Compositional Symbolic Execution for the Next 700 Memory Models

ANDREAS LÖÖW, Imperial College London, United Kingdom
SEUNG HOON PARK, Imperial College London, United Kingdom
DANIELE NANTES-SOBRINHO, Imperial College London, United Kingdom
SACHA-ÉLIE AYOUN, Imperial College London, United Kingdom
OPALE SJÖSTEDT, Imperial College London, United Kingdom
PHILIPPA GARDNER, Imperial College London, United Kingdom

Multiple successful compositional symbolic execution (CSE) tools and platforms exploit separation logic (SL) for compositional verification and/or incorrectness separation logic (ISL) for compositional bug-finding, including VeriFast, Viper, Gillian, CN, and Infer-Pulse. Previous work on the Gillian platform, the only CSE platform that is parametric on the memory model, meaning that it can be instantiated to different memory models, suggests that the ability to use custom memory models allows for more flexibility in supporting analysis of a wide range of programming languages, for implementing custom automation, and for improving performance. However, the literature lacks a satisfactory formal foundation for memory-model-parametric CSE platforms.

In this paper, inspired by Gillian, we provide a new formal foundation for memory-model-parametric CSE platforms. Our foundation advances the state of the art in four ways. First, we mechanise our foundation (in the interactive theorem prover Rocq). Second, we validate our foundation by instantiating it to a broad range of memory models, including models for C and CHERI. Third, whereas previous memory-model-parametric work has only covered SL analyses, we cover both SL and ISL analyses. Fourth, our foundation is based on standard definitions of SL and ISL (including definitions of function specification validity, to ensure sound interoperation with other tools and platforms also based on standard definitions).

CCS Concepts: • **Theory of computation → Logic and verification**; **Automated reasoning**; **Separation logic**; **Program reasoning**.

Additional Key Words and Phrases: symbolic execution, memory model, separation logic, incorrectness logic

ACM Reference Format:
Andreas Lööw, Seung Hoon Park, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Opale Sjöstedt, and Philippa Gardner. 2025. Compositional Symbolic Execution for the Next 700 Memory Models. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 373 (October 2025), 28 pages. https://doi.org/10.1145/3763151

1 Introduction

Multiple successful analysis tools and platforms provide compositional verification and/or compositional bug-finding for heap-manipulating programs, including VeriFast [21], Viper [38], Gillian [19, 32, 35], CN [45], Infer-Pulse [26], by animating their automated and semi-automated reasoning using *compositional symbolic execution (CSE)* grounded on ideas from separation logic (SL) [40, 49]

Authors' Contact Information: Andreas Lööw, Imperial College London, London, United Kingdom, a.loow@ic.ac.uk; Seung Hoon Park, Imperial College London, London, United Kingdom, s.park23@ic.ac.uk; Daniele Nantes-Sobrinho, Imperial College London, London, United Kingdom, d.nantes-sobrinho@ic.ac.uk; Sacha-Élie Ayoun, Imperial College London, London, United Kingdom, s.ayoun17@ic.ac.uk; Opale Sjöstedt, Imperial College London, London, United Kingdom, opale.sjostedt23@ic.ac.uk; Philippa Gardner, Imperial College London, London, United Kingdom, p.gardner@ic.ac.uk.

ACM 2475-1421/2025/10-ART373
https://doi.org/10.1145/3763151

and/or incorrectness separation logic (ISL) [46] (usually bottoming out in a call to an underlying SMT solver, such as Z3 [14]). In this case, compositional (or functionally compositional) reasoning means that it is scalable in that the analysis works on functions in isolation, at any point in the codebase, and then records the results in simple function specifications that can be used in broader calling contexts. SL provides a good grounding for sound verification (proving the absence of bugs) through compositional *over-approximate (OX) reasoning*; ISL provides a good grounding for sound bug-finding (proving the presence of bugs) through compositional *under-approximate (UX) reasoning*. Important examples of CSE analyses based on the two logics include OX function-specification verification (implemented in, e.g., VeriFast, Viper, Gillian, and CN) and UX true bug-finding based on bi-abduction [26] (implemented in, e.g., Infer-Pulse and Gillian).

A key challenge with the design of CSE platforms that aim for wide applicability is to manage the diverse range of memory models employed across different applications.[1] This need for different memory models arises from multiple sources. First, of course, different programming languages are defined over different language memory models. Second, different analyses are defined over different types of memory ghost state, e.g., ghost state for different kinds of ownership disciplines, such as exclusive ownership vs. fractional ownership, or negative-information ghost state used in UX analyses to ensure UX compositionality. Third, even choosing the programming language and the analysis still does not determine the memory model: there is *no one-size-fits-all* memory models because the different axes of the memory-model design space are often-times in antagonistic relationship with each other: e.g., the choice of what part of the language to be analysed, accuracy and abstraction level of the model, implementation effort of the model, performance of the model, and automation/annotation burden associated with the analysis of the model. We cannot move freely along different axes in the design space and therefore must solve a difficult trade-off problem when selecting a memory model to use: e.g., a complex memory model might give better performance than a simple memory model but will require more implementation effort.

The analysis platform Gillian stands out as the only CSE platform that faces this memory-model challenge head-on. Gillian is the only CSE platform that is *parametric* on the memory model, meaning that no memory model is hard-coded into the platform and instead the platform can be instantiated to different memory models, depending on which model has the best position in the model design trade-off space for a situation at hand. All other CSE platforms are *monomorphic* on the memory model in that they only support one fixed memory model that has been hard-coded into the platform. It is therefore awkward, or impossible, to use the memory model that is most appropriate for a situation at hand since it must be encoded into the fixed memory model of the platform.

Literature gap. No previous work provides a satisfactory formal foundation for CSE platforms that are parametric on the memory model; in particular, a formal foundation for Gillian's approach to memory-model parametricity is missing. There was some initial work on the foundations of Gillian [19, 35], which outlined mathematical definitions and gave a sketch of a soundness proof for parts of the CSE engine of Gillian. This work, however, suffers from *four weaknesses*:

(1) it was not mechanised;
(2) it did not prove that any of their memory-model instances were sound, and thus did not validate their definitions and conjectures;
(3) it only covered SL-based analyses, not ISL-based analyses;
(4) it did not use standard SL definitions, such as the definition of function specification validity, thus making the engine awkward to interoperate with other analysis tools and platforms.

[1] To avoid confusion: whereas some authors reserve the term memory model for weak-memory concurrency, we use the term broadly in this work (the many different ways of representing, updating, and analysing the heap).

Table 1. Summary of our memory-model instances. The columns "OX" and "UX" specify whether the model is OX sound and/or UX sound; "Rocq kLoc" specifies the size of the Rocq proof script file for the model; "Origin" specifies which CSE platforms have implemented the model. The star ("*") in the Rocq column for the OOP model specifies that we did not mechanise the model because of its large overlap with the block-offset model, and the double star ("**") for the CHERI model specifies that the mechanised proof only covers OX soundness (UX soundness is left for future work because of the size of the task).

Memory-model name	§	OX	UX	Rocq kLoC	Origin
Linear model (running example in paper)	6.1	✓	✓	1	
Linear model with unique-match branching	6.1	✓	✓	1	
Linear model with cut branching	6.1		✓	≈ 0.5	
Linear model without negative information	6.1	✓		≈ 0.5	
Fractional ownership model	6.2	✓	✓	2	
Block-offset model for C	6.3	✓	✓	4	Gillian
Model for OOP languages (e.g., JavaScript)	6.4	✓	✓	0*	Gillian
CHERI-assembly model	6.5	✓	✓	19**	New model
VeriFast-and-Viper-inspired model for C	6.6	✓		≈ 0.5	VeriFast and Viper

Other previous work on the foundations of CSE have only covered tools and platforms that are monomorphic on the memory model [13, 22, 32, 58]. (We discuss related work in more detail in §7.)

Contribution. In this paper, we contribute a new CSE theory that addresses all four weaknesses in the current formal foundations of CSE platforms that are parametric on the memory model. Our new CSE theory is inspired by the design of Gillian but is independent of its particular implementation; we have designed our theory to be a CSE analogue of separation logics that are parametric on the memory model, such as abstract separation logic [11] and subsequent generic/modular/parametric separation-logic frameworks like the views framework [15, 47] and, perhaps the most well-known example, the Iris framework [24].

An important strength of our CSE theory is that it is remarkably simple; in fact, its definitions and metatheory are not much more complex than existing monomorphic CSE theories. *This suggests that, while there are clear advantages, there are no clear disadvantages in making CSE platforms parametric on the memory model.*

Technically, our new CSE theory, which we have mechanised in Rocq [50], consists of:

- a definition of "memory model" in the CSE setting, including, two sets of OX and UX soundness requirements on memory models;
- a formal semantics for a CSE engine that is parametric on the memory model;
- two soundness results for the engine: if a memory model satisfies our OX/UX soundness requirements, then the engine is OX/UX sound when instantiated with the memory model.

To validate our CSE theory and show that it has broadly applicability, we instantiate our theory to a broad collection of memory models ranging from models for low-level languages like assembly and C to high-level languages like JavaScript, as summarised in Tab. 1. In more detail: in §6.1 and §6.2, we cover the standard models used in theoretical investigations into SL and ISL, which we call linear memory models. In §6.1, we show that multiple variants of these linear memory models fit into our theory, including OX-and-UX sound models, OX-only models, and UX-only models. In §6.2, to show that different ownership disciplines fit into our theory, we instantiate our theory with a linear memory model implementing fractional ownership rather than standard exclusive ownership. In §6.3, we shift the discussion towards more realistic memory models, starting of the

discussion with a memory model inspired by the memory model of the CompCert compiler. The memory model is implemented in Gillian and has been used in Gillian-based teaching. In §6.4, to show that memory models for high-level languages like object-oriented languages also fit our theory, we discuss the JavaScript memory model implemented in Gillian. In §6.5, we discuss a new memory model for CHERI, which is the largest model we have mechanised. As this model is new and, hence, has not been implemented in any CSE platform, the model shows that new models can be designed using only our CSE theory as guidance. Lastly, in §6.6, to show broad CSE platform coverage, we discuss the hard-coded memory model of VeriFast and Viper.

Our main technical contributions can be summarised as follows:

- We provide the first foundation of CSE platforms that are parametric on the memory model (§5) that: (1) is mechanised, (2) is validated, (3) covers both SL- and ISL-based analyses, and (4) is interoperable.
- We demonstrate that two important analyses can be soundly hosted on top of our memory-model-parametric CSE engine (§5): namely, OX function-specification verification; and UX true bug-finding based on bi-abduction.
- We discuss instantiations of our CSE theory (§6), as summarised in Tab. 1.
- We make available all source code and proofs of our Rocq mechanisation of our CSE theory and its instantiations in the artefact of this paper (see our data-availability statement).

Scope limitations and caveats. For this paper, we only consider *sequential* memory models not *concurrent* memory models. We, however, believe our work is a useful starting point for future work on symbolic execution of different concurrent memory models. Additionally, we work with a simple demonstrator programming language, specifically, a memory-model-parametric variant of a standard (sequential) imperative language. In other words, to focus the discussion on our core contribution, which is memory-model parametricity, we do not vary other parts of the language.

2 Overview

In this section, we highlight the main *takeaways* of our new CSE theory. To be able to do so, we give a compressed overview of our theory. The *core contribution of our theory* is that it is parametric on a set of parameters that factors out its memory-model dependent part; the focus in this section is therefore these parameters.

2.1 Background: Monomorphic CSE Theory

Before introducing our new memory-model-parametric CSE theory, we give a summary of traditional memory-model-monomorphic CSE theory. The judgements we use in the summary are simplified judgements from our theory.

Engine architecture. In this paper, we work specifically with CSE engines implementing the consume-produce engine architecture. This is the architecture implemented by Gillian and other similar modern CSE engines like VeriFast and Viper. In this architecture, two operations called consume and produce are used to implement the execution of commands/constructs based on assertions (separation-logic points-to assertions, etc.), which are the commands/constructs making the engine a compositional engine, such as using function specifications to reason about function calls. In short, the consume operation takes as input an assertion and removes ("consumes") the corresponding symbolic state from the engine's current symbolic state and the produce operation also takes as input an assertion but instead adds ("produces") the corresponding symbolic state to the current symbolic state. For example, to execute a function call using a function specification, the precondition of the specification is first consumed and its postcondition is then produced.

Judgements. Assuming we work with (a monomorphised version of) our demonstrator language, a monomorphic CSE theory needs to specify at least the following judgements:

- $\sigma, \mathsf{C} \Downarrow \sigma'$ – judgement for the *concrete* semantics of the language, where C denotes a language command, σ is a concrete input state, and σ' is a concrete output state.
- $\sigma \models A$ – judgement for the satisfaction relation for assertions (used to, e.g., define the semantics of function specifications).
- $\hat{\sigma}, \mathsf{C} \Downarrow \hat{\sigma}'$ – judgement for the CSE engine, i.e., the *symbolic* semantics of the language, where $\hat{\sigma}$ and $\hat{\sigma}'$ are symbolic states.
- $\sigma \models \hat{\sigma}$ – judgement for the satisfaction relation for symbolic states, i.e., the relation between concrete and symbolic states.

The two important operations consume and produce are part of the definition of the symbolic engine, i.e., $\hat{\sigma}, \mathsf{C} \Downarrow \hat{\sigma}'$.

Definition of soundness. There are two standard soundness statements that relate the concrete and symbolic execution judgements: OX soundness and UX soundness [3] (although different papers use different terminology). OX soundness is the following relation between the two types of execution:

$$\sigma, \mathsf{C} \Downarrow \sigma' \land \sigma \models \hat{\sigma} \Rightarrow \exists \hat{\sigma}', \hat{\sigma}, \mathsf{C} \Downarrow \hat{\sigma}' \land \sigma' \models \hat{\sigma}'.$$

Intuitively, the relation enforces that all states reachable by concrete execution are reachable by symbolic execution, i.e., symbolic reachability overapproximates concrete reachability. UX soundness enforces the opposite relation, where, note, the universal quantification is over the final states:

$$\hat{\sigma}, \mathsf{C} \Downarrow \hat{\sigma}' \land \sigma' \models \hat{\sigma}' \Rightarrow \exists \sigma, \sigma, \mathsf{C} \Downarrow \sigma' \land \sigma \models \hat{\sigma}.$$

In more analysis application-oriented terms: OX soundness is a good foundation for verification and UX soundness is a good foundation for bug-finding. E.g., for verification: it follows from OX soundness that if we have proved that a behaviour (such as a bug) is unreachable using symbolic execution, then the behaviour is also unreachable by concrete execution.

2.2 The Step to Parametric CSE Theory

We now discuss how our CSE theory parameterises the judgements and soundness statements introduced above. We differentiate between two types of parameters, which we also refer to as *instance data*: *instance definitions* (abbreviation: "IDefs") and *instance properties* (abbreviation: "IProps"). Additionally, we group the parameters into the following *four abstractions*:

(1) concrete memory model (CMM) – the parameters of $\sigma, \mathsf{C} \Downarrow \sigma'$ (concrete semantics);
(2) resource model (RM) – the parameters of $\sigma \models A$ (satisfaction relation for assertions);
(3) symbolic memory model (SMM) – the parameters of $\hat{\sigma}, \mathsf{C} \Downarrow \hat{\sigma}'$ (symbolic semantics/engine) and $\sigma \models \hat{\sigma}$ (satisfaction relation for symbolic states);
(4) OX and UX soundness relations (RELs) between CMMs, RMs, and SMMs – the parameters of the soundness proofs of the engine.

Fig. 1 depicts the dependency structure of the four abstractions (blue boxes) and the engine definition and its soundness proofs (grey boxes). All IDefs. and IProps. parameters are summarised, respectively, in Tab. 2 and Tab. 3. We now introduce the parameters in more detail.

Concrete memory model and resource model. Because a CSE engine and a program logic for the same language have the same trusted computing base, the parameters of the concrete language and the assertion language of our CSE theory are the same as for comparable memory-model-parametric separation logics such as abstract separation logic [11] (we are here speaking in terms of big-picture

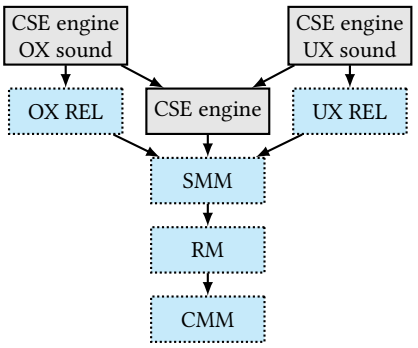


Fig. 1. Dependency structure of our theory.

Table 2. Summary of required IDefs.

#	Abs.	Description
1	CMM	Memory data type, empty memory, and composition operator
2	CMM	Concrete semantics of memory actions
3	RM	Resource satisfaction relation
4	SMM	Memory data type and empty memory
5	SMM	Memory satisfaction relation
6	SMM	Symbolic semantics of memory actions
7	SMM	Semantics of $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$

Table 3. Summary of required IProps., where the "Deps." column specifies the IDefs. dependencies.

#	Abs.	Deps.	Description
1	CMM	1	Memory forms a partial commutative monoid (PCM)
2	CMM	1 and 2	Memory actions satisfy OX/UX frame properties
3	REL	All except 3 and 7	OX/UX soundness of symbolic memory actions
4	REL	All except 2 and 6	OX/UX soundness of $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$

ideas, of course parameter details differ between different program logics). Therefore, we discuss the two abstractions concrete memory model and resource model together.

A concrete memory model, i.e., the parameters of $\sigma, \mathsf{C} \Downarrow \sigma'$ (concrete language), specifies: the data type of memory (IDef. 1) and the memory actions and their semantics (IDef. 2). Examples of common memory actions include memory read, memory write, allocation, etc. Analogous to the setup in program logics, we require that the data type of memory comes with a composition operator that forms a PCM together with the data type (IProp. 1) such that we can build the standard separation-logic infrastructure on top of the language. Additionally, to ensure that the concrete language satisfies the standard separation-logic frame properties, we require that the memory actions satisfy frame properties (IProp. 2) that we have derived from the standard properties.

A resource model, i.e., the parameters of $\sigma \models A$ (satisfaction relation for assertions), specifies: the resource assertions for the memory-model instance and their satisfaction relation (IDef. 3).[2] These resource assertions are the assertions that differ between memory models, the remaining assertion language is fixed. E.g., one common type of resource assertion is points-to assertions for heap cells (usually denoted $E_1 \mapsto E_2$). An alternative approach, followed by the original work on Gillian [35], is to define the meaning of assertions in terms of consume and produce instead of a traditional satisfaction relation. This approach requires less instance data but makes the theory awkward to connect to other formalisms since the meaning of assertions is nonstandard.

Takeaway. Concrete memory models and resource models should look familiar; they are analogous to instance data also required by memory-model-parametric separation logics.

[2]Our assertions are deeply embedded because our CSE engine must be able to pattern match over their structure. Note that some presentations of memory-model-parametric separation logics (e.g., the abstract separation logic paper [11]) shallowly embed their assertions, i.e., do not include an explicit satisfaction relation and instead define assertions to be sets of state.

Symbolic memory model. A symbolic memory model, i.e., the parameters of $\hat{\sigma}, \mathsf{C} \Downarrow \hat{\sigma}'$ (symbolic semantics/engine), specifies: the data type of symbolic memory (IDef. 4) and the satisfaction relation for symbolic memory (IDef. 5); the symbolic semantics of memory actions over the memory (IDef. 6); additionally, the consume and produce operations of the engine are parametrised by $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations for the resource assertions of the memory model (IDef. 7).

Note that, in contrast to concrete memories, we do not require that symbolic memories form PCMs or satisfy any frame properties. This is unlike the original work on Gillian [35], which defined concrete and symbolic memory models uniformly and therefore required the same instance data for both, i.e., required more instance data than us. Our work shows that the IDefs. and IProps. of a parametric CSE theory can be stated such that symbolic PCM and frame data is not needed for either the definition of the theory's engine or for its soundness proofs; instead, in the theory, all PCM reasoning and frame reasoning can be carried out at the concrete level.

Takeaway. Symbolic memory models and concrete memory models have different parameters and should therefore not be treated uniformly.

OX and UX soundness relations. Our soundness relations, i.e., the parameters of the soundness proofs, tie together concrete memory models, resource models, and symbolic memory models. The OX soundness of our CSE engine (Thm. 5.2) follows from a series of OX IProps., while UX soundness (Thm. 5.3) follows from a series of UX IProps. Specifically, we define what it means for symbolic memory actions (IProp. 3) and the $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations (IProp. 4) to be OX sound and require this as instance data. See again Tab. 3, which specifies which IDefs. each IProp. ties together. We give analogous IProps. for UX soundness.

Takeaway. From the Gillian implementation we know that the *definition* of a CSE engine can be built on top of a symbolic memory model, this paper shows that the same is true of the *soundness proof* of the engine; i.e., the soundness proof of the engine can be built on top of the soundness requirements for symbolic memory models as we define them in this paper (i.e., our IProps.).

Takeaway. Additionally, our work shows that it is possible to make a clear separation between OX and UX soundness requirements and lift the two to the full engine independently of each other. In other words, OX soundness and UX soundness are independent of each other.

2.3 Structure of Rest of Paper

The rest of the paper is structured as follows. In §3 to §5, we formally and incrementally present the parameters of our CSE theory. Throughout this presentation, we use a simple linear memory model as a running instantiation example. In §6, having introduced the parameters, we discuss other examples of memory models that fit our CSE theory.

3 Programming Language

We introduce the *syntax* and *concrete semantics* of our demonstrator programming language, which is parametrised by a *concrete memory model*, specifically, concrete memory data type (IDef. 1), memory actions (IDef. 2), and their associated IProps.

Syntax. The syntax of the language is standard except that it is equipped with a *memory-action command* $\vec{\mathsf{x}} := \alpha(\vec{\mathsf{E}})$, where $\alpha \in \mathsf{Str}$, which is given a semantics using the memory-model IDefs.

introduced below. The full definition of the syntax is as follows:

$$\begin{aligned} v \in \mathsf{Val} &::= \mathtt{null} \mid b \in \mathsf{Bool} \mid n \in \mathsf{Nat} \mid q \in \mathsf{Rat}^+ \mid s \in \mathsf{Str} \mid [\vec{v}] \qquad \mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots \in \mathsf{PVar} \\ \mathsf{E} \in \mathsf{PExp} &::= v \mid \mathsf{x} \mid \neg \mathsf{E} \mid \mathsf{E} = \mathsf{E} \mid \mathsf{E} \wedge \mathsf{E} \mid \mathsf{E} + \mathsf{E} \mid \mathsf{E} - \mathsf{E} \mid \mathsf{E} \,/\, \mathsf{E} \mid \mathsf{E} < \mathsf{E} \mid \ldots \\ \mathsf{C} \in \mathsf{Cmd} &::= \mathtt{skip} \mid \mathsf{x} := \mathsf{E} \mid \mathtt{if}\ (\mathsf{E})\ \mathsf{C}\ \mathtt{else}\ \mathsf{C} \mid \mathsf{C}; \mathsf{C} \mid \mathsf{y} := f(\vec{\mathsf{E}}) \mid \vec{\mathsf{x}} := \alpha(\vec{\mathsf{E}}) \end{aligned}$$

where the vector notation (e.g. $\vec{v}$) denotes a list, Val the set of values (Rat^+ is the positive rationals), PVar the set of program variables, PExp the set of expressions, and Cmd the set of commands.

Concrete semantics. To define the semantics of the language and to ensure that the language can be used in compositional reasoning, we require the following instance data:[3]

Instance Definition 1. We require a tuple $(\mathsf{CMem}, \mathcal{W}f, \mu_\emptyset, \cdot)$, where CMem is a set of memories, $\mathcal{W}f \subseteq \mathsf{CMem}$ is a well-formedness predicate, $\mu_\emptyset \in \mathsf{CMem}$ the empty-memory element, and $\cdot : (\mathsf{CMem}, \mathsf{CMem}) \rightharpoonup \mathsf{CMem}$ is a memory composition operator.[4] The empty memory must be well-formed and composition must maintain well-formedness.

Instance Definition Example 1. In our running example linear memory model, CMem is $\mathsf{Nat} \rightharpoonup_{fin} (\mathsf{Val} \uplus \{\emptyset\})$, where the symbol $\emptyset$ records that a memory cell has been freed. Tracking freed memory cells is a standard technique used in compositional UX reasoning [46] to ensure that the memory model satisfies UX frame (IProp. 2). All memories are well-formed, i.e., $\mathcal{W}f = \mathsf{CMem}$. The empty memory $\mu_\emptyset$ is the empty function and the composition of two memories μ and μ' is their disjoint union $\mu \uplus \mu'$ (i.e., their union defined only for nonoverlapping memories).

Instance Property 1. The components $(\mathsf{CMem}, \mu_\emptyset, \cdot)$ form a PCM.

We now discuss our big-step operational semantics for the language, with judgement

$$\sigma, \mathsf{C} \Downarrow_\gamma o : \sigma'$$

reading "the execution of command C with function implementation context γ in state σ results in a state σ' with outcome o". A program state is a pair $\sigma = (s, \mu)$ comprising a variable store $s : \mathsf{PVar} \rightharpoonup_{fin} \mathsf{Val}$ and a memory $\mu \in \mathsf{CMem}$. Outcomes are defined as $o ::= ok \mid err \mid miss$, denoting, respectively, a successful execution, a fault due to a language error, and a fault due to a missing resource error. We must distinguish between the two kinds of faults as the missing-resource errors have a different role to play in compositional reasoning (see IProp. 2 below) and bi-abduction (see §5.6). Function implementation contexts γ provide the function definitions used in function calls.

The only interesting case in the definition of the semantics is the case for memory actions, which is given by instance data. We only discuss this case; for other cases, see the extended paper [31].

Instance Definition 2. Memory actions are defined by a relation, written $\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v'})$, which executes an action α on memory μ with parameters $\vec{v}$, and returns an outcome o, memory μ', and return values $\vec{v'}$. All memory actions must preserve well-formedness ($\mathcal{W}f$).

Instance Definition Example 2. Our linear memory model has four memory actions: `lookup`, `mutate`, `new`, and `free`. We give the *ok* and *miss* rules for defining the `lookup` action; the full set of rules is in the extended paper [31]:

$$\frac{\mu(n) = v}{\mu.\mathtt{lookup}([n]) \rightsquigarrow ok : (\mu, [v])} \qquad \frac{n \notin \mathrm{dom}(\mu)}{\mu.\mathtt{lookup}([n]) \rightsquigarrow miss : (\mu, [\text{“MissingCell”}, n])}$$

[3] We use the $X \rightharpoonup Y$ to denote partial functions from X to Y and $X \rightharpoonup_{fin} Y$ to denote partial functions with finite support.
[4] It would also be possible to incorporate the $\mathcal{W}f$ predicate into the concrete memory type itself using subtyping or dependent types. We chose to keep it separate so that our meta-theory is simply typed. This is a presentational choice; the condition is the same with both choices.

The following two rules lift the semantics of memory actions to the command level:

$$\frac{[\![\vec{\mathsf{E}}]\!]_s = \vec{v} \quad \mu.\alpha(\vec{v}) \rightsquigarrow ok : (\mu', \vec{v}') \quad |\vec{v}'| = |\vec{\mathsf{x}}|}{(s,\mu), \vec{\mathsf{x}} := \alpha(\vec{\mathsf{E}}) \Downarrow_\gamma ok : (s[\vec{\mathsf{x}} \mapsto \vec{v}'], \mu')} \qquad \frac{[\![\vec{\mathsf{E}}]\!]_s = \vec{v} \quad \mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}) \quad o \neq ok}{(s,\mu), \vec{\mathsf{x}} := \alpha(\vec{\mathsf{E}}) \Downarrow_\gamma o : (s[\mathsf{err} \mapsto \vec{v}], \mu')}$$

where $[\![\mathsf{E}]\!]_s$ denotes the standard evaluation of an expression E with respect to a store s, resulting either in a value or a dedicated symbol $\lightning \notin \mathsf{Val}$ denoting an evaluation error.

OX and UX frame. As is standard in compositional reasoning based on SL and ISL, we rely on the fact that the concrete semantics of the language satisfies frame properties. To ensure that we have these properties, we require that the concrete semantics of memory actions satisfy the standard OX and/or UX frame properties (which in turn straightforwardly lift to the full concrete semantics):

Instance Property 2. For $\mathcal{W}f(\mu)$ and $\mathcal{W}f(\mu_f)$:

(OX) If $(\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$
then $\exists \mu'', \vec{v}'', o'.\ \mu.\alpha(\vec{v}) \rightsquigarrow o' : (\mu'', \vec{v}'')$ and
$(o' \neq \mathit{miss} \Rightarrow (o' = o \text{ and } \vec{v}'' = \vec{v}' \text{ and } \mu' = \mu'' \cdot \mu_f))$

(UX) If $\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$ and $o \neq \mathit{miss}$ and $\mu' \cdot \mu_f$ is defined
then $(\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow o : (\mu' \cdot \mu_f, \vec{v}')$

The OX property is more subtle than the UX property since to capture that we can extend analysis results to larger states we must say, perhaps counterintuitively, that *removing* a "frame" μ_f from $\mu \cdot \mu_f$ results in either a *miss* outcome or the same behaviour as executing from the full state, rather than more straightforwardly stating something about *adding* more state; see Yang and O'Hearn [55] for an in-depth discussion of the OX property. Note that while both properties capture that the analysis results can be extended to larger states, the frame μ_f is added to the initial state μ for OX reasoning (as, recall the definition in §2, OX soundness universally quantifies over all initial states) and to the final state μ' for UX reasoning (as UX soundness instead universally quantifies over all final states).

4 Assertions and Function Specifications

We introduce our assertion language and its satisfaction relation, parametric on a *resource model* comprising the resource assertions and satisfaction relation described in IDef. 3. The assertions provide the pre- and postconditions of SL and ISL function specifications and are also used in assertion-based constructs of our CSE engine such as folding and unfolding of user-defined predicates.

Assertion syntax. We define assertions, Asrt, assuming a set of logical variables, $x, y, z, \in \mathsf{LVar}$, distinct from program variables, and a set of logical expressions, $E \in \mathsf{LExp}$, which extends program expressions PExp to include these logical variables and two new expressions $E \in \mathsf{Val}$ and $E \in \tau$ where $\tau ::= \mathsf{Null} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \ldots$, meaning, that the expression E successfully evaluates to a value and successfully evaluates to a value of type τ, respectively. The syntax of assertions is defined by:

$$A \in \mathsf{Asrt} \stackrel{\text{def}}{=} E \mid \mathsf{True} \mid A_1 \Rightarrow A_2 \mid A_1 \vee A_2 \mid \exists x.\, A \mid \mathsf{emp} \mid A_1 \star A_2 \mid r(\vec{E_1}; \vec{E_2}) \mid p(\vec{E_1}; \vec{E_2})$$

for $x \in \mathsf{LVar}$, $E \in \mathsf{LExp}$, $\vec{E_1}, \vec{E_2} \in \vec{\mathsf{LExp}}$, $r \in \mathsf{Str}$, and $p \in \mathsf{Str}$. Our assertions comprise Boolean assertions E, several first-order connectives and quantifiers, the empty-memory assertion emp, assertions built using the separating conjunction $\star$, resource assertions $r(\vec{E_1}; \vec{E_2})$, and user-defined predicate assertions $p(\vec{E_1}; \vec{E_2})$. The parameters of resource and user-defined predicate assertions are split into *in-parameters* and *out-parameters* for automation purposes: in our CSE engine, the consume operation requires the in-parameters *to be known* before consumption and *learns* the out-parameters during consumption; see Lööw et al. [33] for further details.

Satisfaction relation. To define the satisfaction relation for assertions, we introduce logical interpretations, $\theta : \mathsf{LVar} \rightharpoonup_{fin} \mathsf{Val}$, and the evaluation of logical expressions, $[\![E]\!]_{\theta,s}$, which extends program expression evaluation to interpret logical variables using θ. The satisfaction relation, $\theta, \sigma \models A$, is defined in the extended paper [31]; the interesting cases are:

$$\theta, (s, \mu) \models A_1 \star A_2 \;\Leftrightarrow\; \exists \mu_1, \mu_2.\, \mu = \mu_1 \cdot \mu_2 \text{ and } \theta, (s, \mu_1) \models A_1 \text{ and } \theta, (s, \mu_2) \models A_2$$
$$\theta, (s, \mu) \models r(\vec{E_1}; \vec{E_2}) \;\Leftrightarrow\; [\![\vec{E_1}]\!]_{\theta,s} = \vec{v_1} \text{ and } [\![\vec{E_2}]\!]_{\theta,s} = \vec{v_2} \text{ and } \mu \models_{\mathsf{Res}} r(\vec{v_1}; \vec{v_2}) \text{ defined in IDef. 3.}$$

As is standard for parametric separation logics, the semantics of $\star$ is defined with respect to the composition operator $\cdot$ from the memory PCM instance data (IDef. 1). The semantics of resource assertions $r(\vec{E_1}; \vec{E_2})$ is also defined by instance data:

Instance Definition 3. A set of resource assertions of the form $r(\vec{v_1}; \vec{v_2})$, and a satisfaction relation for the resource assertions, $\mu \models_{\mathsf{Res}} r(\vec{v_1}; \vec{v_2})$.

Instance Definition Example 3. In our linear memory model, there are two types of resources: the positive cell assertion, $E_1 \mapsto E_2$ (in prettified syntax) with in-parameter E_1 and out-parameter E_2, and the negative cell assertion, $E \mapsto \varnothing$ with in-parameter E. Their satisfaction relation is as follows:

$$\mu \models_{\mathsf{Res}} n \mapsto v \;\Leftrightarrow\; \mu = \{n \mapsto v\}$$
$$\mu \models_{\mathsf{Res}} n \mapsto \varnothing \Leftrightarrow\; \mu = \{n \mapsto \varnothing\}$$

Program logics and function specifications. On top of our assertion language, we define the standard SL and ISL function triples (and quadruples, to describe both successful and erroneous outcomes). Our theory additionally supports exact separation logic triples [34], which are triples that are valid both in the SL and ISL senses (i.e., they can be used for both types of reasoning). As the SL, ISL, and ESL definitions are standard, we only give the formal definitions in the extended paper [31]. It is important that the definitions are indeed standard, otherwise CSE implementations based on our theory would not be formally interoperable with other analysis implementations based on SL and ISL (i.e., one could not formally exchange specifications between the implementations).

We additionally need the following definitions to eventually state our engine soundness theorems. A function specification context, Γ, is a function from function identifiers to sets of function specifications. We say a function specification context Γ is valid w.r.t. a function implementation context γ, denoted $\models (\gamma, \Gamma)$, when all function specifications in Γ are valid w.r.t. γ. Formal definitions are, again, in the extended paper [31].

5 CSE Engine

We introduce our CSE engine and prove its OX and UX soundness theorems. Our discussion is focused on the parameters of the relevant theory that we say form a *symbolic memory model* and our *soundness relations*. The relevant parameters are: the symbolic memory data type (IDef. 4), its satisfaction relation (IDef. 5), symbolic memory actions (IDef. 6), and consume and produce operations (IDef. 7) and their associated IProps. A larger excerpt of the formal rules of the engine is given in the extended paper [31]; and the full set of rules is available in our Rocq mechanisation.

5.1 Symbolic States

The symbolic states of our engine, denoted $\hat{\sigma}$, are built out of logical expressions LExp with the extra condition that none of the logical expressions have program variables. We use hat-notation to distinguish symbolic definitions, such as $\hat{\sigma}$ for symbolic state compared with σ for concrete state.

The most interesting component of symbolic states is their symbolic memory component, which is given by instance data:

Instance Definition 4. A pair $(\mathsf{SMem}, \hat{\mu}_\emptyset)$, where SMem is a symbolic memory and $\hat{\mu}_\emptyset \in \mathsf{SMem}$ is the empty memory.

Instance Definition Example 4. Our linear memory model comprises SMem equalling $\mathsf{LExp} \rightharpoonup_{\mathtt{fin}} (\mathsf{LExp} \uplus \{\emptyset\})$ and $\hat{\mu}_\emptyset$ equalling the empty function.

We are now ready to give the full definition of symbolic state: a symbolic state $\hat{\sigma}$ is a tuple $(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi})$ where: $\hat{s} : \mathsf{PVar} \rightharpoonup_{\mathtt{fin}} \mathsf{LExp}$ is a symbolic store; $\hat{\mu} \in \mathsf{SMem}$ is a symbolic memory from IDef. 4; $\hat{\mathcal{P}}$ is a multiset of symbolic user-defined predicates, where a symbolic predicate has the form $p(\vec{\hat{E}}_1; \vec{\hat{E}}_2)$ where $p \in \mathsf{Str}$ is a predicate name and $\vec{\hat{E}}_1 \in \vec{\mathsf{LExp}}$ are the in-parameters and $\vec{\hat{E}}_2 \in \vec{\mathsf{LExp}}$ the out-parameters; and $\hat{\pi} \in \mathsf{LExp}$ is a path condition that captures constraints imposed during execution. We use the syntax $\hat{\sigma}.\mathsf{mem}$, $\hat{\sigma}.\mathsf{pc}$ etc. to access components of symbolic state and the syntax $\hat{\sigma}[\mathsf{mem} := \hat{\mu}']$ etc. to update components of symbolic state. We use $\mathtt{lv}(\hat{\sigma})$ to refer to the logical variables of a symbolic state $\hat{\sigma}$.

We define the semantic meaning of symbolic states using a satisfaction relation between concrete and symbolic states of the form $\theta, (s, \mu) \models \hat{\sigma}$ where $\theta : \mathsf{LVar} \to \mathsf{Val}$ is a logical interpretation, $s : \mathsf{PVar} \to \mathsf{Val}$ is a variable store and $\mu \in \mathsf{CMem}$ a concrete memory. The satisfaction relation for symbolic states depends on the satisfaction relations for symbolic stores $\theta, s \models_{\mathrm{Sto}} \hat{s}$, symbolic memories $\theta, \mu \models_{\mathrm{Mem}} \hat{\mu}$, and symbolic predicates $\theta, \mu \models_{\mathrm{Pred}} \hat{\mathcal{P}}$: the satisfaction relations for symbolic stores and symbolic predicates are as expected, see the extended paper [31]; the satisfaction relation for symbolic memories is given by instance data:

Instance Definition 5. A satisfaction relation for symbolic memory of the form $\theta, \mu \models_{\mathrm{Mem}} \hat{\mu}$, with the property that $\theta, \mu \models_{\mathrm{Mem}} \hat{\mu}_\emptyset \Leftrightarrow \mu = \mu_\emptyset$.

Instance Definition Example 5. In our linear memory model, the satisfaction relation for symbolic memory is defined as follows, where for succinct presentation we say $[\![\emptyset]\!]_\theta = \emptyset$:

$$\theta, \mu \models_{\mathrm{Mem}} \{\hat{E}_a^1 \mapsto \hat{E}_e^1, \ldots, \hat{E}_a^n \mapsto \hat{E}_e^n\} \Leftrightarrow \mu = \uplus_{i=1}^n \{[\![\hat{E}_a^i]\!]_\theta \mapsto [\![\hat{E}_e^i]\!]_\theta\}$$

The satisfaction relation $\theta, (s, \mu) \models (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi})$ for symbolic state is defined by:

$$\exists \mu_1, \mu_2.\ \mu = \mu_1 \cdot \mu_2 \text{ and } \theta, s \models_{\mathrm{Sto}} \hat{s} \text{ and } \theta, \mu_1 \models_{\mathrm{Mem}} \hat{\mu} \text{ and } \theta, \mu_2 \models_{\mathrm{Pred}} \hat{\mathcal{P}} \text{ and } [\![\hat{\pi}]\!]_\theta = \mathsf{true}$$

We say that a symbolic state $\hat{\sigma}$ is satisfiable, denoted $\mathtt{SAT}(\hat{\sigma})$, when $\exists \theta, s, \mu.\ \theta, (s, \mu) \models \hat{\sigma}$, and say that it implies an expression, denoted $\hat{\sigma} \models \hat{E}$, when $\forall \theta, s, \mu.\ \theta, (s, \mu) \models \hat{\sigma} \Rightarrow [\![\hat{E}]\!]_\theta = \mathsf{true}$.

5.2 Engine Judgement

The symbolic semantics of our CSE engine is given by a judgement of the form:

$$O, \hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^m o : O', \hat{\sigma}'$$

where: oracles O, O' of type $\mathsf{Nat} \to \mathsf{Nat}$ are used to model angelic nondeterminism,[5] e.g. when there are multiple applicable function specifications to choose from for a function call; the mode m, either OX, UX, or EX (for exact), enables the engine to switch its behaviour depending on what type of soundness we need, e.g. what kind of function specifications to use in function calls, and the set of *outcomes*, $o ::= ok \mid err \mid miss \mid abort$, extends the outcomes of the concrete language with *abort*, e.g. when a chosen function specification is not applicable.

[5] See, e.g., Owens et al. [41] for further discussion on oracles. In short, each number $O(0), O(1), \ldots$ represents an answer to a choice and the oracle is shifted once every time a choice is made (such that the oracle can always be queried by looking at $O(0)$), ultimately resulting in an output oracle $O' = \lambda n.\ O(n + m)$ where m is the number of choices made. The oracle semantics we use is an intentionally simplistic model to avoid cluttering our formalism. In particular, the same angelic choice is taken in each demonic branch. See Dardinier et al. [13] (using multi-relations) or Jacobs et al. [22] (using monads) for more expressive formalisms for nondeterminism.

5.3 Memory-Action Command

The symbolic semantics of memory-action commands, analogous to the concrete semantics, is parametric on an instance-given symbolic action execution relation.

Instance Definition 6. A relation $\hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}')$, which executes an action α on memory $\hat{\mu}$ with arguments $\vec{\hat{E}}$, and returns an outcome o, memory $\hat{\mu}'$, path condition $\hat{\pi}'$, and return values $\vec{\hat{E}}'$. If $o \in \{\mathit{miss}, \mathit{abort}\}$, then the symbolic memory must remain unchanged, i.e., $\hat{\mu} = \hat{\mu}'$.

Instance Definition Example 6. To illustrate, we give some of the symbolic rules for the `lookup` action of our linear memory model (for the rest, see the extended paper [31]), specifically, the success and missing resource rules:

$$\frac{\hat{\mu}(\hat{E}_l') = \hat{E} \qquad \hat{\pi}' = (\hat{E}_l = \hat{E}_l')}{\hat{\mu}.\texttt{lookup}([\hat{E}_l]) \rightsquigarrow \mathit{ok} : (\hat{\mu}, \hat{\pi}', [\hat{E}])} \qquad \frac{\hat{\pi}' = (\hat{E}_l \in \mathsf{Nat} \wedge \hat{E}_l \notin \mathsf{dom}(\hat{\mu}))}{\hat{\mu}.\texttt{lookup}([\hat{E}_l]) \rightsquigarrow \mathit{miss} : (\hat{\mu}, \hat{\pi}', [\text{“MissingCell”}, \hat{E}_l])}$$

where, note, the first rule branches over all possible addresses $\hat{E}_l'$ where $\hat{E}_l = \hat{E}_l'$ holds. We discuss other “branching strategies” in §6.1, where we discuss variants of the memory model.

The following symbolic rules lift, again analogously to the concrete semantics, the memory actions to the full semantics, where $[\![\mathsf{E}]\!]_{\hat{s}}$ denotes symbolic expression evaluation, which evaluates a program expression E w.r.t. a symbolic store $\hat{s}$:

$$\frac{[\![\vec{\mathsf{E}}]\!]_{\hat{s}} = \vec{\hat{E}} \qquad \hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow \mathit{ok} : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}') \qquad |\vec{\hat{E}}'| = |\vec{\mathsf{x}}| \qquad \hat{s}' = \hat{s}[\vec{\mathsf{x}} \mapsto \vec{\hat{E}}'] \qquad \hat{\pi}'' = (\hat{s}(\vec{\mathsf{x}}), \vec{\hat{E}}, \vec{\hat{E}}' \in \mathsf{Val} \wedge \hat{\pi}' \wedge \hat{\pi})}{O, (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \vec{\mathsf{x}} := \alpha(\vec{\mathsf{E}}) \Downarrow_\Gamma^m \mathit{ok} : O, (\hat{s}', \hat{\mu}', \hat{\mathcal{P}}, \hat{\pi}'')}$$

$$\frac{[\![\vec{\mathsf{E}}]\!]_{\hat{s}} = \vec{\hat{E}} \qquad \hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}') \qquad o \neq \mathit{ok} \qquad \hat{s}' = \hat{s}[\mathsf{err} \mapsto \vec{\hat{E}}'] \qquad \hat{\pi}'' = (\vec{\hat{E}} \in \mathsf{Val} \wedge \hat{\pi}' \wedge \hat{\pi})}{O, (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \vec{\mathsf{x}} := \alpha(\vec{\mathsf{E}}) \Downarrow_\Gamma^m \mathit{err} : O, (\hat{s}', \hat{\mu}', \hat{\mathcal{P}}, \hat{\pi}'')}$$

We require the following property to be able to prove that our CSE engine, specifically, the memory-action commands, satisfy our two soundness theorems:

Instance Property 3. The symbolic memory-action semantics must be OX/UX sound w.r.t. the concrete memory-action semantics, i.e., the two semantics must satisfy the OX/UX soundness definitions introduced in §2 but with the command-level concrete semantics replaced by the memory-action-level concrete semantics, the satisfaction relation $\models$ replaced by the satisfaction relation $\models_{\mathsf{Mem}}$, etc. The full formal properties are given in the extended paper [31].

5.4 Consume and Produce Operations and Assertion-Based Commands

We now discuss the definition, soundness, and usage of our memory-model-parametric consume and produce operations that form the basis of our CSE engine's consume-produce architecture.

Definition. The judgements of consume and produce are as follows. First, we introduce symbolic substitutions, $\hat{\theta} : \mathsf{LVar} \rightharpoonup_{\mathtt{fin}} \mathsf{LExp}$, which the two operations use to instantiate free logical variables. Now, the judgements of consume and produce are:

$$\mathsf{consume}(m, O, A, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, O', \hat{\theta}', \hat{\sigma}') \qquad \text{and} \qquad \mathsf{produce}(A, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$$

where the judgement for consume states that the consumption of assertion A in mode m (which decides how Boolean information is consumed) with an oracle O and an initial symbolic substitution $\hat{\theta}$ from state $\hat{\sigma}$ results in outcome o (ok or abort), an updated oracle O', symbolic state $\hat{\sigma}'$ where the symbolic state corresponding to A has been removed, and extended symbolic substitution $\hat{\theta}'$ now containing mappings for all free logical variables of A; and the judgement for produce states that the production of A with symbolic substitution $\hat{\theta}$ in state $\hat{\sigma}$ results in state $\hat{\sigma}'$ where the symbolic state corresponding to A has been added.

We only discuss how resource assertions $r(\vec{E}_1; \vec{E}_2)$ are consumed and produced (remaining rules are inspired by the rules of Lööw et al. [32],[6] which in turn are inspired by Gillian). The consume and produce operations are parametric on two resource-level consume and produce operations, which we call, respectively, *resource consume* $\mathsf{consume}_{\mathsf{Res}}$ and *resource produce* $\mathsf{produce}_{\mathsf{Res}}$:

Instance Definition 7. Two operations $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ of the form explained below:

$$\mathsf{consume}_{\mathsf{Res}}(m, O, r, \vec{\hat{E}}_{\mathrm{in}}, \hat{\mu}) \rightsquigarrow (o, O', \vec{\hat{E}}_{\mathrm{out}}, (\hat{\mu}', \hat{\pi}_i, \hat{\pi})) \quad \text{and} \quad \mathsf{produce}_{\mathsf{Res}}(r, \vec{\hat{E}}_{\mathrm{in}}, \vec{\hat{E}}_{\mathrm{out}}, \hat{\mu}) \rightsquigarrow (\hat{\mu}', \hat{\pi})$$

Instance Definition Example 7. For our linear memory model, the following are two of the rules for $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ (again, the rest of the rules are in the extended paper [31]):

$$\frac{\hat{\mu} = \hat{\mu}_f \uplus \{\hat{E}_1 \mapsto \hat{E}_2\}}{\mathsf{consume}_{\mathsf{Res}}(m, O, \mapsto, [\hat{E}], \hat{\mu}) \rightsquigarrow (ok, O, \hat{E}_2, (\hat{\mu}_f, \mathsf{true}, \hat{E} = \hat{E}_1))} \qquad \frac{\hat{\mu}' = \hat{\mu} \uplus \{\hat{E}_1 \mapsto \hat{E}_2\}}{\mathsf{produce}_{\mathsf{Res}}(\mapsto, [\hat{E}_1], [\hat{E}_2], \hat{\mu}) \rightsquigarrow (\hat{\mu}', \mathsf{true})}$$

The $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations, analogous to memory actions, are lifted into consume and produce. We explain the consume case, the produce case is similar. See the consume rule in Fig. 2, which illustrates the most interesting parts of how $\mathsf{consume}_{\mathsf{Res}}$ is lifted into consume. There are two conditions that $\mathsf{consume}_{\mathsf{Res}}$ outputs: $\hat{\pi}_i$, which must be implied by the initial state, and $\hat{\pi}$, which is appended to the path condition of the updated state. The two conditions are used to implement different types of branching, which we illustrate by example when discussing variants of the linear memory model in §6.1.

$$\frac{\begin{array}{c}\mathsf{consume}_{\mathsf{Res}}(m, O, r, \hat{\theta}(\vec{E}_{\mathrm{in}}), \hat{\sigma}.\mathsf{mem}) \rightsquigarrow (o, O', \vec{\hat{E}}_{\mathrm{out}}, (\hat{\mu}', \hat{\pi}_i, \hat{\pi})) \\ \hat{\sigma} \models \hat{\pi}_i \qquad \hat{\sigma}' = \hat{\sigma}[\mathsf{mem} := \hat{\mu}', \mathsf{pc} := \hat{\pi} \wedge \hat{\sigma}.\mathsf{pc}] \\ \text{rest of the rule omitted}\end{array}}{\mathsf{consume}(m, O, r(\vec{E}_{\mathrm{in}}; \vec{E}_{\mathrm{out}}), \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, O', \hat{\theta}', \hat{\sigma}'')}$$

Fig. 2. Implementation of consume.

Soundness. We introduce OX and UX soundness properties that formalise that the consume and produce operations "correctly" consume and produce their input assertions. Our soundness properties are based on the soundness properties for consume and produce introduced by Lööw et al. [32], which we have refactored into four properties: (1) consume OX soundness, (2) produce OX soundness, (3) consume UX soundness, (4) produce UX soundness. The full properties are available in the extended paper [31]; in short, the properties relate the behaviour of consume and produce to the satisfaction relation of the assertion language. For example, UX soundness of consume requires that the composition of the models of the input assertion and the output symbolic state forms a model of the input symbolic state:

$$\begin{array}{ll}\text{If} & \mathsf{consume}(m, O, A, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (ok, O', \hat{\theta}', \hat{\sigma}_f) \\ & \text{and } \theta, (s, \mu_A) \models \hat{\theta}'(A) \text{ and } \theta, (s, \mu_f) \models \hat{\sigma}_f \text{ and } (\mu_A \cdot \mu_f) \text{ is defined} \\ \text{then} & \theta, (s, \mu_A \cdot \mu_f) \models \hat{\sigma}\end{array}$$

To ensure that our soundness properties of consume and produce hold, we require that resource-only variants of the properties to hold for the $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations.

Instance Property 4. The $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations must be OX/UX sound. Again, the full properties are stated in the extended paper [31]. To exemplify the resource-only variant of

[6] Our operations support the subset of assertions usually supported by consume and produce operations. That is, both operations support Boolean assertions, existential quantification (for consume only in OX mode, we do not know of a use-case in UX mode), the empty-memory assertion, separating-conjunction assertions, resource assertions, and user-defined-predicate assertions, and, additionally, produce supports disjunction assertions.

the properties, we give the resource-only variant of the UX soundness property for consume (the reader should compare the property with the property above):

$$\begin{array}{ll}\text{If} & \mathsf{consume}_{\mathsf{Res}}(m, O, r, \vec{\hat{E}}_{\text{in}}, \hat{\mu}) \rightsquigarrow (ok, O', \vec{\hat{E}}_{\text{out}}, (\hat{\mu}_f, \hat{\pi}_{fi}, \hat{\pi}_f)) \text{ and } [\![\vec{\hat{E}}_{\text{in}}]\!]_\theta = \vec{v}_{\text{in}} \text{ and } [\![\vec{\hat{E}}_{\text{out}}]\!]_\theta = \vec{v}_{\text{out}} \\ & \text{and } [\![\hat{\pi}_f]\!]_\theta = \mathsf{true} \text{ and } \theta, \mu_f \models_{\mathsf{Mem}} \hat{\mu}_f \text{ and } \mu_r \models_{\mathsf{Res}} r(\vec{v}_{\text{in}}; \vec{v}_{\text{out}}) \text{ and } (\mu_r \cdot \mu_f) \text{ is defined} \\ \text{then} & [\![\hat{\pi}_{fi}]\!]_\theta = \mathsf{true} \text{ and } \theta, (\mu_r \cdot \mu_f) \models_{\mathsf{Mem}} \hat{\mu}\end{array}$$

With the above definitions in place, we have been able to prove the following:

Lemma 5.1. *Given OX/UX sound* $\mathsf{consume}_{\mathsf{Res}}$ *and* $\mathsf{produce}_{\mathsf{Res}}$ *operations, our consume and produce operations are OX/UX sound.*

Usage. We have mechanised and proved sound the standard consume-produce definitions of the function-call command and fold/unfold commands for user-defined predicates, the full definitions are available in the extended paper [31] for completeness. The function-call command we have proved OX and UX sound, whereas the fold/unfold commands we have proved OX sound.[7] The successful proofs of these commands show that our consume-produce properties are sufficient for their core use cases. Additionally, as we discuss in §5.6, the analyses we have built on top of our engine show that our consume-produce properties are also sufficient for those analyses.

5.5 Engine Soundness

Our OX soundness and UX soundness theorems are as follows:[8]

Theorem 5.2 (OX soundness). *Let* $m \in \{OX, EX\}$ *and assume* $\models (\gamma, \Gamma)$*, when the CSE engine is instantiated with OX sound memory actions,* $\mathsf{consume}_{\mathsf{Res}}$*, and* $\mathsf{produce}_{\mathsf{Res}}$*, then the following holds:*

$$\begin{array}{ll}\textit{If} & \sigma, \mathsf{C} \Downarrow_\gamma o : \sigma' \textit{ and } \theta, \sigma \models \hat{\sigma} \textit{ and} \\ & (\forall o, O', \hat{\sigma}'.\ O, \hat{\sigma}, \mathsf{C} \Downarrow^m_\Gamma o : O', \hat{\sigma}' \textit{ and } \mathsf{SAT}(\hat{\sigma}') \Rightarrow \\ & \quad o \neq \textit{abort and } (o = \textit{miss} \Rightarrow \hat{\sigma}'.\mathsf{preds} = \emptyset)) \\ \textit{then} & \exists O', \hat{\sigma}', \theta'.\ O, \hat{\sigma}, \mathsf{C} \Downarrow^m_\Gamma o : O', \hat{\sigma}' \textit{ and } \theta'|_{\mathsf{lv}(\hat{\sigma})} = \theta \textit{ and } \theta', \sigma' \models \hat{\sigma}'\end{array}$$

Theorem 5.3 (UX soundness). *Let* $m \in \{UX, EX\}$ *and assume* $\models (\gamma, \Gamma)$*, when the CSE engine is instantiated with UX sound memory actions,* $\mathsf{consume}_{\mathsf{Res}}$*, and* $\mathsf{produce}_{\mathsf{Res}}$*, then the following holds:*

$$\begin{array}{ll}\textit{If} & O, \hat{\sigma}, \mathsf{C} \Downarrow^m_\Gamma o : O', \hat{\sigma}' \textit{ and } o \neq \textit{abort and } (o = \textit{miss} \Rightarrow \hat{\sigma}'.\mathsf{preds} = \emptyset) \textit{ and } \theta, \sigma' \models \hat{\sigma}' \\ \textit{then} & \exists \sigma.\ \sigma, \mathsf{C} \Downarrow_\gamma o : \sigma' \textit{ and } \theta, \sigma \models \hat{\sigma}\end{array}$$

Both theorems have restrictions on *abort* and *miss* outcomes. For the OX theorem, the condition should be read as follows: no reachable satisfiable state has an outcome *abort* or an outcome *miss* unless there are no symbolic predicates in the state. For both theorems, the soundness of *miss* outcomes cannot be guaranteed in the presence of symbolic predicates because the source of *miss* outcomes, memory actions (IDef. 6), do not take symbolic predicates into consideration (doing so would require implementing an automated complete unfolding procedure, which none of the existing CSE tools or platforms implement). To exemplify, say we are working with our running example linear memory model and have defined the following user-defined predicate: $\mathsf{foo}(;)\{1 \mapsto 1\}$. First, let $\mathsf{C} \stackrel{\text{def}}{=} \mathsf{x} := \mathtt{lookup}(1)$, $\hat{\sigma} \stackrel{\text{def}}{=} (\emptyset, \hat{\mu}_\emptyset, \{\mathsf{foo}(;)\}, \mathsf{true})$, and $\sigma \stackrel{\text{def}}{=} (\emptyset, \{1 \mapsto 1\})$. Now, note that $\emptyset, \sigma \models \hat{\sigma}$, concrete execution of C from σ only results in an *ok* outcome, and symbolic execution of C from $\hat{\sigma}$ only results in a *miss* outcome. This breaks both OX and UX soundness: there is no corresponding execution for the concrete execution and vice versa.

[7] Lööw et al. [32] claim that the fold command is UX sound if folding is restricted to strictly exact predicates (an assertion A is strictly exact iff $\theta, (s, \mu) \models A \wedge \theta, (s, \mu') \models A \Rightarrow \mu' = \mu$ [54, p. 149]), but during our mechanisation work we found a counterexample to this claim. We have not investigated a new condition to make the command UX sound.

[8] Here, we have simplified away some uninteresting details of the statements, see the Rocq mechanisation for the full details.

5.6 Analyses

We now discuss two analyses we have built on top of our CSE engine and proved sound: a *function specification verification analysis* to exemplify an OX analysis application and a *true bug-finding analysis based on bi-abduction* to exemplify an UX analysis application. We additionally discuss the trusted computing base (TCB) of analysis results when using our engine.

The two analyses. The development and verification of our OX analysis application was relatively uneventful: our work validates that our adapted consume-produce properties are sufficient for this OX application, but the analysis itself is standard in the consume-produce literature, and we did not run into any particular problems with porting its proof to our parametric setting. We therefore only discuss our UX application here; see the extended paper [31] for our OX application.

Bi-abduction is a technique that facilitates automatic ISL specification synthesis by incrementally discovering the resources needed to execute a given piece of code starting from an empty pre-condition/symbolic state. It was first introduced in the OX setting [9, 10], forming the basis of the Infer tool [8]. It was later ported to the OX consume-produce setting in the JaVerT 2.0 project [18], by re-imagining bi-abduction as *fixes-from-missing-resource-errors*. With the introduction of incorrectness separation logic [46], the original bi-abduction algorithm was ported to the UX setting of true bug-finding, underpinning the Infer-Pulse tool [26]. Following this, Lööw et al. [32] showed that the fixes-from-missing-resource-errors approach is UX sound in the setting of linear memory. Here, we generalise Lööw et al. [32]'s UX result to our memory-model-parametric setting.

We have built a bi-abductive engine with judgement $O, \hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{bi}} o : O', (\hat{\sigma}', A)$ on top of our engine with judgement $O, \hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{UX}} o : O', \hat{\sigma}'$. The A in the judgement is an assertion representing an *anti-heap*, which captures the missing resources needed to execute the command C in the following sense:

Theorem 5.4 (CSE with Bi-Abduction: Soundness).

$$\begin{array}{ll} \textit{If} & \models (\gamma, \Gamma) \textit{ and } O, \hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{bi} o : O', (\hat{\sigma}', A) \textit{ and } \theta, \sigma' \models \hat{\sigma}' \\ \textit{then} & \exists s, \mu, \mu_{fix}.\ \theta, (s, \mu) \models \hat{\sigma}[\mathrm{pc} := \hat{\sigma}'.\mathrm{pc}] \textit{ and } \theta, (s, \mu_{fix}) \models A \textit{ and } (s, \mu \cdot \mu_{fix}), \mathsf{C} \Downarrow_\gamma o : \sigma' \end{array}$$

In short, the bi-abductive engine works by catching missing-resource errors and abort errors during execution, which are given to a memory-model-dependent operation fix that takes, as input, the current symbolic state and constructs one or more assertions representing the resources needed for continued execution, which in turn are produced into the current symbolic state and appended to the anti-heap. The soundness of the engine (Thm. 5.4) follows from the UX soundness of our CSE engine (Thm. 5.3) and produce (Lem. 5.1). To prove soundness, we had to adapt the soundness statement and proofs from previous work [18, 32], which relied on having a symbolic composition operator and a symbolic frame property, which we do not require as parameters.

Trusted computing base. As one would expect from a mechanised theory, our CSE theory comes with a strong TCB story: the TCB of analysis results includes only the semantics of assertions (to express pre- and postconditions) and the concrete semantics of the language. The TCB can be further reduced by considering only first-order assertions, as then the assertion language can be removed from the TCB (this is analogous to sound program logics, see, e.g., Iris' adequacy theorem [24]).

The fact that the concrete memory model is part of the TCB (as it is part of the concrete semantics), means that one must choose the model carefully. Appropriate TCB models have a direct correspondence to a memory model specified by a language standard or the like. Because of space constraints, we do not discuss this point further but show in our Rocq development how analysis artefacts like ghost state annotations (e.g., like the annotations used by our fractional ownership model introduced in §6.2) can easily be removed from the TCB by a standard simulation argument.

6 Memory-Model Instances

Now having introduced the IDefs. and IProps. required to instantiate our CSE theory, we discuss additional examples of memory-model instances that fit into our CSE theory, as summarised in Tab. 1.

We emphasise that the primary purpose of our discussion in this section is to show that a wide array of memory models fit into our CSE theory. We do not have sufficient page budget to formally introduce all required IDefs. (i.e, IDefs. 1–7) and discuss proofs of the required IProps. (i.e., IProp. 1–4) for each memory instance. Therefore, our discussion is informal and focused on what we have found to be the primary difficulty to get right in designing memory models: the memory data type (IDef. 1), such that compositional memory actions (IDef. 2) that work over SL/ISL-style partial state can be defined etc. (We give additional definitions in the extended paper [31] and all definitions are available in our Rocq development.) With the right data type in place, we have found other major instance data, such as symbolic memory actions (IDef. 6) and $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations (IDef. 7), to be relatively straightforward to define. In particular, for simpler memory models, symbolic components can be designed by "symbolically lifting" of the corresponding concrete component, in particular, the symbolic data type and memory actions. For an example, compare the concrete data type (IDef. 1) and symbolic data type (IDef. 4) of our running example memory model and note how the symbolic data type is structurally identical to the concrete data type but abstracts both Nat and Val to LExp.

6.1 Linear Memory Models and Memory-Model Design Considerations

We have mechanised and proved sound multiple variants of our running example linear memory model (see again Tab. 1). As discussed in the introduction (§1), there are multiple degrees of freedom available when designing a symbolic memory model. Simple linear memory models provide a good stage to illustrate this; as we have already introduced the various IDefs. of our running example linear memory model, we in this section discuss design considerations relating to OX vs. UX analysis, such as what types of "branching strategies" are allowed by different reasoning modes.

Operational meaning of OX vs. UX. The different requirements arising from OX and UX soundness can be exploited in memory-model design. Intuitively, OX analyses like verification must consider all execution paths whereas UX analyses like bug-finding only need to consider paths with bugs. More precisely: operationally, OX soundness allows for dropping information along execution paths but not dropping paths, whereas UX soundness allows for dropping paths but not information.

Path maintenance, illustrated through branching strategies. When updating and removing parts of memory, there are multiple ways to handle "branching", i.e., situations where there are multiple potential parts of memory to update/remove, sometimes referred to as "matches". To exemplify, we discuss branching in the context of $\mathsf{consume}_{\mathsf{Res}}$ for linear memory models (that is, variants of IDef. 7). For the discussion, it is important to have Fig. 2 fresh in mind. Say we have $\hat{\mu} = \{1 \mapsto 1, 2 \mapsto 1\}$ and $\hat{\pi} = 1 \leq \hat{x} \leq 2$ and are about to consume a resource assertion $x \mapsto 1$ knowing $x = \hat{x}$. Now consider the following three branching strategies, where $(\hat{E}_1, O') \in O(\mathrm{dom}(\hat{\mu}))$ denotes that we angelically pick an element from $\mathrm{dom}(\hat{\mu})$:

$$\frac{\hat{\mu} = \hat{\mu}_f \uplus \{\hat{E}_1 \mapsto \hat{E}_2\}}{\mathsf{consume}_{\mathsf{Res}}(m, O, \mapsto, [\hat{E}], \hat{\mu}) \rightsquigarrow (ok, O, [\hat{E}_2], (\hat{\mu}_f, \mathsf{true}, \hat{E} = \hat{E}_1))} \qquad \frac{(\hat{E}_1, O') \in O(\mathrm{dom}(\hat{\mu})) \quad \hat{\mu} = \hat{\mu}_f \uplus \{\hat{E}_1 \mapsto \hat{E}_2\}}{\mathsf{consume}_{\mathsf{Res}}(m, O, \mapsto, [\hat{E}], \hat{\mu}) \rightsquigarrow (ok, O', [\hat{E}_2], (\hat{\mu}_f, \hat{E} = \hat{E}_1, \hat{E} = \hat{E}_1))}$$

$$\frac{(\hat{E}_1, O') \in O(\mathrm{dom}(\hat{\mu})) \quad \hat{\mu} = \hat{\mu}_f \uplus \{\hat{E}_1 \mapsto \hat{E}_2\}}{\mathsf{consume}_{\mathsf{Res}}(m, O, \mapsto, [\hat{E}], \hat{\mu}) \rightsquigarrow (ok, O', [\hat{E}_2], (\hat{\mu}_f, \mathsf{true}, \hat{E} = \hat{E}_1))}$$

The *left rule* belongs to our running example linear memory model and the two other rules to variant models we have defined. The left rule branches over all possible matches; in our example, we get two branches: one branch with $\hat{\mu}_f = \{2 \mapsto 1\}$ and $\hat{\pi} = 1 \leq \hat{x} \leq 2 \land \hat{x} = 1$ and one branch with

$\hat{\mu}_f = \{1 \mapsto 1\}$ and $\hat{\pi} = 1 \leq \hat{x} \leq 2 \wedge \hat{x} = 2$. The *middle rule* implements unique-match branching since the rule is only applicable when there is a unique match. The rule is not applicable to our example as neither $\hat{x} = 1$ nor $\hat{x} = 2$ is implied by the current symbolic state. We have proved a linear memory model implementing this type of branching to be both OX sound and UX sound. Interestingly, the same rule but with conclusion $\cdots \rightsquigarrow (ok, O', [\hat{E}_2], (\hat{\mu}_f, \hat{E} = \hat{E}_1, \mathsf{true}))$ is OX sound but not UX sound because our CSE engine use the entire symbolic input state in the implication check – meaning that the implication might not hold w.r.t. the smaller output state. (With a stricter implication check requiring that the path condition, rather than the full symbolic state, implies the matching condition (in the example: $\hat{E} = \hat{E}_1$), the rule would be UX sound.) Lastly, the *right rule* implements, what we call, cut branching because the rule simply angelically picks one branch without checking if there are more matches. We have proved a linear memory model implementing this type of branching to be UX sound but not OX sound; the model is not OX sound because in OX reasoning we are not allowed to drop matches.

Information maintenance. Beyond variants of our running example linear memory model with different branching strategies, we have also mechanised and proven sound a memory-model instance for the traditional linear memory model from the OX literature, with concrete data type $\mathsf{Nat} \rightharpoonup_{fin} \mathsf{Val}$. This is an OX-only model: the model does not keep track of freed cells, using $\varnothing$, and can therefore not be proven UX sound (because it does not satisfy UX frame). Since it is an OX-only model, dropping information is allowed. As a simple illustration, consider Fig. 3, containing the successful `mutate` rule of our running example linear memory model. Note that the rule ensures that the information that the previous cell value successfully evaluates is kept by updating the path condition with $\hat{E}_{old} \in \mathsf{Val}$.[9] This is optional in OX-only models: our OX-only model is defined using rules that do not add evaluation information to the path condition, and we have still been able to prove the model to be OX sound.

$$\frac{\hat{\mu}(\hat{E}'_l) = \hat{E}_{old} \qquad \hat{\mu}' = \hat{\mu}[\hat{E}'_l \mapsto \hat{E}] \qquad \hat{\pi}' = (\hat{E}_l = \hat{E}'_l \wedge \hat{E}_{old} \in \mathsf{Val})}{\hat{\mu}.\mathtt{mutate}([\hat{E}_l, \hat{E}]) \rightsquigarrow ok : (\hat{\mu}', \hat{\pi}', [])}$$

Fig. 3. Successful `mutate` rule.

6.2 Fractional Ownership Memory Model

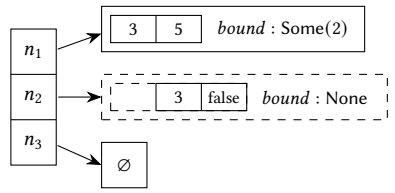
To illustrate that different ownership disciplines fit into our CSE theory, we have mechanised and proved OX and UX sound a linear memory model with fractional ownership [5, 6] rather than exclusive ownership, as utilised in the memory models discussed up to this point. A memory model similar to the fractional ownership memory model we discuss here has previously been implemented in an experimental branch of Gillian and tested on a small set of hand-written examples.

Model description. The memory model is best explained in terms of resources (i.e., IDef. 3). Points-to assertions for the model are of the form $n \overset{q}{\mapsto} v$, where $q \in (0, 1] \subset \mathsf{Rat}^+$ specifies the amount of ownership. *Less-than-1 ownership* ($q < 1$) gives read permission to the location n, while *full ownership* ($q = 1$) gives both read and write permissions. The other IDefs. of the model are relatively straightforward extensions of our running example memory model. In particular, the concrete memory data type of the model is $\mathsf{Nat} \rightharpoonup_{fin} ((\mathsf{Val}, \mathsf{Rat}^+) \uplus \{\varnothing\})$ and the symbolic memory data type is derived from this data type by symbolic lifting, i.e., $\mathsf{LExp} \rightharpoonup_{fin} ((\mathsf{LExp}, \mathsf{LExp}) \uplus \{\varnothing\})$. The implementation of memory actions, $\mathsf{consume}_{\mathsf{Res}}$, and $\mathsf{produce}_{\mathsf{Res}}$ are, as one would expect, also similar but with additional ownership checks. For illustration, we show the two successful rules of $\mathsf{consume}_{\mathsf{Res}}$ for points-to assertions:[10]

[9] This is not the only way to ensure this. It is also possible to, e.g., maintain an invariant saying that the path condition must include this type of information. The point being made here is that this needs to be ensured *in some way*.

[10] The "$\hat{E}'_l \notin \mathrm{dom}(\hat{\mu}_f)$" expression in the first rule is needed for UX soundness, to not drop disjointedness information.

$$\begin{array}{rcl} \mu & \in & \mathsf{CMem} \\ \mu(n_1) & = & (\{0 \mapsto 3, 1 \mapsto 5\}, \mathsf{Some}(2)) \\ \mu(n_2) & = & (\{1 \mapsto 2, 2 \mapsto \mathsf{false}\}, \mathsf{None}) \\ \mu(n_3) & = & \varnothing \end{array}$$



$$\begin{array}{l} (n_1, 0) \mapsto 3 \star \\ (n_1, 1) \mapsto 5 \star \\ \mathsf{Bound}(n_1; 2) \star \\ (n_2, 1) \mapsto 3 \star \\ (n_2, 2) \mapsto \mathsf{false} \star \\ n_3 \mapsto \varnothing \end{array}$$

Fig. 4. Example block-offset memory instance μ expressed formally (left), visually (centre), and as a composition of resource assertions (right).

$$\frac{\hat{\mu} = \hat{\mu}_f \uplus \{\hat{E}'_l \mapsto (\hat{E}_v, \hat{E}'_q)\}}{\begin{array}{c}\mathsf{consume}_{\mathsf{Res}}(m, O, \mapsto, [\hat{E}_l, \hat{E}_q], \hat{\mu}) \rightsquigarrow \\ (ok, O, [\hat{E}_v], (\hat{\mu}_f, \mathsf{true}, \hat{E}_l = \hat{E}'_l \wedge \hat{E}_q = \hat{E}'_q \wedge \hat{E}'_l \notin \mathsf{dom}(\hat{\mu}_f)))\end{array}} \qquad \frac{\begin{array}{c}\hat{\mu} = \hat{\mu}_f \uplus \{\hat{E}'_l \mapsto (\hat{E}_v, \hat{E}'_q)\} \\ \hat{\mu}' = \hat{\mu}_f \uplus \{\hat{E}'_l \mapsto (\hat{E}_v, \hat{E}'_q - \hat{E}_q)\}\end{array}}{\begin{array}{c}\mathsf{consume}_{\mathsf{Res}}(m, O, \mapsto, [\hat{E}_l, \hat{E}_q], \hat{\mu}) \rightsquigarrow \\ (ok, O, [\hat{E}_v], (\hat{\mu}', \mathsf{true}, \hat{E}_l = \hat{E}'_l \wedge \hat{E}_q < \hat{E}'_q))\end{array}}$$

6.3 Block-Offset Memory Model for C

Our CSE theory is not limited to different variants of the linear memory model. To illustrate this, we have mechanised and proved OX and UX sound a block-offset memory model for C. Originally inspired by the memory model of the verified CompCert C compiler [29], the model has previously been implemented in Gillian and has been used in Gillian-based teaching, but no detailed definition or soundness results have previously been given.

We describe, component by component, the concrete-memory instance data $(\mathsf{CMem}, \mathcal{W}f, \mu_\emptyset, \cdot)$ for IDef. 1. The memory data type is as follows:

$$\mathsf{CMem} \stackrel{\text{def}}{=} \mathsf{Nat} \rightharpoonup_{fin} (\mathsf{CMem}_\mathrm{B} \uplus \{\varnothing\}) \quad \text{where} \quad \mathsf{CMem}_\mathrm{B} \stackrel{\text{def}}{=} (\mathsf{Nat} \rightharpoonup_{fin} \mathsf{Val}, \mathsf{Nat}?)$$

using notation $t?$ to denote the option type for type t, with constructors None and Some. The concrete memory comprises two parts: CMem is a mapping from block identifiers to blocks; CMem_B is a block comprising a linear array and a bound indicating the fixed size of the array. (In C terms, the block identifiers are essentially pointers returned by `malloc()`, and the blocks describe the contents and size of the corresponding allocated memory.) This data structure allows us to represent *partial blocks*, required to be able to define $\models_{\mathsf{Res}}$, i.e., IDef. 3, which we discuss shortly. Fig. 4 gives an example of a partial concrete memory given by map μ with domain of block identifiers $\{n_1, n_2, n_3\}$. The mapping $\mu(n_1)$ is a *complete* block as the bound is 2 and both cells are present in the block. The mapping $\mu(n_2)$ is a *partial* block due to both the missing bound and the missing map at offset 0. The mapping $\mu(n_3)$ is a deallocated block, denoted by $\varnothing$.

The well-formedness condition $\mathcal{W}f$ provides constraints on the formation of blocks. Block-offset memories may not be well-formed for two reasons: first, a memory such as $\{3 \mapsto (\{0 \mapsto 1, 1 \mapsto 0\}, \mathsf{Some}(1))\}$ is not well-formed because its cells do not respect the bound; second, and more interestingly, a memory such as $\{3 \mapsto (\emptyset, \mathsf{None})\}$, which comprises an *empty block* is not well-formed since such blocks break the frame properties of the model (IProp. 2). We elaborate on this aspect more when we introduce concrete memory actions later (IDef. 2).

The empty memory is the empty mapping $\mu_\emptyset = \emptyset$. Note that this trivially satisfies $\mathcal{W}f$.

The definition of $\cdot$ is slightly complex. We exemplify using Fig. 4. Given another memory $\mu' = \{n_2 \mapsto (\{0 \mapsto 1, 3 \mapsto 0\}, \mathsf{Some}(4))\}$, we have the composition $\mu \cdot \mu'$ which is a mapping that gives the same results as μ for block identifiers n_1 and n_3 and, for n_2, gives:

$$(\mu \cdot \mu')(n_2) = (\{0 \mapsto 1, 1 \mapsto 3, 2 \mapsto \mathsf{false}, 3 \mapsto 0\}, \mathsf{Some}(4))\}.$$

We use the memory $\mu'' = \{n_2 \mapsto (\{1 \mapsto 12\}, \text{Some}(2))\}$ to illustrate the two reasons why blocks may fail to compose. Indeed, the composition $\mu \cdot \mu''$ is not defined for two reasons: first, the addresses of the blocks at n_2 overlap (i.e., $\text{dom}(\text{fst}(\mu(n_2))) \cap \text{dom}(\text{fst}(\mu''(n_2))) \neq \emptyset$); second, the addresses of the block $\mu(n_2)$ are not contained within the bound of the block $\mu''(n_2)$, meaning that if the composition would be defined, then the result would not satisfy $\mathcal{W}f$.

We now cover memory actions (IDef. 2). The successful rules of `new` and `free` are given below:

$$\frac{n_b \notin \text{dom}(\mu) \quad \mu_b = (\{0 \mapsto \texttt{null}, \ldots, n-1 \mapsto \texttt{null}\}, \text{Some}(n))}{\mu.\texttt{new}([n]) \rightsquigarrow \mathit{ok} : (\mu[n_b \mapsto \mu_b], [n_b])} \qquad \frac{\mu(n_b) = (\mu_b, \text{Some}(n)) \quad |\mu_b| = n}{\mu.\texttt{free}([n_b]) \rightsquigarrow \mathit{ok} : (\mu[n_b \mapsto \varnothing], [])}$$

Because we are now working with blocks, allocation returns a fresh, complete block, and freeing a block deallocates an entire complete block given a block identifier. Load and store operations now also take in the offset as input in addition to the block identifier (and also value for store), and these operations only require components necessary to load or store, i.e., requiring the relevant partial block and not the complete block.

Going back to why empty blocks are not allowed by $\mathcal{W}f$, note that UX frame breaks when allocation returns a block identifier n_b pointing to a fresh, complete block, but the frame contains the same n_b pointing to an empty block. OX frame instead breaks when you free a block identifier n_b pointing to a complete block, but the frame contains the same n_b pointing to an empty block.

We now define the resource assertions and their satisfaction relation (IDef. 3):

$$\begin{array}{lll} \mu \models_{\text{Res}} (n_b, n_o) \mapsto v & \Leftrightarrow & \mu = \{n_b \mapsto (\{n_o \mapsto v\}, \text{None})\} \\ \mu \models_{\text{Res}} \text{Bound}(n_b; n) & \Leftrightarrow & \mu = \{n_b \mapsto (\emptyset, \text{Some}(n)\} \\ \mu \models_{\text{Res}} n_b \mapsto \varnothing & \Leftrightarrow & \mu = \{n_b \mapsto \varnothing\} \end{array}$$

The resource assertions consist of: the cell assertion, the bound assertion, and the freed cell assertion. Note that these resource assertions represent the smallest unit of memory from which to build larger memory using the separating conjunction. For example, the μ memory model from Fig. 4 can be represented by the assertion given on the right of the figure. Note that when we defined CMem_B (for IDef. 1), we used $(\text{Nat} \rightharpoonup_{\mathit{fin}} \text{Val}, \text{Nat?})$ instead of the simpler $[\text{Val}]$. This is to ensure we can define resource assertions for each unit of memory. If the concrete memory model used lists instead of finite maps, then defining $\models_{\text{Res}}$ would become impossible since the relation must define the entire memory in the relevant block.

The instance data $(\text{SMem}, \hat{\mu}_\emptyset)$ for IDef. 4 is as follows. The memory data type SMem is a simple symbolic lifting of CMem:

$$\text{SMem} \stackrel{\text{def}}{=} \text{LExp} \rightharpoonup_{\mathit{fin}} (\text{SMem}_\text{B} \uplus \{\varnothing\}) \quad \text{where} \quad \text{SMem}_\text{B} \stackrel{\text{def}}{=} (\text{LExp} \rightharpoonup_{\mathit{fin}} \text{LExp}, \text{LExp?})$$

The empty symbolic memory is, unsurprisingly, $\hat{\mu}_\emptyset \stackrel{\text{def}}{=} \emptyset$. Because the memory data type is a symbolic lifting, $\models_{\text{Mem}}$ (IDef. 5) is straightforward to define. The symbolic action semantics (IDef. 6) is also symbolically lifted from its concrete counterpart and similarly straightforward. Lastly, the implementation of $\text{consume}_{\text{Res}}$ and $\text{produce}_{\text{Res}}$ (IDef. 7) are also straightforward.

6.4 Memory Model for Object-Oriented Languages

Our CSE theory is not limited to low-level languages such as C, it is also compatible with high-level languages such as object-oriented languages like JavaScript and Python. In fact, the Gillian project that inspired our theory has a strong history of JavaScript support, starting from its predecessor JaVerT [17, 18] (an analysis tool specific to JavaScript). For example, Gillian has been used to test the data structure library Buckets.js [19, 51] and to verify a JavaScript implementation of a message header deserialisation module in the AWS Encryption SDK [35].

We now briefly describe the JavaScript memory model implemented in Gillian to show that it fits our theory. The model is a variant of the block-offset memory model introduced in the previous section; because of the large overlap between the models, we have not mechanised this JavaScript model.

Model description. The concrete and symbolic memory data types of the JavaScript memory model implemented in Gillian are as follows (i.e., the data types of IDefs. 1 and 4):[11]

$$\mathsf{CMem} \stackrel{\text{def}}{=} \mathsf{Nat} \rightharpoonup_{fin} (\mathsf{CMem_B}, \{\mathsf{Str}\}?) \quad \text{where} \quad \mathsf{CMem_B} \stackrel{\text{def}}{=} \mathsf{Str} \rightharpoonup_{fin} (\mathsf{Val} \uplus \{\varnothing\})$$
$$\mathsf{SMem} \stackrel{\text{def}}{=} \mathsf{LExp} \rightharpoonup_{fin} (\mathsf{SMem_B}, \{\mathsf{LExp}\}?) \quad \text{where} \quad \mathsf{SMem_B} \stackrel{\text{def}}{=} \mathsf{LExp} \rightharpoonup_{fin} (\mathsf{LExp} \uplus \{\varnothing\})$$

where $\{\mathsf{Str}\}$ and $\{\mathsf{LExp}\}$ denote sets of Strs and LExps, respectively, and $\mathsf{CMem_B}$ and $\mathsf{SMem_B}$ represent JavaScript objects. The reader should compare these memory data types with the memory data types of the block-offset memory model and note the following differences. First, note that offsets (natural numbers) here have been replaced by property names (strings). Because of this, the bound from the block-offset memory model has been replaced by a set of strings, representing the "domain" of the object.[12] Second, in JavaScript, objects cannot be deallocated, but $\varnothing$-annotations for negative information are needed for a different reason. In JavaScript, reading fields that *have not been set* or *have been deleted* evaluates to `undefined` (see the extended paper [31] for an example JavaScript REPL session where properties are added and deleted). To not break the frame properties of the model, such "unset" properties must be annotated with $\varnothing$.

The memory model has the following memory actions (IDef. 2):

$$\mathsf{x} := \mathtt{newObj}(), \mathtt{deleteField}(\mathsf{E}, \mathsf{E}), \mathsf{x} := \mathtt{lookup}(\mathsf{E}, \mathsf{E}), \mathtt{mutate}(\mathsf{E}, \mathsf{E}, \mathsf{E}).$$

E.g., $\mathtt{deleteField}(o, f)$ deletes field f from the object at address o and $\mathtt{mutate}(o, f, v)$ sets field f in the object at address o to v. While the semantics of JavaScript is complex, this simple memory model is enough to capture its basic operations: in the JavaScript instantiation of Gillian, JavaScript programs are compiled to GIL, its intermediate representation, where complex operations (such as looking up an object field by following the "prototype chain") are compiled to a sequence of lower-level operations that are either side-effect free, or one of the actions provided above.

Lastly, the semantics of the actions of the memory model are obtained by applying minor modifications to the actions of the block-offset memory model. For instance, out-of-bound accesses happen when a memory lookup is outside the object domain instead of when an offset is outside the bound of the block, and so on.

6.5 CHERI-Assembly Memory Model

To show that our CSE theory can fit novel memory models beyond the usual suspects, we have designed and mechanised a new symbolic memory model for a CHERI-enabled idealised assembly language. CHERI [53] is a recently introduced memory-model-based capability protection model: it guarantees runtime spatial memory safety via hardware, and this is achieved using *capabilities*: fat pointers that carry spatial metadata such as bound, permission, and a tag bit stating the validity of the capability, in addition to the memory address. Additional capability-aware instructions are added to the instruction set, where the *monotonic property* is preserved: valid capabilities cannot gain more bounds or permissions than what they originally had.

We have proved our memory model OX sound, and plan to prove UX soundness and implement the memory model in Gillian in future work. To our best knowledge, our memory model is the

[11] The memory model in Gillian additionally includes object metadata, which we do not discuss here.

[12] This change also leads to a slightly different well-formedness condition. Informally, a memory $\mu \in \mathsf{CMem}$ is only well-formed if $\forall (o, \mathsf{Just}(d)) \in \mathrm{codom}(\mu).\, \mathrm{dom}(o) \subseteq d$ (see heap-domain invariant [39]). The rest of the well-formedness condition is similar to the block-offset memory model.

$$\begin{array}{rcl} \mathsf{Cap} & \stackrel{\text{def}}{=} & \{blo : \mathsf{Nat};\ off : \mathsf{Nat};\ base : \mathsf{Nat};\ len : \mathsf{Nat}; \vec{perm}_x : \vec{\mathsf{Bool}};\ tag : \mathsf{Bool}\} \\ & & \quad \text{where } x \in \{\mathsf{load}, \mathsf{store}, \ldots\} \\ \mathsf{Cap}_{\mathsf{frag}} & \stackrel{\text{def}}{=} & \{cap : \mathsf{Cap}, nth : \mathsf{Nat}\} \end{array}$$

$$\begin{array}{rcl} \mathsf{CMem}_{\text{B-CH}} & \stackrel{\text{def}}{=} & (\mathsf{Nat} \rightharpoonup_{fin} \mathsf{Val} + \mathsf{Cap}_{\mathsf{frag}}, \mathsf{Nat?}) \\ \mathsf{CMem}_{\text{BO-CH}} & \stackrel{\text{def}}{=} & \mathsf{Nat} \rightharpoonup_{fin} (\mathsf{CMem}_{\text{B-CH}} \uplus \varnothing) \\ \mathsf{CMem}_{\mathsf{CReg}} & \stackrel{\text{def}}{=} & \mathsf{Nat} \rightharpoonup_{fin} \mathsf{Cap} \\ \mathsf{CMem} & \stackrel{\text{def}}{=} & (\mathsf{CMem}_{\mathsf{CReg}}, \mathsf{CMem}_{\text{BO-CH}}) \end{array}$$

Fig. 5. The concrete memory data type.

$$\begin{array}{rcl} \mu & = & (\mu_{\text{R}}, \mu_{\text{BO}}) \\ (\mu_{\text{R}}, \mu_{\text{BO}}) & \in & \mathsf{CMem} \\ \mu_{\text{BO}}(n_1) & = & (\{0 \mapsto 3, 1 \mapsto c_1\}, \mathsf{Some}(2)) \\ \mu_{\text{BO}}(n_2) & = & (\{i \mapsto c_i\}, \mathsf{Some}(|\mathsf{Cap}|) \\ & & \text{for } i \in \{0, 1, \ldots, |\mathsf{Cap} - 1|\} \\ \mu_{\text{R}}(r_1) & = & c \end{array}$$

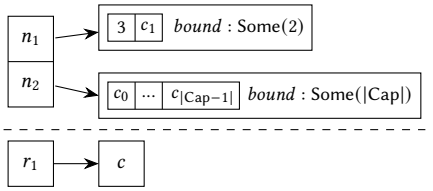


Fig. 6. Example CHERI memory instance μ expressed formally (left), visually (centre), and as a composition of resource assertions (right).

first memory model for CHERI that supports SL-based symbolic execution and also comes with a soundness theorem. Details about related work are given in §7.

We first discuss the definition of our new memory model. The CHERI-assembly model is the most substantial instantiation in this work: the CHERI-assembly model has roughly 19K lines of code, about 5 times larger than the block-offset model, the second most substantial instantiation. Afterwards, we discuss our design process. The design process is interesting because the memory model is the first memory model we have designed for our CSE theory without the guidance of an existing implementation. We explain how our CSE theory guided us to obtain the appropriate design: we made two failed design attempts before finally arriving at our current design.

Model description. Our CHERI memory model extends the block-offset memory model of §6.3 to be capability-aware. While the bit-level layout of capabilities may differ between architectures, even if the metadata is mostly similar; in this work, we work with a CHERI-assembly model with an abstract and architecture-agnostic design.

We now give the instance data $(\mathsf{CMem}, \mathcal{W}f, \mu_\emptyset, \cdot)$ conforming to IDef. 1; we first discuss CMem. Fig. 5 shows the structure of the CHERI-assembly concrete memory model, and Fig. 6 gives as an instance example μ. There are two main differences with the block-offset memory model: a separate mapping for capability registers (i.e. $\mathsf{CMem}_{\mathsf{CReg}}$) is added, and the main memory (i.e. $\mathsf{CMem}_{\text{BO-CH}}$) is extended to be capability-aware. For capability registers, we use the abstract capability Cap, which contains spatial metadata of capabilities. For the main memory $\mathsf{CMem}_{\text{BO-CH}}$, we extend the block-offset model by also storing *capability byte fragments*, represented as $\mathsf{Cap}_{\mathsf{frag}}$, in addition to standard values Val. In Fig. 6, we observe $\mu_{\text{BO}}(n_1)$ contains the capability byte fragment c_1, which is the second byte fragment of some capability c. When a capability is stored in memory, the capability is stored as a sequence of abstract, contiguous, well-formed capability byte fragments – and we can observe this in $\mu_{\text{BO}}(n_2)$, where $|\mathsf{Cap}|$ is the size of a capability for a given architecture. We note that each capability byte fragment also stores a *tag fragment*, and a capability in memory only has its tag bit set to true if and only if the tag fragment of all the capability fragments is set to true. Usually, CHERI architectures store tags in the *tagged memory*, separate from the main memory; in our model the two memories are merged, and tags are split into tag fragments – the motivation behind this is explained when we discuss the design process below.

The well-formedness condition $\mathcal{W}f$ also extends from that of the block-offset memory model. The additional constraints relate to the well-formedness of capability fragments in the memory. One obvious property is that the fragment value of $\mathsf{Cap}_{\mathsf{frag}}$ should be between 0 and $|\mathsf{Cap} - 1|$. Another property is capability fragments whose tag fragment bit is set to true must be stored in the appropriate capability-offset-aligned position (the formal description is in the extended paper [31]). This conforms to the specification that valid capabilities in memory are stored in a capability-aligned position [52]. All the capability fragments in $\mu_{\mathrm{BO}}(n_1)$ and $\mu_{\mathrm{BO}}(n_2)$ are stored in a capability-off-aligned position, which makes the overall memory well-formed; but note that if we instead have $\mu_{\mathrm{BO}}(n_1)(1) = c_4$, then the memory is well-formed only if $c_4.cap.tag = \mathsf{false}$. As we will see below, $\models_{\mathsf{Res}}$ must account for this too, and part of $\mathcal{W}f$ is expressed in $\models_{\mathsf{Res}}$.

The empty memory $\mu_\emptyset$ can be straightforwardly defined as $(\emptyset, \emptyset)$ (which satisfies $\mathcal{W}f$), and $\cdot$ is also a straightforward extension of that of the block-offset memory model.

We now discuss memory actions (IDef. 2). There are more than 100 memory-action rules, with relatively complex definitions. To exemplify, we discuss the successful case of the capability store action, shown in Fig. 7. The capability store action takes in r_s and r_d, which are capability register numbers pointing to capability registers c_s and c_d, respectively. The idea is that we store c_d in the location pointed by c_s. The action then performs necessary spatial checks and throws relevant errors when a spatial safety property is violated, e.g. c_s must have the tag bit set to true, and the offset of c_s must be within bound and is capability-offset-aligned, etc. Afterwards, the store_capability function stores c_d as a sequence of well-formed contiguous capability byte fragments in the main memory.

$$\frac{\begin{array}{c}\mathsf{fst}(\mu)(r_s) = c_s \qquad c_s.tag = \mathtt{true} \\ c_s.perm_{\mathsf{store}} = \mathtt{true} \\ r_d \text{ is a capability register} \qquad \mathsf{fst}(\mu)(r_d) = c_d \\ (c_s.perm_{\mathsf{storecap}} = \mathtt{true} \vee c_d.tag = \mathtt{false}) \\ c_s.off + |\mathsf{Cap}| \leq c_s.base + c_s.len \\ c_s.off \geq c_s.base \qquad c_s.off \,\%\, |\mathsf{Cap}| = 0 \\ \mathsf{snd}(\mu)(c_s.blo) = (\mu_b, \mathsf{Some}(m)) \\ c_s.off + |\mathsf{Cap}| \leq m \\ \{c_s.off, \ldots, c_s.off + |\mathsf{Cap}| - 1\} \subseteq \mathsf{dom}(\mu_b) \\ \mathsf{store\_capability}(c_s, c_d, \mu_b) = \mu_b' \\ \mu' = (\mathsf{fst}(\mu), \mathsf{snd}(\mu)[c_s.blo \mapsto (\mu_b', \mathsf{Some}(m))])\end{array}}{\mu.\mathtt{store}([r_s, r_d]) \rightsquigarrow ok : (\mu', [])}$$

Fig. 7. Semantics of the capability store action.

We now discuss the resources of this model and their satisfaction relation $\models_{\mathsf{Res}}$ (IDef. 3). The three resources used in the block-offset memory model are directly ported. Additionally, we have two new resources: $\mathrm{Reg}(r_n; c)$, which describes that at register r_n the capability c is stored, and $(n_b, n_o) \mapsto_{cf} c_n$, which states that at block n_b and offset n_o, the capability byte fragment c_n is stored, where n denotes the nth byte fragment. The resource satisfaction relation $\models_{\mathsf{Res}}$ is given below:

$$\begin{array}{lll}\mu \models_{\mathsf{Res}} \mathrm{Reg}(r_n; c) & \Leftrightarrow & \mu = (\{r_n \mapsto c\}, \emptyset) \\ \mu \models_{\mathsf{Res}} (n_b, n_o) \mapsto_{cf} c_n & \Leftrightarrow & \mu = (\emptyset, \{n_b \mapsto (\{n_o \mapsto c_n\}, \mathsf{None})\}) \\ & & \wedge\ (c_n.cap.tag = \mathsf{true} \implies n_o \,\%\, |\mathsf{Cap}| = n) \wedge\ n < |\mathsf{Cap}|\end{array}$$

Whereas defining $\models_{\mathsf{Res}}$ for the block-offset memory was straightforward, defining $\models_{\mathsf{Res}}$ here is slightly more involved. Due to $\mathcal{W}f$ of CHERI, we cannot allow capability byte fragments whose tag fragment bit is true to be stored anywhere, and we require the fragment value to be valid. The assertion $(n_1, 1) \mapsto_{cf} c_1$ in Fig. 6 is satisfiable, but $(n_1, 1) \mapsto_{cf} c_4$ is not if $c_4.cap.tag = \mathsf{true}$.

Note that one can define a full, valid capability resource assertion as a user-defined predicate assertion $(n_b, n_o) \mapsto_{\mathsf{cap}} c$ as follows:

$$(n_b, n_o) \mapsto_{\mathsf{cap}} c \stackrel{\mathrm{def}}{=} \bigstar_{i=0}^{|\mathsf{Cap}|-1} (n_b, n_o + i) \mapsto_{cf} c_i$$

In Fig. 6, we can see the capability register mapping is represented using the register assertion, the capability fragment in $\mu_{\mathrm{BO}}(n_1)$ is represented using the capability fragment assertion, and the full capability in $\mu_{\mathrm{BO}}(n_2)$ is represented using the user-defined capability predicate assertion.

The symbolic memory (IDef. 4) is a direct lifting of its concrete counterpart. The satisfaction relation $\models_{\mathsf{Mem}}$ (IDef. 5) extends that of the block-offset model by additionally relating symbolic capability registers to concrete ones. The symbolic memory actions (IDef. 6) are a symbolic lifting of the concrete actions. The implementations of $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ (IDef. 7) extend those of the block-offset model: there is now an additional case when consuming or producing capability fragments depending on the tag fragment value of the capability fragment due to $\mathcal{W}f$.

Design process. Our first attempt at designing the memory model was based on that of Park et al. [43]. That work separated the main memory into two: the (tagless) main memory, and the *tagged* memory, where the tagged bit of a capability was stored in the tagged memory. While this model closely represented the CHERI hardware, the separation made it difficult to reason about the complex inter-dependency between the two memories, making it difficult to formalise resource assertions well-formed with respect to the concrete memory and also prove the OX frame property.

In our second attempt, to address the aforementioned issue, we introduced the notion of a "chunk" of memory, where a chunk is either a capability or a sequence of values and capability fragments of size $|\mathsf{Cap}|$. We removed the tagged memory and made capability tags implicitly defined depending on whether the chunk was a capability or not. Because this model no longer separates the main memory, there are no inter-dependencies between memories and no troubles proving the OX frame property or formalising well-formed resource assertions. However, we discovered writing function specification had limitations, e.g. when the precondition requires a capability fragment, but the memory comprises the full capability instead, which made the model not truly compositional.

Our third and final attempt introduced the notion of tag fragments in capability fragments. This ensured true compositionality, unlike the previous attempt, where there are no limitations on how to write specifications, whilst avoiding complex inter-dependencies between memories.

The structure of the concrete memory model naturally guided us to define the current resource assertions and their satisfaction relation. Indeed, this formalisation gave us confidence that our parametric CSE theory is well designed: while this work was done independently from the recently published Iris-MSWasm work [27], we ended up with resource assertions essentially similar to those used in the Iris-MSWasm work.

6.6 VeriFast-and-Viper-Inspired Memory Model for C

We have mechanised a memory model for C inspired by the OX verification platforms VeriFast and Viper. Specifically, we have ported the memory model of Featherweight VeriFast [22] (FVF), a formalisation of (a simplified version of) VeriFast, to our CSE theory and proved it OX sound. The motivation for this work is as follows. Beyond Gillian, VeriFast and Viper are the most well-known consume-and-produce-based CSE platforms. VeriFast and Viper have similar memory models: they both maintain a flat collection of "heap chunks" (explained below). The memory model we discuss in this section shows that our CSE theory can fit such memory models.

Model description. FVF analyses a simplified C language with the same memory actions as our running example memory model. In FVF, the concrete memory model (IDef. 1) is a multiset of concrete heap chunks. A concrete heap chunk is either a *points-to chunk* $l \mapsto v$ denoting that there is an allocated memory cell at address l whose current value is v, or a *malloc-block chunk* $mb(l, size)$ denoting that a memory block of size *size* was allocated at address l by `malloc`, i.e., that the memory cells at addresses l through $l + size - 1$ are part of a single block, which will be freed as one unit when `free` is called with argument l. The memory composition operator is multiset union, and all heap chunks are disjoint. In our instantiation, we represent concrete heap chunks as follows:

$$\mathsf{cchunk} \stackrel{\text{def}}{=} \mathsf{CCPointsTo}(\mathsf{Nat}, \mathsf{Val}) \mid \mathsf{CCMB}(\mathsf{Nat}, \mathsf{Nat})$$

The symbolic memory model (IDef. 4) used in FVF is a multiset of symbolic heap chunks. A symbolic heap chunk is either a points-to chunk, a malloc-block chunk, or a user-defined-predicate chunk. Since user-defined predicates are handled independently of the memory model in our CSE theory, the symbolic chunks of our memory-model instance are as follows:

$$\mathsf{schunk} \stackrel{\text{def}}{=} \mathsf{SCPointsTo}(\mathsf{LExp}, \mathsf{LExp}) \mid \mathsf{SCMB}(\mathsf{LExp}, \mathsf{LExp})$$

The memory actions of the concrete and symbolic memory models (IDefs. 2 and 6 respectively) and $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations of the symbolic memory model (IDef. 7) are straightforward. In FVF, the semantics of the memory actions are simply defined in terms of the consume and produce operations (concrete consume and produce operations are defined for the concrete memory model) – we therefore do not include any memory actions in our instance. The $\mathsf{consume}_{\mathsf{Res}}$ and $\mathsf{produce}_{\mathsf{Res}}$ operations (IDef. 7) of FVF implement unique-match branching (as discussed in §6.1) – our ported implementations therefore do the same.

7 Related Work

Program logics. Multiple program logics – such as abstract separation logic [11], views [15, 47], and Iris [24] – are parametric on different PCM-like structures describing memory state and ghost state. Some of these program logics also feature other types of parametricity, such as programming-language parametricity. In contrast to CSE, program logics only describe sound inferences rather than a way to automate reasoning. For memory-model parametricity, the parameters we introduce in this paper show what is sufficient to animate reasoning and ensuring soundness of this animation.

Compositional symbolic execution. Since we have already discussed the previous work on foundations of memory-model-parametric CSE [19, 35] in the introduction and overview of this paper, we only discuss memory-model-monomorphic foundations here.

Lööw et al. [32] is the only previous work on CSE theory that treats both OX and UX soundness. The work is inspired by Gillian but monomorphised to the memory model we use as a running example in this paper. The work is similar to ours in scope in terms of engine features covered (function calls, user-defined predicates, etc.). Although the work is not mechanised, it is the monomorphic work that has influenced us the most; in particular, our consume and produce properties are inspired by the consume and produce properties they introduce, which they like us use to ensure interoperability of CSE analysis results with program logics and analysis tools built on top of program logics.

OX-only CSE is the most well-explored variant of CSE. We list the most significant projects in chronological order: Appel [1] mechanises a subset of Smallfoot; Jacobs et al. [22] mechanise a subset of VeriFast; Keuchel et al. [25] argue for the use of Kripke specification monads in mechanising CSE and illustrate their techniques on small CSE case studies; Zimmerman et al. [58] formalise on paper a subset of Viper as part of larger work to enable sound gradual verification in Viper; Dardinier et al. [13] mechanise a soundness framework for translational verifiers, including CSE inspired by Viper. These projects have either smaller or similar coverage of engine features compared to us. Most closely related is the work by Dardinier et al. [13], which like us ensure interoperability of CSE analysis results. Whereas our approach to interoperability forms a semantic connection to program logics, through the satisfaction relation for assertions $\theta, (s, \mu) \models A$, Dardinier et al. instead connect up syntactically by proof reconstruction. Larger case studies, for both approaches, are needed to better evaluate the trade-offs between the two approaches.

Lastly, the Infer-Pulse work [26] treats only UX soundness for bi-abduction and is not mechanised. Since their engine is specialised to bi-abduction, their coverage of engine features is small.

Compilation to intermediate verification language. An alternative to symbolic execution is compilation to intermediate verification languages (IVLs) such as Boogie [4] and Why3 [16], which turns the problem of automating reasoning into a compilation problem. We know of no such work addressing memory-model parametricity. Other IVL topics have received formal treatments: e.g., Parthasarathy et al. [44] mechanise proof-producing compilation to IVLs and targets Boogie in one case study, and Cohen and Johnson-Freyd [12] mechanise the satisfaction relation of the logic fragment of Why3 and verify two compilation transformations inspired by Why3.

CHERI memory models. The most closely related previous work on CHERI have targeted CHERI-C, which extends the C language to support CHERI capabilities. There exist mechanised CHERI-C memory models formalised in Isabelle/HOL [42, 43] and Rocq [27, 56, 57]. The work of Park et al. [43] provides an extractable CHERI-C model usable for concrete execution in Gillian, and the work of Legoupil et al. [27] extends the Iris-Wasm work [48] to incorporate *handles*, a synonym for capabilities, and introduces resource assertions for handles. None of these works, however, cover symbolic execution. ESBMC-CHERI [7] is the only tool that supports symbolic execution of CHERI-C programs; however, the tool lacks a formal memory model and soundness proof and does not support compositional reasoning.

Additional interesting memory models. Gillian has two more memory-model instances which we have not discussed in this work. First, there is an optimised "block-of-trees" memory model for C which has been used in C verification case studies [35] (but not described in detail in previous publications). We will instantiate our CSE theory with this model in the future. Second, in work parallel to ours, Ayoun et al. [2] have instantiated Gillian to Rust. Gillian's Rust memory model comprises several components, including a core heap model that extends the block-of-trees model for C with support for polymorphism and unknown layouts required by Rust. The memory model also includes ghost state for lifetime and prophecy reasoning. The model should be expressible using our theory; with the minor exception of the model's novel automation for reasoning about mutable borrows, for which a small generalisation of how Gillian handles user-defined predicates was required. The handling of user-defined predicates in our theory can easily be generalised by moving it from the memory-model-independent part of our theory into each memory-model instance. A bigger obstacle to overcome is the fact that the soundness justification of the memory model relies on results from RustBelt [23] and RustHornBelt [36], requiring a formal connection between our theory and Iris to leverage these results.

Other interesting targets for future memory instantiations include well-validated formal memory models from, e.g., the Cerberus project [28, 37] or the WebAssembly project [20].

8 Conclusion

In this paper, we have introduced a formal foundation for memory-model-parametric CSE platforms for verification and/or bug-finding. Multiple research groups have in recent years turned their attention to formally defining and proving sound CSE tools and platforms; despite this flurry of activity, the analysis platform Gillian is today the only CSE platform that supports memory-model parametricity. We hope this paper will inspire and help other CSE projects to also implement memory-model parametricity. We have also discussed a series of memory-model instantiations of our CSE theory, some based on or inspired by instantiations developed for Gillian.

Looking forward, now having in place a formal definition of memory model for CSE, in particular, sufficient memory-model requirements for memory-model instantiations of our CSE engine to be sound, we are now in the process of developing a combinator library for memory models, as defined in this paper, to make it easy to develop and prove sound large and complex memory models by composing smaller memory-model components together.

Data-Availability Statement

The Rocq mechanisation of our CSE theory and its instantiations are available in our artefact [30].

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank Dawid Lachowicz for careful comments on earlier drafts of this paper.

This work was supported by EPSRC Fellowship "VetSpec: Verified Trustworthy Software Specification" (EP/R034567/1) (Lööw, Nantes-Sobrinho, and Gardner), Gardner's faculty gift from Meta (Lööw and Nantes-Sobrinho), the Amazon Research Award "Gillian-Rust: Unbounded Verification for Unsafe Rust Code" (Ayoun), and an Imperial College London Department of Computing PhD Scholarship (Park).

References

[1] Andrew W. Appel. 2011. VeriSmall: Verified Smallfoot Shape Analysis. In *Certified Programs and Proofs*. doi:10.1007/978-3-642-25379-9_18

[2] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025). doi:10.1145/3729289

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018). doi:10.1145/3182657

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*. doi:10.1007/11804192_17

[5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Symposium on Principles of Programming Languages*. doi:10.1145/1040305.1040327

[6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*. doi:10.1007/3-540-44898-5_4

[7] Franz Brauße, Fedor Shmarov, Rafael Menezes, Mikhail R. Gadelha, Konstantin Korovin, Giles Reger, and Lucas C. Cordeiro. 2022. ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. In *International Symposium on Software Testing and Analysis*. doi:10.1145/3533767.3543289

[8] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods Symposium*. doi:10.1007/978-3-642-20398-5_33

[9] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *Principles of Programming Languages*. doi:10.1145/1480881.1480917

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM* 58, 6 (2011). doi:10.1145/2049697.2049700

[11] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Symposium on Logic in Computer Science*. doi:10.1109/LICS.2007.30

[12] Joshua M. Cohen and Philip Johnson-Freyd. 2024. A Formalization of Core Why3 in Coq. *Proc. ACM Program. Lang.* 8, POPL (2024). doi:10.1145/3632902

[13] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2025. Formal Foundations for Translational Separation Logic Verifiers. *Proc. ACM Program. Lang.* 9, POPL (2025). doi:10.1145/3704856

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. doi:10.1007/978-3-540-78800-3_24

[15] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Symposium on Principles of Programming Languages*. doi:10.1145/2429069.2429104

[16] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *European Symposium on Programming*. doi:10.1007/978-3-642-37036-6_8

[17] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *Proceedings of the ACM on Programming Languages* 2, POPL (2018). doi:10.1145/3158138

[18] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). doi:10.1145/3290379

[19] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-language Platform for Symbolic Execution. In *Conference on Programming Language Design and Implementation*.

doi:10.1145/3385412.3386014

[20] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Conference on Programming Language Design and Implementation*. doi:10.1145/3062341.3062363

[21] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods Symposium*. doi:10.1007/978-3-642-20398-5_4

[22] Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* 11 (2015). Issue 3. doi:10.2168/LMCS-11(3:19)2015

[23] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2017). doi:10.1145/3158154

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming (JFP)* 28 (2018). doi:10.1017/S0956796818000151

[25] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified symbolic execution with Kripke specification monads (and no meta-programming). *Proc. ACM Program. Lang.* 6, ICFP (2022). doi:10.1145/3547628

[26] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022). doi:10.1145/3527325

[27] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024). doi:10.1145/3689722

[28] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: Verifying Real-World C Idioms with Integer-Pointer Casts. *Proc. ACM Program. Lang.* 6, POPL (2022). doi:10.1145/3498681

[29] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). doi:10.1145/1538788.1538814

[30] Andreas Lööw, Seung Hoon Park, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Opale Sjöstedt, and Philippa Gardner. 2025. Compositional Symbolic Execution for the Next 700 Memory Models (Artefact). doi:10.5281/ZENODO.16909361

[31] Andreas Lööw, Seung Hoon Park, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Opale Sjöstedt, and Philippa Gardner. 2025. Compositional Symbolic Execution for the Next 700 Memory Models (Extended Version). doi:10.48550/arXiv.2508.15576

[32] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. 2024. Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In *European Conference on Object-Oriented Programming*. doi:10.4230/LIPIcs.ECOOP.2024.25

[33] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Petar Maksimović, and Philippa Gardner. 2024. Matching Plans for Frame Inference in Compositional Reasoning. In *European Conference on Object-Oriented Programming*. doi:10.4230/LIPIcs.ECOOP.2024.26

[34] Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic. In *European Conference on Object-Oriented Programming*. doi:10.4230/LIPIcs.ECOOP.2023.19

[35] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification*. doi:10.1007/978-3-030-81688-9_38

[36] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Conference on Programming Language Design and Implementation*. doi:10.1145/3519939.3523704

[37] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL (2019). doi:10.1145/3290380

[38] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*. doi:10.1007/978-3-662-49122-5_2

[39] Daiva Naudziuniene. 2018. *An Infrastructure for Tractable Verification of JavaScript Programs*. Ph. D. Dissertation. Imperial College London.

[40] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic*. doi:10.1007/3-540-44802-0_1

[41] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming Languages and Systems*. doi:10.1007/978-3-662-49498-1_23

[42] Seung Hoon Park. 2022. A Formal CHERI-C Memory Model. *Archive of Formal Proofs* (November 2022). https://isa-afp.org/entries/CHERI-C_Memory_Model.html, Formal proof development.
[43] Seung Hoon Park, Rekha Pai, and Tom Melham. 2023. A Formal CHERI-C Semantics for Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. doi:10.1007/978-3-031-30823-9_28
[44] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language. *Proc. ACM Program. Lang.* 8, PLDI (2024). doi:10.1145/3656438
[45] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proceedings of the ACM on Programming Languages* 7, POPL (2023). doi:10.1145/3571194
[46] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*. doi:10.1007/978-3-030-53291-8_14
[47] Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent Incorrectness Separation Logic. *Proceedings of the ACM on Programming Languages* 6, POPL (2022). doi:10.1145/3498695
[48] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023). doi:10.1145/3591265
[49] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science*. doi:10.1109/LICS.2002.1029817
[50] Rocq. [n. d.]. The Rocq Prover. https://rocq-prover.org.
[51] Mauricio Santos. [n. d.]. Buckets-JS: A JavaScript Data Structure Library. https://github.com/mauriciosantos/Buckets-JS.
[52] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987
[53] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *International Symposium on Computer Architecture (ISCA)*. doi:10.1109/ISCA.2014.6853201
[54] Hongseok Yang. 2001. *Local Reasoning for Stateful Programs*. Ph. D. Dissertation. University of Illinois Urbana-Champaign.
[55] Hongseok Yang and Peter O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *Foundations of Software Science and Computation Structures*. doi:10.1007/3-540-45931-6_28
[56] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alexander Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2024. Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. doi:10.1145/3617232.3624859
[57] Vadim Zaliva, Kayvan Memarian, Brian Campbell, Ricardo Almeida, Nathaniel Filardo, Ian Stark, and Peter Sewell. 2025. A CHERI C Memory Model for Verified Temporal Safety. In *International Conference on Certified Programs and Proofs*. doi:10.1145/3703595.3705878
[58] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proceedings of the ACM on Programming Languages* 8, POPL (2024). doi:10.1145/3632927

Received 2025-03-26; accepted 2025-08-12