IMPERIAL

Project Notes

Background & Progress Report

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

July 20, 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 equiped 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 interfae 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 unweildy requirements to prove soundness – this comes into play for the `Freeable` memory 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.

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 $\sigma \neq \lightning$, $\overline{\mathcal{V}}(\sigma)$ 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

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 $a \cdot \bot \stackrel{\text{def}}{=} \bot \cdot a \stackrel{\text{def}}{=} \bot$

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

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

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

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

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

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 $\alpha(SV, \hat{\sigma}, \vec{v_i}) \rightsquigarrow (o, \hat{\sigma}', \hat{\pi}, \vec{v_o})$.

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, for instance, 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 [3]. As such, the only valid outcomes here are `Ok` and `Err`.

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).

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

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.

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.

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 incompati-

bility 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.

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.

1.3.3 Bi-abduction Engine

To support bi-abduction in the style of Infer:Pulse [4], `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 exisentially 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 $\texttt{apply\_fix} : \Sigma \to \texttt{fix\_t} \to \Sigma$ 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 [3], 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 modellable 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, $\texttt{SubError}(i, \texttt{MissingValue})$. When getting the fix, PMap can then call Exc.`fix`, which returns $\exists x.\, \langle \texttt{points\_to} \rangle\, (; x)$, and lift the fix by adding the index as an in-argument, resulting in the final fix $\exists x.\, \langle \texttt{points\_to} \rangle\, (i; x)$, which is a valid assertion and can be added to the UX specification for this execution.

TODO: See how the Iris frame preserving transition $a \rightsquigarrow b$ relates to frame addition/substraction. Seems like a stronger version, ie. semantics that succesfully update state from a to b are necessarily frame preserving?

1.4 RAs and State Models

We now define the state transformers defined in [3], taking advantage of RAs.

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:

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

Note that the above definition is identical 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`.

LoadOk

$$\frac{}{\texttt{load}(SV, \mathrm{ex}(x), []) \rightsquigarrow (\texttt{Ok}, \mathrm{ex}(x), [], [x])}$$

LoadMiss

$$\frac{}{\texttt{load}(SV, \bot, []) \rightsquigarrow (\texttt{Miss}, \mathrm{ex}(x), [], [])}$$

StoreOk

$$\frac{}{\texttt{store}(SV, \mathrm{ex}(x), [x']) \rightsquigarrow (\texttt{Ok}, \mathrm{ex}(x'), [], [])}$$

StoreMiss

$$\frac{}{\texttt{store}(SV, \bot, [x']) \rightsquigarrow (\texttt{Miss}, \mathrm{ex}(x), [], [])}$$

1.4.2 Partial Product

We may now attempt to unify the definitions of PMap and List. For this, we first need to define the RA of *partial products*, denoted $A \bowtie B$. This product supports having one side be empty, but not both – this is different from the usual product RA, that must have both sides have a value. This is needed, to allow for products that are *exclusively owned*, meaning no additional resources can be composed with them, which is crucial for the soundness of freeing memory. Another interpretation of the partial product is that it is of type $A + B + (A * B)$. Given two RAs A and B, their partial product RA is $A \bowtie B$, as defined in Figure 1.2.

$$
\begin{aligned}
A \bowtie B &\stackrel{\text{def}}{=} l(a : A) \mid r(b : B) \mid lr(a : A, b : B) \\
l(a) \cdot l(a') &\stackrel{\text{def}}{=} l(a \cdot a') \\
r(b) \cdot r(b') &\stackrel{\text{def}}{=} r(b \cdot b') \\
l(a) \cdot r(b) \stackrel{\text{def}}{=} r(b) \cdot l(a) &\stackrel{\text{def}}{=} lr(a, b) \\
lr(a, b) \cdot lr(a', b') &\stackrel{\text{def}}{=} lr(a \cdot a', b \cdot b') \\
l(a) \cdot lr(a', b) \stackrel{\text{def}}{=} lr(a', b) \cdot l(a) &\stackrel{\text{def}}{=} lr(a \cdot a', b) \\
r(b) \cdot lr(a, b') \stackrel{\text{def}}{=} lr(a, b') \cdot r(b) &\stackrel{\text{def}}{=} lr(a, b \cdot b') \\
|l(a)| &\stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } |a| = \bot \\ l(|a|) & \text{otherwise} \end{cases} \\
|r(b)| &\stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } |b| = \bot \\ r(|b|) & \text{otherwise} \end{cases} \\
|lr(a, b)| &\stackrel{\text{def}}{=} \begin{cases} lr(|a|, |b| & \text{if } |a| \neq \bot \wedge |b| \neq \bot \\ l(|a|) & \text{if } |a| \neq \bot \\ r(|b|) & \text{if } |b| \neq \bot \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1.2: Definiton of the partial product RA

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 $ex(a) \times ex(b)$, one can't express this as the composition of some $ex(a) \times \bot$ and $\bot \times ex(b)$, which becomes needed for some state transformers (notably, the map and list). This is in turn trivial with the partial product, with both states being. 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:

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

1.4.3 General Map

Reusing the above examples, we may unify the PMap and List transformers, with the *global map* transformer, shortened GMap. It is parametric on an indexing sort $I \subseteq \text{Val}$, a codomain state model Σ and a *discriminator* state model Σ_D that must expose a function $\texttt{is\_within} : \Sigma_D \to I \to \mathbb{B}$. It's states are then of the form $I \xrightarrow{fin} \Sigma$ for the full state model, and $I \xrightarrow{fin} \Sigma \bowtie \Sigma_D$ for the compositional and symbolic state models.

For soundness, the `is_within` function must be true if and only if a singleton partial map with the given index could be validly composed into the state while upholding the desired GMap's invariant. It is used when an index not present in the map is accessed, to tell apart misses from errors (out of bounds accesses). A consequence of this is that when the state is *full*, `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 it must always be false – this explains why the full version of GMap can be $I \xrightarrow{fin} \Sigma$ without the discriminator. If we write $\operatorname{dom} \sigma$ 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 $i \notin \operatorname{dom} \sigma \wedge \sigma \in \underline{\Sigma} \Rightarrow \neg(\texttt{is\_within}\ \sigma\ i)$.

To replicate the usual PMap, one would have $\Sigma_D = \mathrm{Ex}(\mathcal{P}(I))$, with `is_within` true if the value is in the set: $\texttt{is\_within}_{\mathrm{PMap}}\ \sigma_D\ i = i \in \sigma_D$. For List, one has $\Sigma_D = \mathrm{Ex}(\mathbb{N})$ with $\texttt{is\_within}_{\mathrm{List}}\ \sigma_D\ i = 0 \leq i < \sigma_D$. Note here that state transformer may automatically assume the outcome is a `Miss` if the key is not found in the map and the descriminator state isn't set (ie. the state is $l(x)$ for some x), as indeed a state fragment containing only the discriminator can always be composed with it to complete the `Miss`.

Another interesting point is that the discriminator is a state model; this allows it to define its own predicates to control its state. While here both examples use a Ex state model, one could imagine more complex discriminators where information is split across several predicates, which GMap would support too. However, one may note that the discriminator shouldn't define any: because it doesn't exist for the full version of GMap, allowing actions to only exist in the compositional state model would break compatibility.

TODO: Check the above with Sacha: must the discriminator be removed for the full state? Can we not just initialise it to a default value and go from there?

Chapter 2

Tasks to do

Priority	Name
0	**Redefine state models**: Exc, Ag, Frac, PMap, List, Freeable, Sum, Product
0	**Define optimised state models**: ALocPMap, SplitPMap
1	**Define ea/consume/produce for** $\perp$: rules, proofs?
2	**Add `LFail` and `Miss` to consume**: rules, proofs?
3	**Redefine fixes as purely assertions**: signature, biabduction rules, proof of soundness

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

[4] 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.