S}_1 + \mathbb{S}_2$ come with unwieldy requirements to prove soundness – this comes into play for the `Freeable` state model, that could use a sum (like what is done in [1]) but can't because of this.

1.1.2 Iris

Iris [2] departs from this tradition and introduces Resource Algebras (RAs) to model state, defined as a tuple $(M, \overline{\mathcal{V}} : M \to \mathbb{B}, |-| : M \to M^?, (\cdot) : M \times M \to M)$, being respectively the state elements, a validity function, a partial core function and a composition function.

This makes Iris states more powerful, in that they have more flexibility in what they can express; for instance sum state models can be easily and soundly expressed, which isn't possible with PCMs due to the requirement of a single 0 element.

Furthermore, Iris RAs comes with plenty proofs and properties making them easy to use and adapt, whereas PCMs can prove unwieldy even for simpler state models (eg. with the Freeable state model transformer).

A similarity however is that the global RA in Iris must be unital, meaning it must have a single ϵ element, very much as it is the case with the 0 in PCMs. Any RA can be trivially extended to have a unit, which is what Iris defines as the option resource algebra [3].

1.2 Partial RAs

A property of Iris RAs is that composition is *total* – to take into account invalid composition, states are usually extended with a $\lightning$ state, such that $\neg\overline{\mathcal{V}}(\lightning)$ (while for states $\sigma \neq \lightning$, $\overline{\mathcal{V}}(\sigma)$

A *resource algebra* (RA) is a triple $(M, |-| : M \to M^?, (\cdot) : M \times M \rightharpoonup M)$

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-Assoc)}$$
$$\forall a, b.\, a \cdot b = b \cdot a \qquad \text{(RA-Comm)}$$
$$\forall a.\, |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-Core-ID)}$$
$$\forall a.\, |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-Core-Idem)}$$
$$\forall a, b.\, |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(RA-Core-Mono)}$$

where $M^? \stackrel{\text{def}}{=} M \uplus \{\bot\}$, with $a \cdot \bot \stackrel{\text{def}}{=} \bot \cdot a \stackrel{\text{def}}{=} a$

$a \preccurlyeq b \stackrel{\text{def}}{=} \exists c.\, b = a \cdot c$

$a \# b \stackrel{\text{def}}{=} a \cdot b$ is defined

A *unital* resource algebra is a resource algebra M with an element $\epsilon \in M$ such that:

$$\forall a \in M. \epsilon \cdot a = a \qquad |\epsilon| = \epsilon$$

Figure 1.1: Definition of Resource Algebras

holds). While this is needed in the Iris framework for higher-order ghost state and step-index, this doesn't come into play when only manipulating RAs. As such, because this is quite unwieldy, we can remove it by adding partiality instead, such that invalid ($\lightning$) states simply don't exist and the need for a $\overline{\mathcal{V}}$ function vanishes. This is also inline with the core function $(-)$ being partial.

It is worth noting that *partial* RAs are equivalent to regular RAs, *so long as* $\overline{\mathcal{V}}$ *always holds for valid states*[1]. Indeed, compositions that yield $\lightning$ can be made undefined, and the validity function removed, to gain partiality, and inversely to go back to the Iris definition.

An interesting property of this is that because validity is replaced by the fact composition is defined, the validity of a composition is equivalent to the fact two states are disjoint: $\overline{\mathcal{V}}(a \cdot b) \iff a \# b$.

We now define the properties of RAs taking this change into account – see Figure 1.1. From now, the term RA will be used to refer to these partial RAs.

1.3 CSE State Models + RAs

We now propose to redefine the notion of state models. To follow the spirit of CSE, that comes with a core engine, a compositional engine and a bi-abduction engine all built onto each other, we go through each layer, presenting what is for that part of the engine to function.

1.3.1 Core Engine

The core engine enables whole-program symbolic execution. For this state models must firstly define the set of states the execution will happen on; this is done via a partial resource algebra: a tuple $(M, |-| : M \to M^?, (\cdot) : M \times M \rightharpoonup M)$. They are further

[1] This, to our knowledge, is the case for all of the simpler RAs defined in Iris: Ex, Ag_0, sum, product, etc.

equipped with a set of actions $\mathcal{A}$, and an `execute_action` function.

$$\texttt{execute\_action} : \mathcal{P}(\texttt{SVar}) \to \hat{\Sigma}^? \to \mathcal{A} \to \text{Val } \texttt{list} \to \mathcal{P}(\underline{\mathcal{O}}_e \times \hat{\Sigma}^? \times \text{Val } \texttt{list} \times \Pi)$$

The arguments of `execute_action` are, in order: the set of existing symbolic variables (this allows the state model to generate fresh variables, for instance in allocation), the *optional* state the action is executed on, the action, and the received arguments. It returns a set of branches, with an outcome, the new state, the returned values, and the path condition of that branch. It is pretty-printed as $\alpha(SV, \hat{\sigma}, \vec{\hat{v}}_i, o) \rightsquigarrow (\hat{\sigma}', \vec{\hat{v}}_o, \pi)$, with SV omitted when not necessary.

Here, the outcome is an outcome in the set of *full execution outcomes* $\underline{\mathcal{O}}_e = \{\texttt{Ok}, \texttt{Err}\}$. In the next subsection, this set will be extended to account for misses and logical failures, but these do not exist with full semantics.

The main difference here is that the state may be $\bot$, if the action is executed on empty state. This ensures non-unital RAs are not ruled out as invalid – indeed, many useful RAs are not unital and sometimes don't have a unit at all, as is the case for instance for Ex, the exclusively owned cell. One could decide to internally make all RAs of state models unital, and have the state model provide an `empty` function that returns said unit (this is what happens in Gillian). However this introduces unsoundness to certain state model constructions (in particular the sum), as this means the state cannot be *exclusively owned* – the empty state could always be composed with it.

Whole-program symbolic execution is, by definition, non-compositional – it thus operates on *full state*, a notion introduced in [4]. As such, the only valid outcomes here are `Ok` and `Err`.

Because we operate in symbolic memory, an additional piece of information is the *path condition*, the set of constraints accumulated throughout execution. A path condition $\pi \in \Pi$ is a *list* of symbolic values, that evaluates to a boolean. We decide to define it as a list rather than a single conjunction of boolean symbolic values, as this allows us to easily check if a path condition is an extension (or a strengthening) of another, with $\pi' \supseteq \pi$.

1.3.2 Compositional Engine

The compositional engine, built on top of the core engine, allows for verification of function specifications, and handles calling functions by their specification. As such, the state model must be extended with a set of core predicates Δ and a pair of `consume` and `produce` functions (equivalent, respectively, to a symbolic assert and assume). Finally, to link core predicates to states, it provides a $\texttt{sat}_\Delta$ relation.

$$\begin{aligned}
\text{for } M = \{&\texttt{OX}, \texttt{UX}\} \\
&\texttt{consume} : M \to \hat{\Sigma}^? \to \Delta \to \text{Val } \texttt{list} \to \mathcal{P}(\mathcal{O}_l \times \hat{\Sigma}^? \times \text{Val } \texttt{list} \times \Pi) \\
&\texttt{produce} : \hat{\Sigma}^? \to \Delta \to \text{Val } \texttt{list} \to \text{Val } \texttt{list} \to \mathcal{P}(\hat{\Sigma}^? \times \Pi) \\
&\texttt{sat}_\Delta : \Sigma \to \Delta \to \text{Val } \texttt{list} \to \mathcal{P}(\text{Val } \texttt{list})
\end{aligned}$$

Similarly to `execute_action`, the input state can be $\bot$. While intuitively one may assume that the input state of `consume` and the output state of `produce` may never be $\bot$, this would limit what core predicates can do. In particular, this means an *emp* predicate couldn't be defined, since it's production on an empty state results in an empty state.

The arguments of `consume` are, in order: the mode of execution to distinguish between

under-approximate and over-approximate reasoning, the state, the core predicate being consumed, the ins of the predicate. It outputs a *logical outcome*, the state with the matching predicate removed (which may result in an empty state $\bot$), the outs of the predicate and the associated path condition. It is pretty-printed as $\texttt{consume}(m, \hat{\sigma}, \delta, \vec{\hat{v}_i}) \rightsquigarrow (o, \hat{\sigma}_f, \vec{\hat{v}_o}, \pi)$, and when the consumption is valid in both OX and UX the mode is omitted.

For `produce`, the arguments are the state, the core predicate being produced, the ins and the outs of the predicate, resulting in a set of new states and their associated path condition. As an example, producing $x \mapsto 0$ in a state $[1 \mapsto 2]$ results in a new state $[1 \mapsto 2, x \mapsto 0]$ with the path condition $x \neq 1$. If the produced predicate is incompatible with the state (eg. by producing $1 \mapsto y$ in a state containing $1 \mapsto x$), the producer *vanishes*. Inversely, if the assertion can be interpreted in several ways, the producer may branch. It is pretty-printed as $\texttt{produce}(\hat{\sigma}, \delta, \vec{\hat{v}_i}, \vec{\hat{v}_o}) \rightsquigarrow (\hat{\sigma}', \pi)$.

The `sat` relation relates a *concrete* state, core predicate and in-values to a set of out-values. It is pretty-printed as $\sigma \models_\Delta \langle \delta \rangle(\vec{v_i}; \vec{v_o})$. For instance in the linear heap state model, we have $[1 \mapsto 2] \models_\Delta \langle \texttt{points\_to} \rangle(1; 2)$.

Here we define logical outcomes $\mathcal{O}_l = \{\texttt{Ok}, \texttt{LFail}, \texttt{Miss}\}$. These are outcomes that happen during reasoning; in particular, `LFail` equates to a logical failure due to an incompatibility between the consumed predicate and the state. For instance, consuming $1 \mapsto 1$ when in state $1 \mapsto 2$ would yield a `LFail`, while consuming it in state $5 \mapsto 3$ would yield a `Miss`, as a state $1 \mapsto x$ could be composed with it to yield a non-miss outcome.

We must also modify the signature of `execute_action`, to include `Miss` outcomes, via the set of execution outcomes $\mathcal{O}_e = \{\texttt{Ok}, \texttt{Err}, \texttt{Miss}\}$.

An addition to what CSE previously defined is thus the split of what was the `Abort` outcome into `LFail` and `Miss`, this improves the quality of error messages and allows fixing consumption errors due to missing state – this will be described in the next subsection.

A last change compared to CSE is that we drop the path condition parameter to consume and produce – the function instead directly returns the path condition required for the resulting branches, and the engine can filter these. For instance, in UX all consumption branches that result in `LFail` can be dropped, as dropping branches is allowed in UX. This has the advantage of simplifying the axioms, as the path condition is strengthened by definition; the function itself has no way of weakening it. *Mention that this also makes checkpointing with the SAT engine trivial – just set a checkpoint before* `consume/produce`*, add whatever that returns. No need to separate the old from new, avoids duplicating/deduplicating, etc.*

1.3.3 Bi-abduction Engine

To support bi-abduction in the style of Infer:Pulse [5], `Miss` outcomes must be fixed. These outcomes may happen during consumption or during action execution. For this, the state model must provide a `fix` function, that given the details of a miss error (these details being of type Val and returned with the outcome) returns a list of sets of assertions that must be produced to fix the missing error.

$$\texttt{fix} : \text{Val} \to \mathcal{P}(\text{Asrt})\ \texttt{list}$$

Note here we return a *list* of different fixes, which themselves are a set of assertions – this is because, for a given missing error, multiple fixes may be possible which causes branching. For instance, in the typical linear heap, accessing a cell that is not in the state fragment

at address a results in a miss that has two fixes: either the cell exists and points to some existentially quantified variable (the fix is thus $\exists x.\, a \mapsto x$), or the cell exists and has been freed ($a \mapsto \varnothing$).

This approach is different from how Gillian handles it. There the function `fix` returns *pure* assertions (type information, pure formulae) and arbitrary values of type `fix_t`, which can then be used with the `apply_fix` : $\Sigma \to$ `fix_t` $\to \Sigma$ method of the monadic state. This means fixes can be arbitrary modifications to the state that don't necessarily equate to new assertions to add to the anti-frame.

This is a source of unsoundness, as the engine may interpret these modifications as fixes despite them not reliably modifying the state. This can be seen in [4], where not finding the binding in a PMap(X) returns a `MissingBinding` error. While being labelled as a miss, this error can actually not be fixed; PMap simply *lifts* predicates with an additional in parameter for the index. An implementation of that version of PMap(X) could attempt to fix this state by add a binding to $X.0$ (PMaps were originally made for PCMs, which always have a 0 element), which would then eventually lead to another error once the action gets called on the empty state. On top of being under-performing (as several fixes would need to be generated for one action), this requires PMap(X) to allow empty states in the codomain, which means a PMap is never exclusively owned (as a state with a singleton map to $X.0$ can always be composed with it), which limits its usability; aside from not being modelable using RAs, since $\bot$ is not an element of X's carrier set. Finally, if the underlying state model doesn't provide any additional fixes, then the fix for `MissingBinding` cannot be added to the UX specification of a function: there is no assertion generatable from within PMap to represent this modification. As such, having `fix` returns assertions without modifying any state directly ensures fixes are always soundly handled.

To finish this, we may note the solution to the above bug is to proceed executing the action on the underlying state model, giving it an empty state – it will then raise the appropriate `Miss`, which can be fixed, as it is aware of what core predicates are needed to create the required state. For instance, for PMap(Exc) a `load` action on a missing binding would be executed against $\bot$, which would return a `MissingValue` error. The PMap could then wrap the error with information about the index at which the error occurred, `SubError`$(i,$`MissingValue`$)$. When getting the fix, PMap can then call Exc.`fix`, which returns $\exists x.\, \langle points\_to \rangle(; x)$, and lift the fix by adding the index as an in-argument, resulting in the final fix $\exists x.\, \langle points\_to \rangle(i; x)$, which is a valid assertion and can be added to the UX specification for this execution.

1.3.4 Axioms

We may now go over the axioms that must be respected by the above defined functions for the soundness of the engine. Note we will thus focus on the axioms related to the state models in particular, and not the general semantics of the engine.

First, it is worth noting that Gillian supports both over-approximate (OX) and under-approximate (UX) reasoning – for which *frame subtraction* or *frame addition* must hold, respectively. In addition to the axioms in [6] and the upcoming paper on an Abstract CSE, we also take inspiration from [4] and include the notion of *compatibility*, linking full states to compositional states, as well as concrete states to symbolic states.

Indeed there are two orthogonal concepts at play: one is *compositionality*, that introduces `consume` and `produce`, frame addition and subtraction, and ultimately compatibility. The other is *symbolicness*, that introduces soundness and the $\vDash$ relation for symbolic state

modelling. To simplify the axioms, we will only consider the symbolic compositional case, as it is a superset of the other cases.

For all of the axioms we assume we have a symbolic state model $\mathbb{S}$, made of the RA $\hat{\Sigma} \ni \hat{\sigma}$. We consider the initial state $\hat{\sigma}$ well-formed.

Symbolicness Axioms *a nicer name would be good*

$$\theta, s, \sigma \models \bot_{\hat{\Sigma}^?} \implies \sigma = \bot_{\Sigma^?} \qquad \text{(Empty Symbolic Memory)}$$

$$\theta, s, \bot_{\Sigma^?} \models \bot_{\hat{\Sigma}^?} \qquad \text{(Empty Memory)}$$

Note that here we work with the *option* version $\Sigma^?$ of the states Σ, which extends the carrier set with a $\bot_{\Sigma^?}$ element. If the core of a state $|\sigma|$ is undefined, it becomes $|\sigma^?| = \bot_{\Sigma^?}$. Finally, the composition of any state with $\bot_{\Sigma^?}$ is that state.

$$\theta, s, \sigma_1 \models \hat{\sigma}_1 \land \theta, s, \sigma_2 \models \hat{\sigma}_2 \land \sigma_1 \# \sigma_2 \iff \theta, s, (\sigma_1 \cdot \sigma_2) \models (\hat{\sigma}_1 \cdot \hat{\sigma}_2)$$
(Symbolic Memory Composition)

$$\begin{aligned}\theta, s, \sigma \models \hat{\sigma} \land \alpha(\sigma, \vec{v}_i) \rightsquigarrow (o, \sigma', \vec{v}_o) \implies \exists SV, \vec{\hat{v}}_i, \vec{\hat{v}}_o, \hat{\sigma}', \pi, \theta'.\\ \hat{\alpha}(SV, \hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \pi, \vec{\hat{v}}_o) \land \mathsf{SAT}(\pi) \land \theta', s, \sigma' \models \hat{\sigma}' \land\\ [\![\vec{\hat{v}}_i]\!]_{s,\pi} = (\vec{v}_i, \pi') \land [\![\vec{\hat{v}}_o]\!]_{s,\pi'} = (\vec{\hat{v}}_o, \pi'')\end{aligned}$$
(Memory Model OX Soundness)

$$\begin{aligned}&\hat{\alpha}(SV, \hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \pi, \vec{\hat{v}}_o) \land \mathsf{SAT}(\pi) \land \theta, s, \sigma' \models \hat{\sigma}' \land\\ &[\![\vec{\hat{v}}_o]\!]_{\hat{s},\pi} \rightsquigarrow (\vec{v}_o, \pi') \land [\![\vec{\hat{v}}_i]\!]_{\hat{s},\pi'} \rightsquigarrow (\vec{v}_i, \pi'') \implies\\ &\quad \exists \sigma.\, \theta, s, \sigma \models \hat{\sigma} \land \alpha(\sigma, \vec{v}_i) \rightsquigarrow (o, \sigma', \vec{v}_o)\end{aligned}$$
(Memory Model UX Soundness)

Compositionality Axioms

$$\begin{aligned}&\alpha(\hat{\sigma} \cdot \hat{\sigma}_f, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi') \implies\\ &\quad \exists \hat{\sigma}'', o', \vec{\hat{v}}'_o.\, \alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o', \hat{\sigma}'', \vec{\hat{v}}'_o, \pi'') \land\\ &\qquad (o' \neq \mathsf{Miss} \implies o' = o \land \vec{\hat{v}}'_o = \vec{\hat{v}}_o \land \hat{\sigma}' = \hat{\sigma}'' \cdot \hat{\sigma}_f \land \pi'' = \pi')\end{aligned}$$
(Frame subtraction)

$$\begin{aligned}&\alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi') \land o \neq \mathsf{Miss} \land \hat{\sigma}' \# \hat{\sigma}_f \implies\\ &\quad \hat{\sigma} \# \hat{\sigma}_f \land \alpha(\hat{\sigma} \cdot \hat{\sigma}_f, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}' \cdot \hat{\sigma}_f, \vec{\hat{v}}_o, \pi')\end{aligned}$$
(Frame Addition)

Here, we may note that the frame-preserving update $a \rightsquigarrow b$ from Iris is a form of frame subtraction: it guarantees $\forall c.\, \overline{\mathcal{V}}(a \cdot c) \Rightarrow \overline{\mathcal{V}}(b \cdot c)$, with c a frame that can be added to the state ($\hat{\sigma}_f$ in the axiom). This becomes evident when noticing that disjointness of partial RAs equates to validity in Iris RAs, giving us $\forall c.\, a \# c \Rightarrow b \# c$. In fact, the Iris frame-preserving update implies frame subtraction modulo action outcomes. This makes sense, as Iris is used for OX reasoning, and frame subtraction is the property needed for OX soundness. *Maybe move this elsewhere.*

Given the full state model $\underline{\mathbb{S}}$ with the states $\underline{\hat{\Sigma}}$ and actions $\underline{\mathcal{A}}$ and the compositional state model $\mathbb{S}$ with the states $\hat{\Sigma} \supseteq \underline{\hat{\Sigma}}$ and actions $\mathcal{A}$, $\hat{\Sigma}$ is the set of fragments of $\underline{\hat{\Sigma}}$ if $\forall \hat{\sigma} \in \hat{\Sigma}.\, \exists \underline{\hat{\sigma}} \in \underline{\hat{\Sigma}}.\, \hat{\sigma} \preccurlyeq \underline{\hat{\sigma}}$ (completion), and $\forall \underline{\hat{\sigma}}_1, \underline{\hat{\sigma}}_2 \in \underline{\hat{\Sigma}}.\, \underline{\hat{\sigma}}_1 \preccurlyeq \underline{\hat{\sigma}}_2 \implies \underline{\hat{\sigma}}_1 = \underline{\hat{\sigma}}_2$ (inextensibility). These two state models are compositional, denoted $\underline{\mathbb{S}}\tilde{\mathbb{S}}$ if all actions of full states using compositional semantics have the same result as using the full semantics.

$$\underline{\alpha}(\underline{\hat{\sigma}}, \vec{\hat{v}}_i) \rightsquigarrow (o, \underline{\hat{\sigma}}', \vec{\hat{v}}_o, \pi) \iff \alpha(\underline{\hat{\sigma}}, \vec{\hat{v}}_i) \rightsquigarrow (o, \underline{\hat{\sigma}}', \vec{\hat{v}}_o, \pi) \qquad \text{(Compatibility)}$$

Note here that we do not consider compatibility of the semantics of the language, but rather only the compatibility of the actions of the state models – it is assumed the compatibility of the full semantics follow from it.

Does the above make sense? Sacha originally defined compatibility for concrete states, but I should be able to be lifted to symbolic states anyways... I just would want to handle compositionality/symbolicness independently, otherwise I imagine I need to then lift all compositional concrete axioms to compositional symbolic...

$$\begin{array}{l}\mathsf{consume}(\hat{\sigma}, \delta, \vec{\hat{v}}_i) \rightsquigarrow (\mathsf{Ok}, \hat{\sigma}_f, \vec{\hat{v}}_o, \pi_f) \implies \\ \quad \exists \hat{\sigma}_\delta.\, \hat{\sigma} = \hat{\sigma}_f \cdot \hat{\sigma}_\delta \wedge (\forall \theta, s, \sigma, \vec{v}_i, \vec{v}_o. \\ \quad\quad (\theta(\pi \wedge \pi_f) = \mathsf{true} \wedge [\![\vec{\hat{v}}_i]\!]_{\theta,s} = \vec{v}_i \wedge [\![\vec{\hat{v}}_o]\!]_{\theta,s} = \vec{v}_o) \implies \\ \quad\quad\quad \theta, s, \sigma \models \hat{\sigma}_\delta \Leftrightarrow \sigma \models_\Delta \langle\delta\rangle(\vec{v}_i; \vec{v}_o))\end{array}$$

(Consume Soundness and Completeness)

$$\begin{array}{l}(\forall o, \hat{\sigma}' \pi'.\, \mathsf{consume}(OX, \sigma, \delta, \vec{\hat{v}}_i) \rightsquigarrow (o_c, \vec{\hat{v}}_o, \pi') \Rightarrow o_c = \mathsf{Ok}) \wedge \theta, s, \sigma \models \hat{\sigma} \implies \\ \quad \exists \hat{\sigma}', \pi', \sigma'.\, \mathsf{consume}(OX, \hat{\sigma}, \delta, \vec{v}_i, \pi) \rightsquigarrow (\mathsf{Ok}, \hat{\sigma}', \vec{\hat{v}}_o, \pi') \wedge \theta, s, \sigma' \models \hat{\sigma}'\end{array}$$

(Consume OX: No Path Drops)

Because the outcome o_c can be `LFail` *or* `Miss`*, I changed the axiom from having $o_c \neq$* `Abort` *to $o_c =$* `Ok`*. Should be fine?*

$$\begin{array}{l}\mathsf{produce}(\hat{\sigma}_f, \delta, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (\hat{\sigma}, \pi_f) \implies \\ \exists \hat{\sigma}_\delta.\, \hat{\sigma} = \hat{\sigma}_\delta \cdot \hat{\sigma}_f \wedge (\forall \theta, s, \sigma_\delta. \\ \quad \theta(\pi \wedge \pi_f) = \mathsf{true} \Rightarrow [\![\vec{\hat{v}}_i]\!]_{\theta,s} = \vec{v}_i \Rightarrow [\![\vec{\hat{v}}_o]\!]_{\theta,s} = \vec{v}_o \Rightarrow \\ \quad \theta, s, \sigma_\delta \models \hat{\sigma}_\delta \Rightarrow \sigma_\delta \models_\Delta \langle\delta\rangle(\vec{v}_i; \vec{v}_o))\end{array}$$

(Produce: Soundness)

$$\begin{array}{l}[\![\vec{\hat{v}}_i]\!]_{\theta,s} = \vec{v}_i \wedge [\![\vec{\hat{v}}_o]\!]_{\theta,s} = \vec{v}_o \ \wedge \theta, s, \sigma \models \hat{\sigma} \wedge \sigma_\delta \models_\Delta \langle\delta\rangle(\vec{v}_i; \vec{v}_o) \wedge \sigma \# \sigma_\delta \implies \\ \quad \exists \hat{\sigma}_\delta, \pi_f.\, \mathsf{produce}(\hat{\sigma}, \delta, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (\hat{\sigma} \cdot \hat{\sigma}_\delta, \pi_f) \wedge \theta(\pi_f) = \mathsf{true} \wedge \theta, s, \sigma_\delta \models \hat{\sigma}_\delta\end{array}$$

(Produce: Completeness)

1.4 RAs and State Models

We now define the state transformers defined in [4], taking advantage of RAs. We will first define the "leaf" state models: the state models that are don't take any state model as input, Ex, Ag and Frac. We will then look at 3 simple transformer state models, Sum, Product and PProduct. Finally, we will discuss more complex state models, with Freeable, PMap and List.

1.4.1 Exclusive

The exclusive state model Ex(Val) is a simple state model, that represent exclusively owned cells: the cell can can only be owned once, and cannot be composed with any other cell. It is parametric on the values it stores – for the traditional symbolic execution cell, this would

be `Sval`. When clear from context, the type of values is omitted. It's RA is defined as:

$$\textsc{Ex}(X) \stackrel{\text{def}}{=} \text{ex}(x : X)$$
$$|\text{ex}(x)| \stackrel{\text{def}}{=} \bot$$
$$\text{ex}(x_1) \cdot \text{ex}(x_2) \text{ is always undefined}$$

The first notation, $\textsc{Ex}(X)$ is the state model instantiation from the set X to the Ex resource algebra. The notation $\text{ex}(x : X)$ stands for $\{\text{ex}(x) : \forall x \in X\}$ – here '$\text{ex}(x)$' refers to the particular element of the $\textsc{Ex}(X)$ RA where the value is $x \in X$.

Note that the above definition is identical[2] to that in Iris [2], showing that little to no modification is needed to adapt RAs to state models.

It defines two actions, $\mathcal{A} = \{\texttt{load}, \texttt{store}\}$ and a predicate $\Delta = \{\texttt{ex}\}$ We now define the functions for the state model: `execute_action`, `produce`, `consume` and `fix`.

ExLoadOk
$\texttt{load}(\text{ex}(x), []) \rightsquigarrow (\texttt{Ok}, \text{ex}(x), [x], [])$

ExLoadMiss
$\texttt{load}(\bot, []) \rightsquigarrow (\texttt{Miss}, \text{ex}(x), [], [])$

ExStoreOk
$\texttt{store}(\text{ex}(x), [x']) \rightsquigarrow (\texttt{Ok}, \text{ex}(x'), [], [])$

ExStoreMiss
$\texttt{store}(\bot, [x']) \rightsquigarrow (\texttt{Miss}, \text{ex}(x), [], [])$

ExConsOk
$\texttt{consume}(\text{ex}(x), \texttt{ex}, []) \rightsquigarrow (\texttt{Ok}, \bot, [x], [])$

ExConsMiss
$\texttt{consume}(\bot, \texttt{ex}, []) \rightsquigarrow (\texttt{Miss}, \bot, [], [])$

ExProd
$\texttt{produce}(\bot, \texttt{ex}, [], [x]) \rightsquigarrow (\text{ex}(x), [])$

ExFix
$\texttt{fix}\ [] = [\{\exists x.\, \langle \texttt{ex} \rangle (; x)\}]$

1.4.2 Agreement

The agreement state model Ag(Val) is the state model to represent an agreement algebra (sometimes referred to as *knowledge* in the literature [7]): information that can be duplicated.

$$\textsc{Ag}(X) \stackrel{\text{def}}{=} \text{ag}(x : X)$$
$$|\text{ag}(x)| \stackrel{\text{def}}{=} \text{ag}(x)$$
$$\text{ag}(x) \cdot \text{ag}(x') \stackrel{\text{def}}{=} \begin{cases} \text{ag}(x) & \text{if } x = x' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Again, this definition is identical to the one of $\textsc{Ag}_0$ in Iris (the non-step-indexed version of agreement).

Because knowledge is duplicable, it cannot be modified: indeed, one would need to modify all instances of the knowledge to ensure frame preservation still holds. Its actions are thus $\mathcal{A} = \{\texttt{load}\}$, and it has one predicate, $\Delta = \{\texttt{ag}\}$.

AgLoadOk
$\texttt{load}(\text{ag}(x), []) \rightsquigarrow (\texttt{Ok}, \text{ag}(x), [x], [])$

AgLoadMiss
$\texttt{load}(\bot, []) \rightsquigarrow (\texttt{Miss}, \bot, [], [])$

AgConsOk
$\texttt{consume}(\text{ag}(x), \texttt{ag}, []) \rightsquigarrow (\texttt{Ok}, \text{ag}(x), [x], [])$

AgConsMiss
$\texttt{consume}(\bot, \texttt{ag}, []) \rightsquigarrow (\texttt{Miss}, \bot, [][])$

[2]Modulo partiality of composition and the validity function, as explained previously.

AgProdBot

$\mathsf{produce}(\bot, \mathsf{ag}, [], [x]) \rightsquigarrow (\mathrm{ag}(x), [])$

AgProdEq

$\mathsf{produce}(\mathrm{ag}(x), \mathsf{ag}, [], [x']) \rightsquigarrow (\mathrm{ag}(x), [x = x'])$

AgFix

$\mathsf{fix}\ [] = [\{\exists x.\, \langle \mathsf{ag} \rangle(; x)\}]$

1.4.3 Fractional

The fractional state model Frac(Val) is used to handle *fractional permissions* [8], [9]. This allows a cell to be partly owned and its information shared, for instance in multithreading. This is done by pairing every value with a fraction $0 < q \leq 1$, and ensuring the value can only be modified if we own the entire value (ie. $q = 1$).

$$\mathrm{Frac}(X) \stackrel{\text{def}}{=} \mathrm{frac}(x : X, q : (0; 1])$$

$$|\mathrm{frac}(x, q)| \stackrel{\text{def}}{=} \bot$$

$$\mathrm{frac}(x, q) \cdot \mathrm{frac}(x', q') \stackrel{\text{def}}{=} \begin{cases} \mathrm{frac}(x, q + q') & \text{if } x = x' \wedge q + q' \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We may note this definition is different from that in Iris, that chooses to define the fractional state model as Frac $\times$ Ag_0(Val), where Frac is the RA for strictly positive rationals. This work in their case, because they can define actions for any state model easily, so they can define `load` for the product having knowledge of the underlying state models. However, because the state models presented here are aimed at being reused in a variety of contexts while minimising the need for defining new actions and predicates, using their approach would hurt usability, as the `load` action would need to be redefined for this specific instantiation of the product. Furthermore, this construction would yield two predicates: `ag` and `frac`, making its use unpractical.

We instead define the actions $\mathcal{A} = \{\mathsf{load}, \mathsf{store}\}$ and the core predicate $\Delta = \{\mathsf{frac}\}$ as follows:

FracLoadOk

$\mathsf{load}(\mathrm{frac}(x, q), []) \rightsquigarrow (\mathsf{Ok}, \mathrm{frac}(x, q), [x], [], [])$

FracLoadMiss

$\mathsf{load}(\bot, []) \rightsquigarrow (\mathsf{Miss}, \bot, [1], [])$

FracStoreOk

$\mathsf{store}(\mathrm{frac}(x, q), [x']) \rightsquigarrow (\mathsf{Ok}, \mathrm{frac}(x', q), [], [q = 1])$

FracStorePerm

$\mathsf{store}(\mathrm{frac}(x, q), [x']) \rightsquigarrow (\mathsf{Miss}, \mathrm{frac}(x, q), [1 - q], [q < 1])$

FracStoreMiss

$\mathsf{store}(\bot, [x']) \rightsquigarrow (\mathsf{Miss}, \bot, [1], [])$

FracConsAll

$\mathsf{consume}(\mathrm{frac}(x, q), \mathsf{frac}, [q']) \rightsquigarrow (\mathsf{Ok}, \bot, [x], [q = q'])$

FracConsSome

$\mathsf{consume}(\mathrm{frac}(x, q), \mathsf{frac}, [q']) \rightsquigarrow (\mathsf{Ok}, \mathrm{frac}(x, q - q'), [x], [0 < q' < q])$

FracConsMiss

$\mathsf{consume}(\mathrm{frac}(x, q), \mathsf{frac}, [q']) \rightsquigarrow (\mathsf{Miss}, \mathrm{frac}(x, q), [q' - q], [q < q' \leq 1])$

FracConsFail

$\mathsf{consume}(\mathrm{frac}(x, q), \mathsf{frac}, [q']) \rightsquigarrow (\mathsf{LFail}, \mathrm{frac}(x, q), [], [q' \leq 0 \vee 1 < q'])$

FracProdBot

$$\mathsf{produce}(\bot, \mathsf{frac}, [q], [x]) \rightsquigarrow (\mathrm{frac}(x, q), [0 < q \leq 1])$$

FracProdEq

$$\mathsf{produce}(\mathrm{frac}(x, q), \mathsf{frac}, [q'], [x']) \rightsquigarrow (\mathrm{frac}(x, q + q'), [x = x' \land 0 < q' \land q + q' \leq 1])$$

FracFix

$$\mathsf{fix}\ [q] = [\{\exists x.\, \langle \mathsf{frac} \rangle (q; x)\}]$$

Here we note that the fraction part of the state is an in-parameter, whereas the value is an out-parameter. This allows one to explicitly specify the required fraction of the state that is consumed.

1.4.4 Sum

The sum of two state models, denoted $\mathbb{S}_1 + \mathbb{S}_2$, represents all states that are in either one of the two states. Sums are one of the reasons for which the 0 of PCMs was removed, in favour of the core, as it allows both sides of the sum to have a different unit (if any). We re-use the definition of sum from Iris.

$$\textsc{Sum}(X, Y) \stackrel{\text{def}}{=} X + Y \stackrel{\text{def}}{=} l(x : X) \mid r(y : Y)$$

$$l(x) \cdot l(x') \stackrel{\text{def}}{=} l(x \cdot x')$$

$$r(y) \cdot r(y') \stackrel{\text{def}}{=} r(y \cdot y')$$

$$|l(x)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } |x| = \bot \\ l(|x|) & \text{otherwise} \end{cases}$$

$$|r(y)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } |y| = \bot \\ r(|y|) & \text{otherwise} \end{cases}$$

Similarly for the actions and predicates, we simply re-use the underlying function. The actions are defined as $\mathcal{A} = \{\alpha_l : \alpha \in \mathbb{S}_1.\mathcal{A}\} \uplus \{\alpha_r : \alpha \in \mathbb{S}_2.\mathcal{A}\}$, and the core predicates $\Delta = \{\delta_l : \delta \in \mathbb{S}_1.\Delta\} \uplus \{\delta_r : \delta \in \mathbb{S}_2.\Delta\}$.

$$\text{Given } \mathit{wrap}_l(x) = \begin{cases} \bot & \text{if } x = \bot \\ l(x) & \text{otherwise} \end{cases} \text{ and } \mathit{unwrap}_l(x_l) = \begin{cases} \bot & \text{if } x = \bot \\ x_l & \text{if } x = l(x_l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

SumLAction

$$\frac{x = \mathit{unwrap}_l(x_l) \qquad \alpha(x, \vec{v}_i) \rightsquigarrow (o, x', \vec{v}_o, \pi) \qquad x'_l = \mathit{wrap}_l(x') \qquad o \neq \mathtt{Miss}}{\alpha_l(x_l, \vec{v}_i) \rightsquigarrow (o, x'_l, \vec{v}_o, \pi)}$$

SumLActionMiss

$$\frac{x = \mathit{unwrap}_l(x_l) \qquad \alpha(x, \vec{v}_i) \rightsquigarrow (o, x', \vec{v}_o, \pi) \qquad x'_l = \mathit{wrap}_l(x') \qquad o = \mathtt{Miss}}{\alpha_l(x_l, \vec{v}_i) \rightsquigarrow (o, x'_l, \text{`}\mathtt{l}\text{'} :: \vec{v}_o, \pi)}$$

SumLActionIncompat

$$\alpha_l(r(y), \vec{v}_i) \rightsquigarrow (\mathtt{Err}, r(y), [], [])$$

SumLCons

$$\frac{x = \mathit{unwrap}_l(x_l) \qquad \mathtt{consume}(x, \delta_l, \vec{\hat{v}}_i) \rightsquigarrow (o, x', \vec{\hat{v}}_o, \pi) \qquad x'_l = \mathit{wrap}_l(x') \qquad o \neq \mathtt{Miss}}{\mathtt{consume}(x_l, \delta_l, \vec{\hat{v}}_i) \rightsquigarrow (o, x'_l, \vec{\hat{v}}_o, \pi)}$$

SumLConsMiss

$$\frac{x = \mathit{unwrap}_l(x_l) \qquad \mathtt{consume}(x, \delta_l, \vec{\hat{v}}_i) \rightsquigarrow (o, x', \vec{\hat{v}}_o, \pi) \qquad x'_l = \mathit{wrap}_l(x') \qquad o = \mathtt{Miss}}{\mathtt{consume}(x_l, \delta_l, \vec{\hat{v}}_i) \rightsquigarrow (o, x'_l, \text{`l'} :: \vec{\hat{v}}_o, \pi)}$$

SumLConsIncompat

$$\mathtt{consume}(r(y), \delta_l, \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{LFail}, r(y), [], [])$$

SumLProd

$$\frac{x = \mathit{unwrap}_l(x_l) \qquad \mathtt{produce}(x, \delta_l, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (x', \pi) \qquad x'_l = \mathit{wrap}_l(x')}{\mathtt{produce}(x_l, \delta_l, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (x'_l, \pi)}$$

SumLFix

$$\frac{\mathbb{S}_1.\mathtt{fix}\ \vec{\hat{v}}_i = a}{\mathtt{fix}\ \text{`l'} :: \vec{\hat{v}}_i = a}$$

We only describe the rules for the left side of the sum – the equivalent rules for the right hand side are defined analogously.

For fixes to be retrieved from the correct side of the sum, `Miss` outcomes must be extended with an indicator of what side the information comes from, so that the correct `fix` function is then called. Some extra care also needs to be taken to handle $\bot$ states separately, since it is not an element of the underlying state models and as such $l(\bot)$ or $r(\bot)$ are not valid – the auxiliary wrap_l and unwrap_l functions help do this without multiplying by four the number of rules.

The sum is one of the main reasons for the switch from PCMs to RAs, as being able to handle $\bot$ separately is an advantage: it avoids situations where the underlying state may be *observably empty*, but the state of the sum is $l(\mathbb{S}_1.0)$ (if we use PCMs). This causes unsoundness, as for instance actions and predicates belonging to the right side of the sum would then yield `Err` and `LFail` respectively, despite the fact that if the state of the sum was simply $\bot$ they'd succeed.

We may give an example to illustrate: let there be the state model $\mathrm{Ex}(\{1\}) + \mathrm{Ex}(\{2\})$. A valid function specification in this state model is $\{\!\{\langle \mathtt{ex}_l \rangle(;1)\}\!\}\ \mathtt{swap()}\ \{\!\{\langle \mathtt{ex}_r \rangle(;2)\}\!\}$[3], where the `swap` function switches the state from the left hand side to the right hand side. Now if this was constructed using PCMs, the engine would first consume the core predicate for the precondition. The consumption for $\mathrm{Ex}(\{1\})$ is $\mathtt{consume}(\mathrm{ex}(1), \mathtt{ex}_l, []) \rightsquigarrow (\mathtt{Ok}, 0_l, [1], [])$. The consumption for the sum would thus be $\mathtt{consume}(l(\mathrm{ex}(1)), \mathtt{ex}_l, []) \rightsquigarrow (\mathtt{Ok}, l(0_l), [1], [])$. Note, here, that the sum state model has *no way of knowing if the state became empty*, and must thus keep it as $l(0_l)$. The engine would then produce the postcondition, which is $\langle \mathtt{ex}_r \rangle(;2)$ – this would however result in the branch vanishing, as $\mathtt{ex}_r$ is not a predicate that can be produced into some $l(x)$. The function call would thus vanish, which is unsound in OX (and would result in no branches in UX, which is sound but useless). *I have a formal proof of this unsoundness, I'll add it here eventually, or put it in the appendix, TBD.*

Of course one may decide state models must expose an `is_empty` function, and use that instead – however this would needlessly complexify state models and would reinvent the wheel; multi-core resource algebras were specifically created to solve this problem [10], and the partial core of Iris was *also* created for this reason among others [2].

[3]The signature of this `swap` function is analogous to that of the `free` action, for the Freeable state model that will later be described.

1.4.5 Product

The product $\mathbb{S}_1 \times \mathbb{S}_2$ of two state models is the cartesian product of both sets of states. Its RA is defined by lifting all elements pointwise:

$$\begin{aligned} \textsc{Product}(X, Y) &\stackrel{\text{def}}{=} X \times Y \\ (x, y) \cdot (x', y') &\stackrel{\text{def}}{=} (x \cdot x', y \cdot y') \\ |(x, y)| &\stackrel{\text{def}}{=} \begin{cases} (|x|, |y|) & \text{if } |x| \neq \bot \land |y| \neq \bot \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

An interesting property of the product is that if one side of the product has no core, then so does the entire product; this also means that both sides of the product must be defined, or neither are defined. This creates a challenge: if an empty ($\bot$) product produces a core predicate for one of its sides, what happens to the other side of the product? Indeed, while one side becomes, defined, $x \times \bot$ is not a valid state, since $\bot \notin \mathbb{S}_2.\hat{\Sigma}$.

It seems here that the consume-produce interface of our engine, which allows creating state bit by bit, can thus be unadapted for certain RAs. Instead, we define an alternative product RA that is more suited to this engine.

1.4.6 Partial Product

The *partial product* state model, denoted $A \bowtie B$, is an alternative to the Iris product RA, that carries some of its useful properties while being adapted to a produce-consume interface. This product supports having only one side be empty – this is different from the usual product RA, that must have both sides have a value.

$$\begin{aligned} \textsc{PProduct}(X, Y) &\stackrel{\text{def}}{=} X \bowtie Y \stackrel{\text{def}}{=} X^? \times Y^? \backslash \{(\bot, \bot)\} \\ (x, y) \cdot (x', y') &\stackrel{\text{def}}{=} (x \cdot x', y \cdot y') \\ |(x, y)| &\stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } |x| = \bot \land |y| = \bot \\ (|x|, |y|) & \text{otherwise} \end{cases} \end{aligned}$$

The advantage of this definition is twofold. Firstly, it allows expressing fragments of products, where one side may not have a defined core. For instance, given the product state $\text{ex}(a) \times \text{ex}(b)$, one can't express this as the composition of some $\text{ex}(A) \times \bot$ and $\bot \times \text{ex}(b)$, which becomes needed for some state transformers (notably, PMap and List). This is in turn possible with the partial product . The second advantage of the partial product and one of the main motivations behind it is that it *carries on the exclusivity of its components.* Given $\text{exclusive}(a) \stackrel{\text{def}}{=} \forall c.\, \neg(a \# c)$, the following rule holds:

$$\frac{\text{exclusive}(a) \qquad \text{exclusive}(b)}{\text{exclusive}(lr(a, b))}\ \textsc{PartProductEx}$$

This is needed to allow frame-preserving transitions from one side of a sum to another. We may note also that the state model transformer itself could be generalised to handle an arbitrary number of state models, as $(A \bowtie B) \bowtie C \iff A \bowtie (B \bowtie C)$ – for the brevity of this presentation, we will only consider the partial product of two state models.

Similarly to the sum, its actions are $\mathcal{A} = \{\alpha_l : \alpha \in \mathbb{S}_1.\mathcal{A}\} \uplus \{\alpha_r : \alpha \in \mathbb{S}_2.\mathcal{A}\}$, and core predicates $\Delta = \{\delta_l : \delta \in \mathbb{S}_1.\Delta\} \uplus \{\delta_r : \delta \in \mathbb{S}_2.\Delta\}$.

$$\text{Given } \mathit{wrap}(x,y) = \begin{cases} \bot & \text{if } x = \bot \land y = \bot \\ (x,y) & \text{otherwise} \end{cases} \text{ and } \mathit{unwrap}(s) = \begin{cases} (\bot,\bot) & \text{if } s = \bot \\ (x,y) & \text{otherwise} \end{cases}$$

PProductLAction
$$\frac{(x,y) = \mathit{unwrap}(s) \qquad \alpha(x,\vec{\hat{v}_i}) \rightsquigarrow (o,x',\vec{\hat{v}_o},\pi) \qquad s' = \mathit{wrap}(x',y) \qquad o \neq \texttt{Miss}}{\alpha_l(s,\vec{\hat{v}_i}) \rightsquigarrow (o,s',\vec{\hat{v}_o},\pi)}$$

PProductLActionMiss
$$\frac{(x,y) = \mathit{unwrap}(s) \qquad \alpha(x,\vec{\hat{v}_i}) \rightsquigarrow (o,x',\vec{\hat{v}_o},\pi) \qquad s' = \mathit{wrap}(x',y) \qquad o = \texttt{Miss}}{\alpha_l(s,\vec{\hat{v}_i}) \rightsquigarrow (o,s',\texttt{`l'::}\vec{\hat{v}_o},\pi)}$$

PProductLCons
$$\frac{(x,y) = \mathit{unwrap}(s) \qquad \texttt{consume}(x,\delta_l,\vec{\hat{v}_i}) \rightsquigarrow (o,x',\vec{\hat{v}_o},\pi) \qquad s' = \mathit{wrap}(x',y) \qquad o \neq \texttt{Miss}}{\texttt{consume}(s,\delta_l,\vec{\hat{v}_i}) \rightsquigarrow (o,s',\vec{\hat{v}_o},\pi)}$$

PProductLConsMiss
$$\frac{(x,y) = \mathit{unwrap}(s) \qquad \texttt{consume}(x,\delta_l,\vec{\hat{v}_i}) \rightsquigarrow (o,x',\vec{\hat{v}_o},\pi) \qquad s' = \mathit{wrap}(x',y) \qquad o = \texttt{Miss}}{\texttt{consume}(s,\delta_l,\vec{\hat{v}_i}) \rightsquigarrow (o,s',\texttt{`l'::}\vec{\hat{v}_o},\pi)}$$

PProductLProd
$$\frac{(x,y) = \mathit{unwrap}(s) \qquad \texttt{produce}(x,\delta_l,\vec{\hat{v}_i},\vec{\hat{v}_o}) \rightsquigarrow (x',\pi) \qquad s' = \mathit{wrap}(x',y)}{\texttt{produce}(s,\delta_l,\vec{\hat{v}_i},\vec{\hat{v}_o}) \rightsquigarrow (s',\pi)}$$

PProductLFix
$$\frac{\mathbb{S}_1.\texttt{fix}\ \vec{\hat{v}_i} = a}{\texttt{fix `l'::}\vec{\hat{v}_i} = a}$$

1.4.7 Freeable

The Freeable($\mathbb{S}$) state model transformer allows extending a state model with a `free` action, that allows freeing a part of memory. The freed memory can then not be used, and attempting to access it raises a `UserAfterFree` error. This is similar to the OneShot RA of Iris, with the key difference that the `freed` predicate used to mark a resource as freed is *not duplicable* – whereas Iris defines OneShot(X) $\stackrel{\text{def}}{=}$ Frac + Ag(X), this definition is not UX-sound. *Maybe a small proof of this? Or is it evident?* This is not a problem for Iris, that is only concerned with OX soundness, but justifies this modification for this engine.

To ensure only full states are freed, the underlying state model must provide a function `is_exclusively_owned` : $\hat{\Sigma} \to$ `LVal`, which equates to the property 'exclusive' presented before. Note here this returns a *symbolic value*, that can be appended to the path condition (it must thus be a boolean). This allows symbolic ownership, for instance having the fraction of Frac be a symbolic number.

While not needed for the core and compositional engines, the bi-abductive engine requires that misses may be fixed; Freeable($\mathbb{S}$) however cannot access directly the core predicates of $\mathbb{S}$ to provide fixes when freeing an empty state. As such, the state model must also provide a `fix_owned` : $\Sigma^? \to \mathcal{P}$(Asrt) `list` function that returns possible fixes to

make the given state exclusively owned. This also implies that full states of $\mathbb{S}$ are exclusively owned – this is not the case, for instance, for Ag. As such, constructions such as Freeable(Ag) are not sound.

Its RA is defined as a construction: $\textsc{Freeable}(\mathbb{S}) \stackrel{\text{def}}{=} \mathbb{S} + \textsc{Ex}(\{\text{freed}\})$, which allows all associated rules to be kept. For clarity, we rename the core predicate $\mathtt{ex}_r$ (defined by $\textsc{Ex}(\{\text{freed}\})$) as `freed`. We also extend its actions with the `free` action.

Because Freeable is constructed via other state model transformers, we only need to describe the rules for `free` – the rest of the construction is already sound. This is an example of how simpler state models can be extended while alleviating the user from the burden of proving the soundness of the base construction.

FreeableActionFree

$\mathtt{free}(l(x), []) \rightsquigarrow (\mathtt{Ok}, r(\text{ex}(\text{freed})), [], [\mathbb{S}.\mathtt{is\_exclusively\_owned}\ x])$

FreeableActionFreeErr

$\mathtt{free}(l(x), []) \rightsquigarrow (\mathtt{Miss}, l(x), \mathbb{S}.\mathtt{fix\_owned}\ x, [\neg\mathbb{S}.\mathtt{is\_exclusively\_owned}\ x])$

FreeableActionFreeMiss

$\mathtt{free}(\bot, []) \rightsquigarrow (\mathtt{Miss}, \bot, \mathbb{S}.\mathtt{fix\_owned}\ \bot, [])$

FreeableActionDoubleFree

$\mathtt{free}(r(\text{ex}(\text{freed})), []) \rightsquigarrow (\mathtt{Err}, r(\text{ex}(\text{freed})), [], [])$

We may note that use-after-free errors are already handled by the sum construction, thanks to the SumLActionIncompat rule.

1.4.8 PMap

The partial map transformer, $\textsc{PMap}(I, \mathbb{S})$, allows modelling partial finite maps. It receives a domain I, and the codomain state model $\mathbb{S}$.

$$\textsc{PMap}(I, \mathbb{S}) \stackrel{\text{def}}{=} I \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathcal{P}(I)^?$$

$$(h, d) \cdot (h', d') \stackrel{\text{def}}{=} (h'', d'')$$

$$\text{where } h'' \stackrel{\text{def}}{=} \lambda i. \begin{cases} h(i) \cdot h'(i) & \text{if } i \in \text{dom}(h) \cap \text{dom}(h') \\ h(i) & \text{if } i \in \text{dom}(h) \setminus \text{dom}(h') \\ h'(i) & \text{if } i \in \text{dom}(h') \setminus \text{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' \stackrel{\text{def}}{=} \begin{cases} d & \text{if } d' = \bot \\ d' & \text{if } d = \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' = \bot \vee \text{dom}(h'') \subseteq d''$$

$$|(h, d)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \text{dom}(h') = \emptyset \\ (h', \bot) & \text{otherwise} \end{cases}$$

$$\text{where } h' \stackrel{\text{def}}{=} \lambda i. \begin{cases} |h(i)| & \text{if } i \in \text{dom}(h) \wedge |h(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

The state is thus the bindings from index to substate, as well as a possibly unset *domain set*. This domain set allows distinguishing, when an index is not found in the map, if the

outcome is an `Err` or `LFail` (because the binding cannot exist), or if the state at that index is actually $\bot$ (in which case the `execute_action`, `consume` or `produce` call is done with $\bot$ as an input). More than improving automation, this is a requirement, for compatibility with the full state model (Compatibility) and to preserve Frame subtraction and Frame Addition.

PMap has a well-formedness condition, to ensure the domain of the bindings is a subset of the domain set – for instance, the state $(\{2 \mapsto x\}, \{1\})$ is not valid, as $\{2\} \not\subseteq \{1\}$. If the domainset is missing, then there is no restrictions on the bindings.

PMap$(I, \mathbb{S})$ defines one action, `alloc`, and one predicate, `domainset`. Furthermore, it lifts all predicates of the wrapped state model, by adding the index of the cell as an in-parameter. Furthermore, to allow allocation, $\mathbb{S}$ must provide an `instantiate` : Val `list` $\to \Sigma$ function, to instantiate a new state from a list of arguments. This is needed because PMap has no awareness of how $\mathbb{S}$ works, or or what it should be initialised to. *Maybe specify that CSE/Sacha don't mention this: PMap allocation requires an interface for allocation.*

For the rules below, we define $h[i \leftarrow s]$ as setting the binding at index i of h to s, and $h[i \not\leftarrow]$ as removing the binding at i from h.

We first define a helper methods `get` and `set`, that allows modifying a symbolic map with branching. After a look up, it returns the value at the location (which may be $\bot$ if it's not found), and the path condition corresponding to the branch. This allows simplifying shared rules for `execute_action`, `produce` and `consume` for PMap.

$$\texttt{get} : ((I \xrightarrow{fin} X) \times \mathcal{P}(I)^?) \to I \to \mathcal{P}(I \times X \times \Pi)$$
$$\texttt{set} : ((I \xrightarrow{fin} X) \times \mathcal{P}(I)^?) \to I \to X \to (I \xrightarrow{fin} X \times \mathcal{P}(I)^?)$$

We pretty-print `get` and `set` as $\texttt{get}(s, i) \rightsquigarrow (i', x, \pi)$ and $\texttt{set}(s, i, x) \to s'$.

$$\text{Given } \mathit{wrap}(h, d) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \mathrm{dom}(h) = \emptyset \wedge d = \bot \\ (h, d) & \text{otherwise} \end{cases}$$
$$\mathit{unwrap}(s) \stackrel{\text{def}}{=} \begin{cases} ([], \bot) & \text{if } s = \bot \\ (h, d) & \text{if } s = (h, d) \end{cases}$$

PMapGetMatch
$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i' \in \mathrm{dom}(h) \qquad s_{i'} = h(i')}{\texttt{get}(s, i) \rightsquigarrow (i', s_{i'}, [i = i'])}$$

PMapGetAdd
$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad d \neq \bot}{\texttt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h) \wedge i \in d])}$$

PMapGetBotDomain
$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad d = \bot}{\texttt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h)])}$$

PMapSetSome
$$\frac{(h, d) = \mathit{unwrap}(s) \qquad s_i \neq \bot \qquad h' = h[i \leftarrow s_i] \qquad s' = \mathit{wrap}(h', d)}{\texttt{set}(s, i, s_i) \to s'}$$

PMapSetNone

$$\frac{(h,d) = \mathit{unwrap}(s) \qquad s_i = \bot \qquad h' = h[i \not\leftarrow] \qquad s' = \mathit{wrap}(h',d)}{\mathtt{set}(s,i,s_i) \to s'}$$

An interesting thing to outline here is that we may branch in three different ways when retrieving a binding:

- PMapGetMatch: the index is equal to some index in the map, and we execute the action for that location – note here the resulting path condition has $i = i'$, such that branches where the index isn't equal will be cut. This allows taking into account symbolic values.
- PMapGetAdd: the index is not already in the map, but is part of the domain set; this means the state is $\bot$.
- PMapGetBotDomain: the index is not already in the map, and the domain set is not owned; the given index may thus be valid, and we again return $\bot$.

There is no branching when setting the binding, as the index is already known.

We now define the rules for PMap(I,$\mathbb{S}$):

$$\text{Given } \mathit{lift\_if\_miss}(o,i,\vec{\hat{v}}_i) \stackrel{\text{def}}{=} \begin{cases} i :: \vec{\hat{v}}_i & \text{if } o = \mathtt{Miss} \\ \vec{\hat{v}}_i & \text{otherwise} \end{cases}$$

PMapAction

$$\frac{\mathtt{get}(s,i) \rightsquigarrow (i', s_{i'}, \pi) \qquad \alpha(s_{i'}, \vec{\hat{v}}_i) \rightsquigarrow (o, s'_{i'}, \vec{\hat{v}}_o, \pi') \qquad \mathtt{set}(s, i', s'_{i'}) \to s'}{\alpha(s, i :: \vec{\hat{v}}_i) \rightsquigarrow (o, s', i' :: \vec{\hat{v}}_o, \pi :: \pi')}$$

PMapActionOutOfBounds

$$\frac{d \neq \bot}{\alpha((h,d), i :: \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{Err}, (h,d), [], [i \notin d])}$$

PMapAlloc

$$\frac{\begin{array}{c} d \neq \bot \qquad i \notin SV \\ i \notin \mathrm{dom}(h) \qquad i \notin d \qquad s_i = \mathtt{instantiate}(\vec{\hat{v}}_i) \qquad h' = h[i \leftarrow s_i] \qquad d' = d \uplus i \end{array}}{\mathtt{alloc}(SV, (h,d), \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{Ok}, (h',d'), [i], [i = i])}$$

PMapAllocMiss

$$\frac{(h,d) = \mathit{unwrap}(s) \qquad d = \bot}{\mathtt{alloc}(SV, s, \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{Miss}, s, [\text{`domainset'}], [])}$$

PMapCons

$$\frac{\begin{array}{c} \mathtt{get}(s,i) \rightsquigarrow (i', s_{i'}, \pi) \\ \mathtt{consume}(s_{i'}, \delta, \vec{\hat{v}}_i) \rightsquigarrow (o, s'_{i'}, \vec{\hat{v}}_o, \pi') \qquad \mathtt{set}(s, i', s'_{i'}) \to s' \qquad \vec{\hat{v}}'_o = \mathit{lift\_if\_miss}(o, i', \vec{\hat{v}}_o) \end{array}}{\mathtt{consume}(s, \delta, i :: \vec{\hat{v}}_i) \rightsquigarrow (o, s', \vec{\hat{v}}'_o, \pi :: \pi')}$$

PMapConsIncompat

$$\frac{d \neq \bot}{\mathtt{consume}((h,d), \delta, i :: \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{LFail}, (h,d), [], [i \notin d])}$$

PMapConsDomainSet

$$\frac{d \neq \bot}{\mathtt{consume}((h,d), \mathtt{domainset}, []) \rightsquigarrow (\mathtt{Ok}, (h, \bot), [d], [])}$$

$$\textsc{PMapConsDomainSetMiss}\quad \frac{(h, d) = \mathit{unwrap}(s) \qquad d = \bot}{\mathsf{consume}(s, \mathsf{domainset}, []) \rightsquigarrow (\mathsf{Miss}, s, [\text{`}\mathsf{domainset}\text{'}], [])}$$

$$\textsc{PMapProd}\quad \frac{\mathsf{get}(s, i) \rightsquigarrow (i', s_{i'}, \pi) \qquad \mathsf{produce}(s_{i'}, \delta, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (s'_{i'}, \pi) \qquad \mathsf{set}(s, i', s'_{i'}) \to s'}{\mathsf{produce}(s, \delta, i :: \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (s', (i = i') :: \pi)}$$

$$\textsc{PMapProdDomainSet}\quad \frac{(h, \bot) = \mathit{unwrap}(s)}{\mathsf{produce}(s, \mathsf{domainset}, [], [d]) \rightsquigarrow ((h, d), [\mathrm{dom}(h) \subseteq d])} \qquad \textsc{PMapFix}\quad \frac{\mathbb{S}.\mathtt{fix}\ \vec{\hat{v}}_i = a \qquad a' = \mathit{lift}(a, i)}{\mathtt{fix}\ i :: \vec{\hat{v}}_i = a'}$$

$$\textsc{PMapFixDomainSet}\quad \mathtt{fix}\ [\text{`}\mathsf{domainset}\text{'}] = \exists d.\, \langle \mathsf{domainset} \rangle(; d)$$

This simple construction is thus enough to recreate the standard "points to" predicate of standard separation logic [11], [12]: with PMap($\mathbb{N}$, Ex(Val)), one can formulate the core predicates $\langle \mathsf{ex} \rangle(i; x)$, which is equivalent to $i \mapsto x$.

Note here we must unwrap and re-wrap the state model after every action, to properly handle the case where there are no bindings and no domain set as $\bot$. For the `fix` function we must also lift all core predicates, by recursing through the assertions and adding the index to the in-values of all assertions of type $\langle \delta \rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o)$.

We note here that action execution and predicate consumption and production may branch. For instance, if the state is $(\{1 \mapsto y\}, \{1, 2\})$ and a `load` action is executed with an unconstrained symbolic index $\hat{x}$, then three branches are created: one where $\hat{x} = 1$ and the action is executed on the cell y, one where $\hat{x} = 2$ and the action is executed on $\bot$, and finally one where $\hat{x} \notin \{1, 2\}$, and an `Err` is raised for out of bounds access. A fourth branch also gets created, with $\hat{x} \notin \{1, 2\} \land \hat{x} \in \{1, 2\}$ – it however gets cut because of course this path condition is false. This outlines a key difficulty of PMap, that will be discussed more in depth later: implementation-wise, it is a complex and expensive transformer that has the potential to create many branches – this makes it an important target for optimisation.

Using RAs rather than PCMs makes well-formedness easier to uphold, as a binding to $\bot$ is not valid, since $\bot \notin \mathbb{S}.\Sigma$. To exemplify why this is helpful, consider the state model PMap($\mathbb{N}$, Ex($\{1\}$)) and the function `move`, that relocates a memory cell:

$$\{\!\!\{ \langle \mathsf{ex} \rangle(1; 1) \star \langle \mathsf{domainset} \rangle(; \{1\}) \}\!\!\}\ \mathtt{move()}\ \{\!\!\{ \langle \mathsf{ex} \rangle(2; 1) \star \langle \mathsf{domainset} \rangle(; \{2\}) \}\!\!\}$$

Let the initial state be $(\{1 \mapsto \mathrm{ex}(1)\}, \{1\})$: a heap with one cell, at address 1 – this matches exactly the precondition of `move`.

If we are using PCMs, Ex(1) is defined as ex(1) | 0, with 0 the unit of the PCM. Note here that executing actions, consuming and producing doesn't return an element of $\Sigma^?$, but of Σ directly, since the 0 is already in its carrier set; one has no reason to extend it with $\bot$. The engine first consumes the `ex` predicate, at address 1 – this means the pointed-to state becomes 0 (or empty), and the bigger (or outer) state becomes $(\{1 \mapsto 0\}, \{1\})$. The `domainset` predicate is then successfully consumed too, leaving $(\{1 \mapsto 0\}, \bot)$. Now, the post-condition is produced onto the state, as calling the function modified the state of our program. First, the engine thus produces $\langle \mathsf{ex} \rangle(2; 1)$, which adds a binding to PMap and creates the cell: our bigger state is now $(\{1 \mapsto 0, 2 \mapsto \mathrm{ex}(1)\}, \bot)$. Finally, $\langle \mathsf{domainset} \rangle(; \{2\})$ is produced onto the state, however this *doesn't succeed*. Indeed, the bindings of the heap are $\{1, 2\}$, which are not a subset of the produced domainset $\{2\}$. This is because the engine

cannot tell apart empty (0) states from non-empty states. Of course one could handle this by checking for 0, or by explicitly excluding units from the codomain of PMap. This however makes rules and definitions more complex and prone to error, as it is easy to forget to check for units.

If this examples is now reproduced with RAs, the consumption of $\langle \mathsf{ex} \rangle(1;1)$ makes the cell return $\bot$, which cannot be added into the heap – the PMap must thus removes the binding, and the postcondition can be produced properly, as the bindings of the heap is only $\{2\}$. RAs are a solution to the above problem, as they facilitate the sound construction of state models and state model transformers, by allowing unitless state models.

1.4.9 Dynamic PMap

The *dynamic* partial map state transformer, DynPMap$(I, \mathbb{S})$ is similar to the regular (or *static*) PMap transformer, but allows modelling "dynamic" maps that can be modified without allocation. This is used, for instance, to model the JavaScript memory model, where one can set a field of an object directly if it doens't exist, and where reading a field that doesn't exist does not raise an error but simply returns a default `undefined` value.

This transformer has the same RA as PMap, as well as the same predicates. It however only lifts the actions of the underlying state model $\mathbb{S}$, without adding an `alloc` action (since allocation does not exist; the field is just created on access). However, since this requires instantiating the underlying state model, it must still implement an `instantiate` : Val `list` $\to \Sigma$ function. Here we keep Val `list` for compatibility with PMap, but in fact the arguments are always the empty list.

1.4.10 List

The List state model transformer is similar to PMap, but instead of receiving an domain I as a parameter it only operates on positive integers. It allows representing a list of cells, up to a bounded size – it can be used, for instance, to represent a block of memory, with the index serving as the offset of the base address of the block. Similarly to PMap, it represents state as a finite partial map from integers to state, as well as a *bound*: a strictly positive integer specifying the size of the list, and allowing one to distinguish out of bounds from $\bot$ elements.

$$\textsc{List}(\mathbb{S}) \stackrel{\text{def}}{=} \mathbb{N} \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathbb{N}^?$$

$$(b,n) \cdot (b',n') \stackrel{\text{def}}{=} (b'',n'')$$

$$\text{where } b'' \stackrel{\text{def}}{=} \lambda i. \begin{cases} b(i) \cdot b'(i) & \text{if } i \in \text{dom}(b) \cap \text{dom}(b') \\ b(i) & \text{if } i \in \text{dom}(b) \setminus \text{dom}(b') \\ b'(i) & \text{if } i \in \text{dom}(b') \setminus \text{dom}(b) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } n'' \stackrel{\text{def}}{=} \begin{cases} n & \text{if } n' = \bot \\ n' & \text{if } n = \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } \forall i \in \text{dom}(b''). 0 \leq i \wedge (n'' = \bot \vee i < n'')$$

$$|(b,n)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \text{dom}(b') = \emptyset \\ (b', \bot) & \text{otherwise} \end{cases}$$

$$\text{where } b' \stackrel{\text{def}}{=} \lambda i. \begin{cases} |b(i)| & \text{if } i \in \text{dom}(b) \wedge |b(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

Just like in PMap, all core predicates are lifted, and a `bound` core predicate is added for the bound. It does not include an `alloc` action – instead, all cells are created when the list is instantiated (or taken from the specification of the executed function).

For brevity, its rules will not be outlined – they are analogous to those of PMap, with the main difference being that checks for the membership of an index in the domain set are replaced with checks for the index in the range $[0, n)$ with the bound n.

1.4.11 General Map

In [4], the PMap and List transformers are presented as two different transformers, however they can both be merged into a single transformer quite succinctly, to prove the modularity of transformers.

We thus introduce the *general map* transformer, $\textsc{GMap}(I, \mathbb{S}, \mathbb{S}_D)$. It is built from a domain set I, a codomain state model $\mathbb{S}$ and a *discriminator* state model $\mathbb{S}_D$. This last state model serves as a way to tell apart invalid accesses from $\bot$ elements – modelling it as a state model allows us to make it more flexible on what predicates and state are used to represent it.

The discriminator state model $\mathbb{S}_D$ must also provide a $\texttt{is\_within} : \mathbb{S}_D.\Sigma \to I \to \texttt{LVal}$ function. `is_within` returns a symbolic boolean, that evaluates to true if and only if a state fragment containing a singleton partial map with the given index could be validly composed into the state while upholding the desired GMap's invariant. A consequence of this is that when the state is a*full* state, `is_within` is only used for out of bounds accesses (as otherwise the key is already in the map, since we're dealing with full states), and as such always returns false. If we write $\text{dom}\,\sigma$ the domain of a GMap's map, given a set of states Σ with the subset of full states $\underline{\Sigma} \subseteq \Sigma$, we have that $i \notin \text{dom}\,\sigma \wedge \sigma \in \underline{\Sigma} \Rightarrow \texttt{SAT}(\neg \texttt{is\_within}\,\sigma\,i)$.

To replicate the usual PMap, one would have $\mathbb{S}_D = \textsc{Ex}(\mathcal{P}(I))$, with `is_within` true if the value is in the set: $\texttt{is\_within}_{\textsc{PMap}}\ \sigma_D\ i = i \in \sigma_D$. For List, one has $\mathbb{S}_D = \textsc{Ex}(\mathbb{N})$ with $\texttt{is\_within}_{\textsc{List}}\ \sigma_D\ i = 0 \leq i < \sigma_D$. Note here that state transformer may automatically

assume the cell can exist if the key is not found in the map and the discriminator state is $\bot$ in the state tuple. Note for both of the above examples the `load` and `store` actions of Ex should be removed, as modifying the domain set or bound directly is not sound – the discriminator is used for the compositional state to distinguish outcomes, and *does not exist in full states*, so actions to modify the discriminator directly cannot exist in the full state either. Allowing them to exist in the compositional state model would break compatibility.

$$
\begin{aligned}
&\textsc{GMap}(I, \mathbb{S}, \mathbb{S}_D) \stackrel{\text{def}}{=} I \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathbb{S}_D.\Sigma^? \\
&\quad (h,d) \cdot (h',d') \stackrel{\text{def}}{=} (h'',d'') \\
&\qquad \text{where } h'' \stackrel{\text{def}}{=} \lambda i. \begin{cases} h(i) \cdot h'(i) & \text{if } i \in \text{dom}(h) \cap \text{dom}(h') \\ h(i) & \text{if } i \in \text{dom}(h) \setminus \text{dom}(h') \\ h'(i) & \text{if } i \in \text{dom}(h') \setminus \text{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases} \\
&\qquad\quad \text{and } d'' \stackrel{\text{def}}{=} d \cdot d' \\
&\qquad\qquad \text{and } d'' = \bot \vee (\forall i \in \text{dom}(h''). \texttt{is\_within}\ d''\ i) \\
&\qquad\qquad |(h,d)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \text{dom}(h') = \emptyset \wedge d' = \bot \\ (h',d') & \text{otherwise} \end{cases} \\
&\qquad\qquad\quad \text{where } h' \stackrel{\text{def}}{=} \lambda i. \begin{cases} |h(i)| & \text{if } i \in \text{dom}(h) \wedge |h(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad \text{and } d' \stackrel{\text{def}}{=} |d|
\end{aligned}
$$

We now define the rules for $\textsc{GMap}(I, \mathbb{S}, \mathbb{S}_D)$. Its actions are only that of $\mathbb{S}$ (not that of the discriminator state model), as explained previously; it does however inherit predicates from both. As such, $\mathcal{A} = \mathbb{S}.\mathcal{A}$, and $\Delta = \mathbb{S}.\Delta \uplus \{\delta_D : \delta \in \mathbb{S}_D.\Delta\}$.

We also redefine the `get` and `set` helper functions, with `get` branching again. This requires lifting `get` to do checks using `is_within` (the rest of the rule is unaffected).

We first define a helper methods `get` and `set`, that allows modifying a symbolic map with branching. After a look up, it returns the value at the location (which may be $\bot$ if it's not found), and the path condition corresponding to the branch. This allows simplifying shared rules for `execute_action`, `produce` and `consume` for PMap.

$$
\begin{aligned}
&\texttt{get} : ((I \xrightarrow{fin} X) \times \mathbb{S}_D.\Sigma^?) \to I \to \mathcal{P}(I \times X \times \Pi) \\
&\texttt{set} : ((I \xrightarrow{fin} X) \times \mathbb{S}_D.\Sigma^?) \to I \to X \to (I \xrightarrow{fin} X \times \mathbb{S}_D.\Sigma^?)
\end{aligned}
$$

$$
\begin{aligned}
\text{Given } \mathit{wrap}(h,d) &\stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \text{dom}(h) = \emptyset \wedge d = \bot \\ (h,d) & \text{otherwise} \end{cases} \\
\mathit{unwrap}(s) &\stackrel{\text{def}}{=} \begin{cases} ([], \bot) & \text{if } s = \bot \\ (h,d) & \text{if } s = (h,d) \end{cases} \\
\mathit{lift\_if\_miss}(o, i, \vec{\hat{v}_i}) &\stackrel{\text{def}}{=} \begin{cases} i :: \vec{\hat{v}_i} & \text{if } o = \texttt{Miss} \\ \vec{\hat{v}_i} & \text{otherwise} \end{cases}
\end{aligned}
$$

GMapGetMatch

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i' \in \mathrm{dom}(h) \qquad s_{i'} = h(i')}{\mathtt{get}(s, i) \rightsquigarrow (i', s_{i'}, [i = i'])}$$

GMapGetAdd

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad d \neq \bot}{\mathtt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h) \wedge \mathtt{is\_within}\ d\ i])}$$

GMapGetBotDomain

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad d = \bot}{\mathtt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h)])}$$

GMapSetSome

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad s_i \neq \bot \qquad h' = h[i \leftarrow s_i] \qquad s' = \mathit{wrap}(h', d)}{\mathtt{set}(s, i, s_i) \rightarrow s'}$$

GMapSetNone

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad s_i = \bot \qquad h' = h[i \not\leftarrow] \qquad s' = \mathit{wrap}(h', d)}{\mathtt{set}(s, i, s_i) \rightarrow s'}$$

GMapAction

$$\frac{\mathtt{get}(s, i) \rightsquigarrow (i', s_{i'}, \pi) \qquad \alpha(s_{i'}, \vec{v}_i) \rightsquigarrow (o, s'_{i'}, \vec{v}_o, \pi') \qquad \mathtt{set}(s, i', s'_{i'}) \rightarrow s'}{\alpha(s, i :: \vec{v}_i) \rightsquigarrow (o, s', i' :: \vec{v}_o, \pi :: \pi')}$$

GMapActionOutOfBounds

$$\frac{d \neq \bot}{\alpha((h, d), \vec{v}_i) \rightsquigarrow (\mathtt{Err}, s, [], [\neg \mathtt{is\_within}\ d\ i])}$$

GMapCons

$$\frac{\mathtt{get}(s, i) \rightsquigarrow (i', s_{i'}, \pi) \qquad \mathtt{consume}(s_{i'}, \delta, \vec{v}_i) \rightsquigarrow (o, s'_{i'}, \vec{v}_o, \pi') \qquad \vec{v}'_o = \mathit{lift\_if\_miss}(o, i', \vec{v}_o) \qquad \mathtt{set}(s, i', s'_{i'}) \rightarrow s'}{\mathtt{consume}(s, \delta, i :: \vec{v}_i) \rightsquigarrow (o, s', \vec{v}'_o, \pi :: \pi')}$$

GMapConsIncompat

$$\frac{d \neq \bot}{\mathtt{consume}((h, d), \delta, i :: \vec{v}_i) \rightsquigarrow (\mathtt{LFail}, (h, d), [], [\neg \mathtt{is\_within}\ d\ i])}$$

GMapConsDiscr

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad \mathtt{consume}(d, \delta_d, \vec{v}_i) \rightsquigarrow (o, d', \vec{v}_o, \pi) \qquad o \neq \mathtt{Miss} \qquad s' = \mathit{wrap}(h, d')}{\mathtt{consume}(s, \delta_D, \vec{v}_i) \rightsquigarrow (\mathtt{Ok}, s', \vec{v}_o, \pi)}$$

GMapConsDiscrMiss

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad \mathtt{consume}(d, \delta_d, \vec{v}_i) \rightsquigarrow (o, d', \vec{v}_o, \pi) \qquad o = \mathtt{Miss} \qquad s' = \mathit{wrap}(h, d')}{\mathtt{consume}(s, \delta_D, \vec{v}_i) \rightsquigarrow (\mathtt{Ok}, s', \text{`D'} :: \vec{v}_o, \pi)}$$

GMapProd

$$\frac{\mathtt{get}(s, i) \rightsquigarrow (i', s_{i'}, \pi) \qquad \mathtt{produce}(s_{i'}, \delta, \vec{v}_i, \vec{v}_o) \rightsquigarrow (s'_{i'}, \pi') \qquad \mathtt{set}(s, i', s'_{i'}) \rightarrow s'}{\mathtt{produce}(s, \delta, i :: \vec{v}_i, \vec{v}_o) \rightsquigarrow (s', \pi :: \pi')}$$

GMapProdDiscr

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad \mathtt{produce}(d, \delta_D, \vec{v}_i, \vec{v}_o) \rightsquigarrow (d', \pi)}{\mathtt{produce}(s, \delta_D, \vec{v}_i, \vec{v}_o) \rightsquigarrow (s', [\forall i \in \mathrm{dom}(h).\ \mathtt{is\_within}\ d\ i] :: \pi)}$$

$$\textsc{GMapFix}\ \frac{\mathbb{S}.\texttt{fix}\ \vec{\hat{v}}_i = a \qquad a' = \mathit{lift}(a, i)}{\texttt{fix}\ i :: \vec{\hat{v}}_i = a'} \qquad \textsc{GMapFixDiscr}\ \frac{\mathbb{S}_D.\texttt{fix}\ \vec{\hat{v}}_i = a}{\texttt{fix 'D'} :: \vec{\hat{v}}_i = a}$$

These rules are very similar to those of PMap, or the rules List would have, only with the discriminator check instead, as well as more generic `consume`, `produce` and `fix` rules for the discriminator. We also do not provide an `alloc` action with GMap, as the discriminator may need to be modified with a new index; it is up to the user to define any additional actions over it.

An example instantiation of GMap is with the empty state model Emp, a state model that provides no actions or predicates, and that can only be $\bot$. `is_within` may then always return `true`. This allows emulating the traditional separation logic linear heap, where one can always do allocation (this is not the case in PMap, where one must own the domain set to allocate). We note that this instantiation does not uphold Compatibility: any out of bounds access will result in a `Miss`, rather that properly separating misses from errors.

1.5 Optimising maps

As mentioned above, GMap is an ideal target for optimisation: it is a common transformer (used once in the C and WISL memory models, and twice in the JS memory model) that has a high performance cost, due to branching: for instance, executing an action in a map with n cells can lead to $n + 2$ branches. While most of these branches eventually get dropped, as the path condition doesn't hold, there is a cost to checking the satisfiability of the path condition. The *ideal* GMap transformer only returns feasible branches, minimising SAT checks.

Another aspect of optimisation is the need for simplifying *substitution*. While not described here, substituting a GMap's state requires recursing through all key-value pairs, and applying the substitution to both the key and the value, before rebuilding a binding and possibly composing values that end with the same key. For instance, given a construction $\textsc{GMap}(\mathbb{N}, \textsc{Ex}(\{1\}) \bowtie \textsc{Ex}(\{2\}))$ (we ignore the discriminator for brevity) applying the substitution $\theta[\hat{x} \to 0]$ to the mappings $[\hat{x} \mapsto (1, \bot), 0 \mapsto (\bot, 2)]$ would yield $[0 \mapsto (1, 2)]$. Evaluation of Gillian with different state model constructions has shown that, especially for states with large partial maps (as is the case in JavaScript), substitution can take up to *50%* of the execution time directly within the state model (here, we ignore the difference in total execution time of the engine).

We will here explore three different optimisations of PMap that have been ported to Gillian, by justifying their theoretical soundness. While they have been adapted for PMap, they could also be adapted for GMap with little additional effort – we choose not to, to keep the presentation simpler.

1.5.1 Syntactic Checking

A technique introduced in [4] that we may reuse is syntactic equality checking for a matching key before attempting to branch on symbolic equality. This means that if the map contains the exact key already, we do not need to do any branching. This method also benefits from *hash consing*, where a hash is associated to every expression's AST, avoiding the need for recursing through possibly deep trees. *We note hash consing for expressions is currently not implemented in Gillian.*

We present here the rules for this technique. We highlight the new rule and condition, noting the last two rules are unchanged.

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \in \mathrm{dom}(h) \qquad s_i = h(i)}{\texttt{get}(s, i) \rightsquigarrow (i, s_i, [])} \text{ SyntacticPMapGetMatch}$$

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad i' \in \mathrm{dom}(h) \qquad s_{i'} = h(i')}{\texttt{get}(s, i) \rightsquigarrow (i', s_{i'}, [i = i'])} \text{ SyntacticPMapGetBranch}$$

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad d \neq \bot}{\texttt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h) \land i \in d])} \text{ SyntacticPMapGetAdd}$$

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad i \notin \mathrm{dom}(h) \qquad d = \bot}{\texttt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h)])} \text{ SyntacticPMapGetBotDomain}$$

This optimisation is also the reason why actions on PMap always return the accessed index; it allows further accesses to the same element to benefit from this syntactic equality check, and thus avoids repetitive SAT checks.

1.5.2 Split PMap

The idea behind this first optimisation is to split the bindings of the map between *concrete* and *symbolic*, effectively storing two maps at once. This has a few advantages: firstly, substitution must only be done on the symbolic part of the map, as only symbolic variables can be substituted. The second effect this has is that when doing lookups of a key, if the key is not concrete we may avoid checking for syntactic equality in the concrete part of the map, since the binding cannot be there. Note this doesn't hold for the opposite: the key may be concrete, but if the associated value is symbolic then the binding will be in the symbolic map – a concrete key does thus require doing a lookup on both maps.

Of course, to distinguish concrete from non-concrete cells, it requires the underlying state model to provide an $\texttt{is\_concrete}_\Sigma : \Sigma \to \mathbb{B}$ function.

We first define the resource algebra of this state model, where h_c denotes the concrete

part of the map and h_s the symbolic (or better said, non-concrete) part:

$$\mathrm{PMap}_{\mathrm{Split}}(I, \mathbb{S}) \stackrel{\text{def}}{=} I \xrightarrow{fin} \mathbb{S}.\Sigma \times I \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathcal{P}(I)^?$$

$$(h_c, h_s, d) \cdot (h'_c, h'_s, d') \stackrel{\text{def}}{=} (h''_c, h''_s, d'')$$

$$\text{where } h_{all} \stackrel{\text{def}}{=} h_c \cup h_s, \quad h'_{all} \stackrel{\text{def}}{=} h'_c \cup h'_s$$

$$\text{where } h''_c \stackrel{\text{def}}{=} \lambda i. \begin{cases} h_{all}(i) \cdot h'_{all}(i) & \text{if } i \in \mathrm{dom}(h_{all}) \cap \mathrm{dom}(h'_{all}) \wedge \\ & \quad \texttt{is\_concrete}_{\Sigma}\ (h_{all}(i) \cdot h'_{all}(i)) \wedge \\ & \quad \texttt{is\_concrete}\ i \\ h_c(i) & \text{if } i \in \mathrm{dom}(h_c) \setminus \mathrm{dom}(h'_{all}) \\ h'_c(i) & \text{if } i \in \mathrm{dom}(h'_c) \setminus \mathrm{dom}(h_{all}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } h''_s \stackrel{\text{def}}{=} \lambda i. \begin{cases} h_{all}(i) \cdot h'_{all}(i) & \text{if } i \in \mathrm{dom}(h_{all}) \cap \mathrm{dom}(h'_{all}) \wedge \\ & \quad \neg(\texttt{is\_concrete}_{\Sigma}\ (h_{all}(i) \cdot h'_{all}(i)) \wedge \\ & \quad\quad \texttt{is\_concrete}\ i) \\ h_s(i) & \text{if } i \in \mathrm{dom}(h_s) \setminus \mathrm{dom}(h'_{all}) \\ h'_s(i) & \text{if } i \in \mathrm{dom}(h'_s) \setminus \mathrm{dom}(h_{all}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' \stackrel{\text{def}}{=} \begin{cases} d & \text{if } d' = \bot \\ d' & \text{if } d = \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' = \bot \vee \mathrm{dom}(h''_c) \cup \mathrm{dom}(h''_s) \subseteq d''$$

$$|(h_c, h_s, d)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \mathrm{dom}(h'_c) = \emptyset \wedge \mathrm{dom}(h'_s) = \emptyset \\ (h'_c, h'_s, \bot) & \text{otherwise} \end{cases}$$

$$\text{where } h'_c \stackrel{\text{def}}{=} \lambda i. \begin{cases} |h_c(i)| & \text{if } i \in \mathrm{dom}(h_c) \wedge |h_c(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } h'_s \stackrel{\text{def}}{=} \text{likewise}$$

We note that composition requires checking for the presence of the index in both maps; if the first state has a binding in its concrete heap while the second has a binding in its symbolic heap, both need to be composed and then verify again if the result of the composition is concrete or symbolic.

Its predicates and actions are the same as for PMap: $\mathcal{A} = \mathbb{S}.\mathcal{A}$ and $\Delta = \mathbb{S}.\Delta \uplus \{\texttt{domainset}\}$. We assume the engine also provides an `is_concrete` function that given an expression returns true if it is concrete; this can be implemented by recursing through the expression's AST.

We again only define the rules for the `get` and `set` helper functions – everything else behaves the same (showing the advantage of having these two functions abstracted away).

$$\text{Given } \mathit{wrap}(h_c, h_s, d) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \mathrm{dom}(h_c) = \emptyset \wedge \mathrm{dom}(h_s) = \emptyset \wedge d = \emptyset \\ (h_c, h_s, d) & \text{otherwise} \end{cases}$$

$$\mathit{unwrap}(s) \stackrel{\text{def}}{=} \begin{cases} ([], [], \emptyset) & \text{if } s = \bot \\ (h_c, h_s, d) & \text{if } s = (h_c, h_s, d) \end{cases}$$

SplitPMapGetMatchCon

$$\frac{(h_c, h_s, d) = \mathit{unwrap}(s) \qquad \texttt{is\_concrete}\ i \qquad i \in \mathrm{dom}(h_c) \qquad s_i = h_c(i)}{\texttt{get}(s, i) \rightsquigarrow (i, s_i, [])}$$

SplitPMapGetMatchSym

$$\frac{(h_c, h_s, d) = \mathit{unwrap}(s) \qquad i \in \mathrm{dom}(h_s) \qquad s_i = h_s(i)}{\texttt{get}(s, i) \rightsquigarrow (i, s_i, [])}$$

SplitPMapGetBranch

$$\frac{\begin{array}{c}(h_c, h_s, d) = \mathit{unwrap}(s) \\ h_{all} = h_c \cup h_s \qquad i \notin \mathrm{dom}(h_{all}) \qquad i' \in \mathrm{dom}(h_{all}) \qquad s_{i'} = h_{all}(i')\end{array}}{\texttt{get}(s, i) \rightsquigarrow (i', s_{i'}, [i = i'])}$$

SplitPMapGetAdd

$$\frac{(h_c, h_s, d) = \mathit{unwrap}(s) \qquad h_{all} = h_c \cup h_s \qquad i \notin \mathrm{dom}(h_{all}) \qquad d \neq \bot}{\texttt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h_{all}) \wedge i \in d])}$$

SplitPMapGetBotDomain

$$\frac{(h_c, h_s, d) = \mathit{unwrap}(s) \qquad h_{all} = h_c \cup h_s \qquad i \notin \mathrm{dom}(h_{all}) \qquad d = \bot}{\texttt{get}(s, i) \rightsquigarrow (i, \bot, [i \notin \mathrm{dom}(h_{all})])}$$

SplitPMapSetSomeCon

$$\frac{\begin{array}{c}(h_c, h_s, d) = \mathit{unwrap}(s) \qquad s_i \neq \bot \qquad \texttt{is\_concrete}_\Sigma\ s_i \\ \texttt{is\_concrete}\ i \qquad h'_c = h_c[i \leftarrow s_i] \qquad h'_s = h_s[i \not\leftarrow] \qquad s' = \mathit{wrap}(h'_c, h'_s, d)\end{array}}{\texttt{set}(s, i, s_i) \rightarrow s'}$$

SplitPMapSetSomeSym

$$\frac{\begin{array}{c}(h_c, h_s, d) = \mathit{unwrap}(s) \qquad s_i \neq \bot \qquad \neg(\texttt{is\_concrete}_\Sigma\ s_i \vee \texttt{is\_concrete}\ i) \\ h'_c = h_c[i \not\leftarrow] \qquad h'_s = h_s[i \leftarrow s_i] \qquad s' = \mathit{wrap}(h'_c, h'_s, d)\end{array}}{\texttt{set}(s, i, s_i) \rightarrow s'}$$

SplitPMapSetNone

$$\frac{\begin{array}{c}(h_c, h_s, d) = \mathit{unwrap}(s) \\ s_i = \bot \qquad h'_c = h_c[i \not\leftarrow] \qquad h'_s = h_s[i \not\leftarrow] \qquad s' = \mathit{wrap}(h'_c, h'_s, d)\end{array}}{\texttt{set}(s, i, s_i) \rightarrow s'}$$

We note here that this optimisation comes with a cost: if syntactic matches are rare, then the two maps need to be merged for the branching check. Furthermore, modifying the map also requires removing the binding from the other map, as the state might have been there and changed (for instance, if the wrapped state was symbolic, but was modified to be concrete, then it needs to be removed from the symbolic map). Verifying if a given state fragment is concrete or not also comes with a cost, as it requires traversing the entire

state, which may be expensive – though this could be partly solved by caching whether it's concrete or not, and invalidating or modifying the cache when the state is modified. This caching has not been implemented, as in practice the cost of verifying concreteness is less than the gain of the optimisation.

In evaluation we will discuss the impact this optimisation has. *I need to measure the proportion of size between concrete/symbolic, to see in what executions it's more useful and in which it's not. Similarly, need to profile the time for a lookup according depending on the size of the joined map!*

1.5.3 Abstract location PMap

This second optimisation is new to Gillian, and uses *abstract locations* (ALocs). We expand the definition of symbolic values, `SVal`, with values of type "`ALoc`", akin to symbolic variables. These are a form of *semantic hash consing*: they can be used to distinguish different locations from their name, when not doing "matching". *Matching* is a mode that is enabled when consuming or producing, and that is disabled when executing actions and doing substitutions. When matching is disabled, two given abstract locations that are syntactically different (have different names) are semantically different (are distinct), whereas when matching is enabled then two syntactically different abstract locations may be equal, depending on the current path condition, which may lead to branching (in which cases the two states at the clashing locations are composed). This is a powerful optimisation: inside the body of a function, all lookups are branchless, as two syntactically different ALocs are considered different (matching is disabled), however once the post-condition is consumed then locations may clash and we thus merge together states at clashing locations; this ensures branching caused by addresses only happens at the very start and very end of the verification of a function.

This has the advantage of allowing storing locations in the map as a string (their name) rather than an expression, and checking for the presence of the string rather than requiring to attempt SAT checks for every index. The cost of this is when resolving an expression into an ALoc: if it already is of type `ALoc` then nothing is needed, but in the case in which it is a symbolic variable we must first check if it already equals an ALoc in the path condition, and if not instantiate a new ALoc, add the fact it is equal to the symbolic variable in the symbolic variable, and proceed.

A limitation of this optimisation is that only abstract locations may be used as the domain type; it is thus a specialisation of PMap, where $\text{PMap}_{\text{Split}}$ was a more general optimisation.

Let's first define the RA for this optimisation, $\text{PMap}_{\text{ALoc}}$.

$$\text{PMap}_{\text{ALoc}}(\mathbb{S}) \stackrel{\text{def}}{=} \text{Str} \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathcal{P}(\text{Str})^?$$

$$(h, d) \cdot (h', d') \stackrel{\text{def}}{=} (h'', d'')$$

$$\text{where } h'' \stackrel{\text{def}}{=} \lambda i. \begin{cases} h(i) \cdot h'(i) & \text{if } i \in \text{dom}(h) \cap \text{dom}(h') \\ h(i) & \text{if } i \in \text{dom}(h) \setminus \text{dom}(h') \\ h'(i) & \text{if } i \in \text{dom}(h') \setminus \text{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' \stackrel{\text{def}}{=} \begin{cases} d & \text{if } d' = \bot \\ d' & \text{if } d = \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' = \bot \vee \text{dom}(h'') \subseteq d''$$

$$|(h, d)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \text{dom}(h') = \emptyset \\ (h', \bot) & \text{otherwise} \end{cases}$$

$$\text{where } h' \stackrel{\text{def}}{=} \lambda i. \begin{cases} |h(i)| & \text{if } i \in \text{dom}(h) \wedge |h(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

We make use of the helper function $\texttt{to\_aloc} : \Pi \to \text{Val} \to \text{Str}^?$, which returns the name of the abstract location matching a given value, if it exists. This function receives the path condition, as for instance given a symbolic variable $\hat{x}$ with the path condition $\hat{x} = \text{aloc}(\texttt{loc\_x})$, the path condition is required to know what abstract location is associated to $\hat{x}$. The engine presented thus far however never receives the path condition, as to ensure it can only be strengthened – this is not the case in Gillian, where this optimisation was first implemented. We thus assume that `to_aloc` is capable of reading the current path condition without getting it as an input, to avoid modifying the previously defined signatures, noting it is trivial to add the path condition as a parameter of `execute_action`, `consume` and `produce`. We thus instead use the signature $\texttt{to\_aloc} : \text{Val} \to \text{Str}^?$. We also use the auxiliary function $\texttt{fresh\_aloc} : \texttt{unit} \to \text{Str}$ to generate fresh abstract location names.

We now present the rules for this state model; in particular, we again only need to concern ourselves with the `get` and `set` internal methods; actions and predicate consumption and production remain the same.

ALocPMapGetMatch

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad a = \texttt{to\_aloc}\ i \qquad a \neq \bot \qquad a \in \text{dom}(h) \qquad s_a = h(a)}{\texttt{get}(s, i) \rightsquigarrow (\text{aloc}(a), s_a, [])}$$

ALocPMapGetMatchBot

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad a = \texttt{to\_aloc}\ i \qquad a \neq \bot \qquad a \notin \text{dom}(h) \qquad d \neq \bot}{\texttt{get}(s, i) \rightsquigarrow (\text{aloc}(a), \bot, [\text{aloc}(a) \in d])}$$

ALocPMapGetNewLoc

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad a = \texttt{to\_aloc}\ i \qquad a = \bot \qquad d = \bot \qquad a' = \texttt{fresh\_aloc}\ ()}{\texttt{get}(s, i) \rightsquigarrow (\text{aloc}(a'), \bot, [i = \text{aloc}(a')])}$$

ALocPMapSetSome

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad a = \texttt{to\_aloc}\ i \qquad a \neq \bot \qquad s_i \neq \bot \qquad h' = h[a \leftarrow s_i] \qquad s' = \mathit{wrap}(h', d)}{\texttt{set}(s, i, s_i) \to s'}$$

ALocPMapSetNone

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad a = \texttt{to\_aloc}\ i \qquad a \neq \bot \qquad s_i = \bot \qquad h' = h[a \not\leftarrow] \qquad s' = \mathit{wrap}(h', d)}{\texttt{set}(s, i, s_i) \to s'}$$

The main advantage of $\text{PMap}_{\text{ALoc}}$ is made evident from the rules: looking up an index does not branch; it either is found directly in the map, or can be added. Branching is instead delegated to the SAT-solver of the engine, that when matching may decide two abstract locations are equal or not, resulting in two branches: one where they're different and one where they're equal and the states at both locations are composed.

Chapter 2

Soundness of allocation in PMap

2.1 Current State

We initially defined PMap as:

$$\textsc{PMap}(I, \mathbb{S}) \stackrel{\text{def}}{=} I \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathcal{P}(I)^?$$

$$(h, d) \cdot (h', d') \stackrel{\text{def}}{=} (h'', d'')$$

$$\text{where } h'' \stackrel{\text{def}}{=} \lambda i. \begin{cases} h(i) \cdot h'(i) & \text{if } i \in \text{dom}(h) \cap \text{dom}(h') \\ h(i) & \text{if } i \in \text{dom}(h) \setminus \text{dom}(h') \\ h'(i) & \text{if } i \in \text{dom}(h') \setminus \text{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' \stackrel{\text{def}}{=} \begin{cases} d & \text{if } d' = \bot \\ d' & \text{if } d = \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' = \bot \vee \text{dom}(h'') \subseteq d''$$

$$|(h, d)| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \text{dom}(h') = \emptyset \\ (h', \bot) & \text{otherwise} \end{cases}$$

$$\text{where } h' \stackrel{\text{def}}{=} \lambda i. \begin{cases} |h(i)| & \text{if } i \in \text{dom}(h) \wedge |h(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

The `alloc` action could then be executed; if the domain set was present, it was simply extended with the new index:

PMapAlloc

$$\frac{d \neq \bot \qquad i \notin SV \qquad i \notin \mathrm{dom}(h) \qquad i \notin d \qquad s_i = \texttt{instantiate}(\vec{v}_i) \qquad h' = h[i \leftarrow s_i] \qquad d' = d \uplus \{i\}}{\texttt{alloc}(SV, (h, d), \vec{v}_i) \rightsquigarrow (\texttt{Ok}, (h', d'), [i], [i = i])}$$

PMapAllocBot

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad d = \bot \qquad i \notin SV \qquad i \notin \mathrm{dom}(h) \qquad s_i = \texttt{instantiate}(\vec{v}_i) \qquad h' = h[i \leftarrow s_i]}{\texttt{alloc}(SV, s, \vec{v}_i) \rightsquigarrow (\texttt{Ok}, (h', \bot), [i], [i = i])}$$

This however breaks Frame subtraction:

Let $\sigma = ([0 \mapsto a], \bot), \quad \sigma_f = ([], \{0\})$

$\sigma \cdot \sigma_f = ([0 \mapsto a], \bot) \cdot ([], \{0\}) = ([0 \mapsto a], \{0\})$

We have that $\texttt{alloc}(\sigma \cdot \sigma_f, []) \rightsquigarrow (o, ([0 \mapsto a, 1 \mapsto b], \{0, 1\}), [])$ with $o = \texttt{Ok}$

and $\texttt{alloc}((\sigma, \bot), []) \rightsquigarrow (o', ([0 \mapsto a, 1 \mapsto b], \bot), [])$ with $o' = \texttt{Ok}$

$o' \neq \texttt{Miss}$ but $([0 \mapsto a, 1 \mapsto b], \{0, 1\}) \neq ([0 \mapsto a, 1 \mapsto b], \bot) \cdot ([], \{1\})$ since that's undefined, because

$\mathrm{dom}([0 \mapsto a, 1 \mapsto b]) = \{0, 1\} \not\subseteq \{1\}$

To solve this, `alloc` on a missing domain set should result in a `Miss`, and only succeed when owning the domain set:

PMapAllocMiss

$$\frac{(h, d) = \mathit{unwrap}(s) \qquad d = \bot}{\texttt{alloc}(SV, s, \vec{v}_i) \rightsquigarrow (\texttt{Miss}, s, [\texttt{`domainset'}], [])}$$

2.2 Concurrency

This however does not work in a concurrency setting, as two threads cannot simultaneously own and modify the domain set. This means that the partial semantics of `alloc` shown above are not compatible with full semantics in which this is permitted (as is the case, for instance, in the traditional model of separation logic).

A solution to this is to make the domain set a set of all indices *known* to exist – indices not in it don't necessarily mean an out of bounds, but may mean the index was created in

another thread that hasn't "rejoined" into the current thread. Its RA is defined as:

$$\begin{aligned}
&\textsc{PMap}_{\textsc{Fixed}}(I, \mathbb{S}) \stackrel{\text{def}}{=} I \xrightarrow{\mathit{fin}} \mathbb{S}.\Sigma \times \mathcal{P}(I) \\
&\quad (h, d) \cdot (h', d') \stackrel{\text{def}}{=} (h'', d \cup d') \\
&\qquad \text{where } h'' \stackrel{\text{def}}{=} \lambda i. \begin{cases} h(i) \cdot h'(i) & \text{if } i \in \text{dom}(h) \cap \text{dom}(h') \\ h(i) & \text{if } i \in \text{dom}(h) \setminus \text{dom}(h') \\ h'(i) & \text{if } i \in \text{dom}(h') \setminus \text{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases} \\
&\qquad \text{and } \text{dom}(h'') \subseteq (d \cup d') \\
&\quad |(h, d)| \stackrel{\text{def}}{=} (h', d) \\
&\qquad \text{where } h' \stackrel{\text{def}}{=} \lambda i. \begin{cases} |h(i)| & \text{if } i \in \text{dom}(h) \wedge |h(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

This means however that $\textsc{PMap}_{\textsc{Fixed}}$ cannot be exclusively owned, as a partial map with a subset of its domain can always be composed with it. Its rule for allocation is the same as before, except there is no miss case:

PMapFixedAlloc

$$\frac{i \notin \text{dom}(h) \qquad i \notin d \qquad s_i = \texttt{instantiate}(\vec{v}_i) \qquad h' = h[i \leftarrow s_i] \qquad d' = d \cup \{i\}}{\texttt{alloc}(SV, (h, d), \vec{v}_i) \rightsquigarrow (\texttt{Ok}, (h', d'), [i], [i = i])} \; i \notin SV$$

Rules are otherwise similar, with consuming the domainset not modifying it, and producing it producing the union with the current domain set.

While allowing concurrent allocation, this state model does not fulfill Compatibility in its current state. Indeed, it cannot reliably distinguish out of bounds from missing outcomes on accesses - as the index not being in the domain set may mean it still exists in another thread.

Chapter 3

// TODO:

Priority	Name
0	**Redefine state models**: ~~Exc, Ag, Frac, PMap, List, GMap, Freeable, Sum, Product~~, DynPMap
0	**Define optimised state models**: ~~ALocPMap, SplitPMap~~
1	**Define stacks**: WISL, JS, C
1	**Write comparative evaluation**: WISL, JS, C, (Viper?)
1	**Write absolute evaluation**: PMap perf according to size, split PMap split rate
2	**Define ea/consume/produce for** $\perp$: rules, proofs?
3	**Add `LFail` and `Miss` to consume**: rules, proofs?
4	**Redefine fixes as purely assertions**: signature, biabduction rules, proof of soundness

Small things to do/fix:

- Mention that CSE2 has SV, but keep it implicit here.
- Remove notion of compatibility

Bibliography

[1] R. Jung, *Understanding and evolving the rust programming language*, 2020. DOI: http://dx.doi.org/10.22028/D291-31946.

[2] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, e20, 2018. DOI: 10.1017/S0956796818000151.

[3] L. Birkedal, *Iris: Higher-order concurrent separation logic - lecture 10: Ghost state*, 2020. [Online]. Available: https://iris-project.org/tutorial-pdfs/lecture10-ghost-state.pdf.

[4] S.-E. Ayoun, "Gillian: Foundations, Implementation and Applications of Compositional Symbolic Execution."

[5] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022. DOI: 10.1145/3527325. [Online]. Available: https://doi.org/10.1145/3527325.

[6] A. Lööw, D. Nantes Sobrinho, S.-E. Ayoun, C. Cronjäger, P. Maksimović, and P. Gardner, "Compositional Symbolic Execution for Correctness and Incorrectness Reasoning."

[7] A. Bizjak and L. Birkedal, "On Models of Higher-Order Separation Logic," *Electronic Notes in Theoretical Computer Science*, vol. 336, pp. 57–78, 2018, The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2018.03.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066118300197.

[8] R. Bornat, C. Calcagno, P. Hearn, and H. Yang, "Fractional and counting permissions in separation logic,"

[9] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *SIGPLAN Not.*, vol. 40, no. 1, pp. 259–270, 2005, ISSN: 0362-1340. DOI: 10.1145/1047659.1040327. [Online]. Available: https://doi.org/10.1145/1047659.1040327.

[10] R. Dockins, A. Hobor, and A. W. Appel, "A Fresh Look at Separation Algebras and Share Accounting," in *Programming Languages and Systems*, Z. Hu, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 161–177, ISBN: 978-3-642-10672-9.

[11] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," in *Computer Science Logic*, L. Fribourg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19, ISBN: 978-3-540-44802-0.

[12] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.