Memory Safety: Uniqueness as Separation

Pilar Selene Linares Arévalo[1](✉), Arthur Azevedo de Amorim[2], Vincent Jackson[1], Liam O'Connor[3], Peter Schachte[1], and Christine Rizkallah[1]

[1] The University of Melbourne, Melbourne, VIC, Australia
{linaresareva,v.jackson,schachte,christine.rizkallah}@unimelb.edu.au
[2] Rochester Institute of Technology, Rochester, NY, USA
[3] Australian National University, Canberra, ACT, Australia
liam.oconnor@anu.edu.au

Abstract. Programming languages with uniqueness type systems prevent pointer aliasing, simplifying memory safety reasoning. However, code implemented in these languages often interoperates through foreign function interfaces with external components implemented in languages lacking the same level of static safety guarantees. To verify safe updates in a combined system, one must manually verify that the external components preserve the safety invariants of the uniqueness type system. In particular, recent work showed that one can manually discharge such obligations on C components from a cross-language Cogent-C system by directly reasoning about the C code in higher-order logic. However, even for simple examples, discharging the uniqueness safety obligations, known as *frame conditions*, within a logic not specifically designed for direct reasoning in terms of heaps and pointers was not ideal. Separation logic is an established logic that facilitates reasoning about imperative programs by localising reasoning to the parts of the heap that the program mutates. This raises a vital question. *Can we use separation logic to discharge the safety obligations imposed by uniqueness types?* The answer is yes. This paper demonstrates that the frame conditions can be inferred from particular separation logic triples and, hence, discharged by reasoning using separation logic. We identify and verify the soundness of specific separation logic triples that imply the frame conditions imposed by a uniqueness type system.

Keywords: Memory Safety · Uniqueness Types · Separation Logic

1 Introduction

Modern languages such as Clean [15], SAC [16], Mercury [18], and more recently, Cogent [9] leverage uniqueness type systems [17], a form of substructural type systems, to rule out the largest known class of common software vulnerabilities: *memory safety errors*. In particular, uniqueness types are typically used in these languages to prevent *pointer aliasing*: having multiple live references to the same memory location. As such, they support the development of code that is *safe* by design and remove the need for garbage collection, enabling efficient compilation.

However, the safety guarantees of uniqueness typing are limited to code that adheres to the type system constraints. While the uniqueness condition is a conceptually simple restriction, it sometimes imposes a considerable burden when writing code in such languages. For example, a simple uniqueness type system would prohibit passing both an array and a reference to one of its elements to a function because of the aliasing this introduces, even if we only read from them, and no mutation is involved.

Therefore, in practice, languages with uniqueness types often include an *opt-out mechanism*, to enable interoperation with external components written in *unsafe* fragments or unsafe languages, such as C, which do not enforce safety by design. This often involves using a foreign function interface (FFI) to enable interoperation with surrounding infrastructure, such as system calls, and with existing components such as external libraries.

For instance, SaC, a functional programming language with uniqueness types, provides an FFI to interoperate with external C libraries [5]. Similarly, the Cogent language [9] has an FFI to act as a bypass mechanism that enables interoperation with C components.

To verify the memory safety of a combined system, one must manually verify that the external components preserve the invariants of the uniqueness type system that result in safety. Leaving these unverified could invalidate the guarantees obtained from using uniqueness types, even for the safe components. Recent work demonstrated that one can manually discharge such uniqueness conditions on C code in the context of a cross-language Cogent-C system, by directly reasoning about the C code in higher-order logic. This shows that composing proofs in such a cross-language setting is possible and that it is possible to discharge the uniqueness conditions on foreign code. However, even for a simple example, discharging uniqueness *frame conditions* in a logic not designed to verify such safety obligations was quite tedious [4].

These proofs should ideally be developed in a framework that facilitates reasoning about memory. One such framework is separation logic [14], a logic for reasoning about heap-manipulating programs that enables *local* reasoning about separate parts of memory, through separating conjunction and the frame rule.

This paper presents an imperative language with specific features for reasoning about memory safety. In particular, we extend an existing imperative language [1] to distinguish memory safety errors from other type of errors and present a sound separation logic for this extension. Moreover, we present separation logic triples FC Triple and prove that they imply the uniqueness type invariants imposed by Cogent on foreign C code, on the shared heap between Cogent and C, called through Cogent's FFI, which allows for verified interoperability of Cogent-C code [4]. We demonstrate that the FC Triple we identified can be discharged for a number of language constructs, including memory-related constructs. While the formal proofs do not form a contribution of this paper, our core results have been formalised in ROCQ (formerly Coq) to gain higher confidence in our results.

2 Enforcing Uniqueness on Foreign Functions

For this paper, we consider the memory safety invariants from Cogent, which is a functional programming language intended for low-level software and designed to write code in a safe, verifiable way without a garbage collector or heavy runtime [10]. Cogent's uniqueness type system ensures exclusive ownership of mutable data, preventing aliasing and enabling direct updates without corrupting memory.

The uniqueness type system used in Cogent is quite sophisticated (see the Cogent paper [10] for a full description); here, we only focus on the precise conditions under which external C code imported into Cogent, maintains the memory safety guarantees that its uniqueness types provide.

Cogent's typing relation tracks a set of pointers w transitively accessible through a value v, called the *heap footprint*—note that w is a subset of the domain of the current heap. Written $v : \tau \ \langle w \rangle$, this judgement states that v has type τ and an associated footprint w. By annotating the relation in this way, the required non-aliasing requirements can be included in the typing rules.

For example, the rule for typing tuples is:

$$\frac{x : \tau_1 \ \langle w_x \rangle \quad y : \tau_2 \ \langle w_y \rangle \quad w_x \cap w_y = \emptyset}{(x, y) : \tau_1 \times \tau_2 \ \langle w_x \cup w_y \rangle}$$

The disjointness condition $w_x \cap w_y = \emptyset$ prevents internal aliasing in non-abstract structures, preserving the uniqueness guarantee. For abstract types implemented in C, however, internal aliasing may occur without breaking the uniqueness guarantee. This is permissible because the type system enforces uniqueness at the abstract interface level, not within the hidden implementation.

Let $f : \tau \to \rho$ be a function implemented in C and imported into Cogent. Let v be the input value such that $v : \tau \ \langle w_i \rangle$, h_i the initial heap (a partial map from pointers to values), and $f(v)$ the result such that $f(v) : \rho \ \langle w_o \rangle$ with final heap h_o. For f to respect the uniqueness invariants, the following *frame conditions* must hold:

Leak Freedom $\forall p.\ p \in w_i \wedge p \notin w_o \longrightarrow p \notin \mathrm{dom}(h_o)$, that is, any pointer in the input footprint that is not in the domain of the final heap must not appear in the initial heap. We use the equivalent yet more convenient formulation: $w_i \cap \mathrm{dom}(h_o) \subseteq w_o$.

Fresh Allocation $\forall p.\ p \notin w_i \wedge p \in w_o \longrightarrow p \notin \mathrm{dom}(h_i)$, that is, any pointer in the output footprint that is not in the input footprint must not alias with any pointer in the initial heap. We use the equivalent yet more convenient formulation: $w_o \cap \mathrm{dom}(h_i) \subseteq w_i$.

Inertia $\forall p.\ p \notin w_i \wedge p \notin w_o \longrightarrow h_i(p) = h_o(p)$, that is, any part of the heap not in the input or output footprint remains unchanged.

These conditions ensure that the function only relies on the memory it is permitted to access. Collectively, they are called the *frame conditions*, named after the frame problem in knowledge representation [8].

Manually discharging these proof obligations for external C code without the support of a suitable logic for reasoning about state is tedious [4]. The next sections will explain how these conditions can be verified using separation logic [14], which is specifically designed for reasoning about heap-manipulating programs.

3 Language Syntax and Operational Semantics

We introduce a simple imperative language $\mathcal{L}_{IMP}$ for which we will define our separation logic. We adopt an imperative language [1] that captures key features of low-level memory manipulation: heap allocation and deallocation as well as reads and writes from and into the heap. This language is mechanised in Rocq, and it was used to define a formal notion of memory safety. As such, it provides a solid foundation for our work. We extend the language to distinguish memory-related errors from other errors.

Block identifier	id	$\in$	$\mathcal{I}$
Variable	x	$\in$	Var
Naturals	n	$\in$	$\mathbb{N}$
Boolean	b	$\in$	$\mathbb{B} = \texttt{true} \mid \texttt{false}$
Pointer	p	$\in$	$\mathcal{P} = \mathcal{I} \times \mathbb{N}$
Value	v	$\in$	$\mathcal{V} = \mathbb{N} \cup \mathbb{B} \cup \mathcal{P} \cup \{\texttt{nil}\}$
Binary operator	$\oplus$	$::=$	$+ \mid - \mid \times \mid = \mid \leq \mid \wedge \mid \vee$
Expression	e	$\in$	$\mathcal{E} ::= x \mid n \mid b \mid \neg e \mid \texttt{offset}\, e \mid e \oplus e$
Command	c	$\in$	$\mathcal{C} ::= \texttt{skip} \mid x := e \mid \texttt{load}\, x\, e \mid \texttt{store}\, e_1\, e_2 \mid \texttt{alloc}\, x\, e \mid \texttt{free}\, e \mid \texttt{seq}\, c_1\, c_2 \mid \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \mid \texttt{while}\, e\, \texttt{do}\, c$

Fig. 1. Syntax

Figure 1 presents the syntax of the extended language and supporting components. It features values v that include booleans, naturals, and pointers. A pointer is a pair (id, n) where id is a block identifier drawn from a countably infinite set $\mathcal{I}$ of memory identifiers, and $n \in \mathbb{N}$ is an offset. The distinguished value `nil` represents the result of an ill-typed expression (e.g., $3 + \texttt{true}$) and is used to uniformly propagate errors. The language has standard expressions e that have no side effects on the state, including $\texttt{offset}\, e$, which returns the offset of a pointer value. It also features standard imperative commands c such as skip, local assignment, heap operations, and control flow constructs including loops. Figure 1 also presents the definition of states on which commands evaluate. States consist of a pair of components: a *local store* l, a finite partial map from variables to values, and a heap h, a finite partial map from pointers to values.

$$\text{Local store} \quad l \in \mathcal{L} = \text{Var} \rightharpoonup_{\text{fin}} \mathcal{V}$$

$$\boxed{\text{Expression evaluation } [\![-]\!]^e :: \mathcal{E} \to \mathcal{L} \to \mathcal{V}}$$

$$\begin{aligned}
[\![x]\!]^e\, l &= \begin{cases} l(x) & \text{if } x \in \operatorname{dom} l \\ \texttt{nil} & \text{otherwise} \end{cases} \qquad (x \in \text{Var}) \\
[\![v]\!]^e\, l &= v \qquad (x \in V) \\
[\![e_1 + e_2]\!]^e\, l &= \begin{cases} n_1 + n_2 & \text{if } [\![e_1]\!]^e\, l = n_1 \text{ and } [\![e_2]\!]^e\, l = n_2 \\ p +_{\text{p}} n & \text{if } [\![e_1]\!]^e\, l = p \text{ and } [\![e_2]\!]^e\, l = n \\ & \text{or if } [\![e_1]\!]^e\, l = n \text{ and } [\![e_2]\!]^e\, l = p \\ \texttt{nil} & \text{otherwise} \end{cases} \\
[\![e_1 - e_2]\!]^e\, l &= \begin{cases} n_1 - n_2 & \text{if } [\![e_1]\!]^e\, l = n_1 \text{ and } [\![e_2]\!]^e\, l = n_2 \\ (id, n_1 - n_2) & \text{if } [\![e_1]\!]^e\, l = (id, n_1),\ [\![e_2]\!]^e\, l = n_2, \text{and } n_2 \leq n_1 \\ \texttt{nil} & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 2. Evaluation of expressions (excerpt).

Now that we have presented the syntax, let's define an operational semantics for evaluating expressions and commands. Expression evaluation, $[\![e]\!]^e : \mathcal{S} \to \mathcal{V}$, depends only on the local store. Hence, pointers may not be dereferenced in an expression. Base values, such as booleans, naturals, and `nil`, evaluate to themselves. Arithmetic and boolean operations behave in the standard way. Addition and subtraction can additionally be used for performing pointer arithmetic on pointer offsets; we use the notation $(i, n_1) +_{\text{p}} n_2$ as shorthand for the pointer addition $(i, n_1 + n_2)$. Equality comparison is allowed on all values, with pointer equality comparing both the block identifier and offset. The $\leq$ operator applies only to naturals. Any operation applied to ill-typed values returns `nil`. Expression evaluation is standard, and an excerpt of the expression evaluation function is presented in Fig. 2.

$$\begin{array}{lll}
\text{Heap} & h & \in \mathcal{H} = \mathcal{P} \rightharpoonup_{\text{fin}} \mathcal{V} \\
\text{State} & s & \in \mathcal{S} = \mathcal{L} \times \mathcal{H} \\
\\
\text{Memory error} & m & \in \mathcal{M} ::= \texttt{invalidRead} \mid \texttt{invalidWrite} \mid \texttt{invalidFree} \\
\text{Error} & err & ::= m \mid \texttt{other} \\
\text{Result} & rs & \in \mathcal{R} ::= \texttt{done}\, s \mid \texttt{error}\, err \mid \texttt{notYet}
\end{array}$$

$$\boxed{\text{Bind operation bind} :: (\mathcal{C} \to \mathcal{S} \to \mathcal{R},\, \mathcal{R}) \to \mathcal{R}}$$

$$\begin{array}{ll}
\text{bind}(f, \texttt{error}\, err) & = \texttt{error}\, err \\
\text{bind}(f, \texttt{notYet}) & = \texttt{notYet} \\
\text{bind}(f, \texttt{done}\,(l,h)) & = \begin{cases} \texttt{done}\,(l',h') & \text{if } f(l,h) = \texttt{done}\,(l',h') \\ \texttt{error}\, err & \text{if } f(l,h) = \texttt{error}\, err \end{cases}
\end{array}$$

$$\boxed{\text{Command evaluation } [\![-]\!] :: \mathcal{C} \to \mathbb{N} \to \mathcal{S} \to \mathcal{R}}$$

$$\begin{array}{ll}
[\![c]\!]_0\, s & = \texttt{notYet} \\
[\![\texttt{skip}]\!]_{n+1}\, s & = \texttt{done}\, s \\
[\![x := e]\!]_{n+1}\, (l,h) & = \texttt{done}(l[x \mapsto [\![e]\!]^e\, l\,], h) \\
[\![\texttt{seq}\, c_1\, c_2]\!]_{n+1}\, s & = \text{bind}([\![c_2]\!]_n\,, [\![c_1]\!]_n\, s) \\
[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{n+1}\, (l,h) & = \text{if } [\![e]\!]^e\, l \text{ then } [\![c_1]\!]_n\, (l,h) \text{ else } [\![c_2]\!]_n\, (l,h) \\
[\![\texttt{while}\, e\, \texttt{do}\, c]\!]_{n+1}\, (l,h) & = \begin{cases} [\![\texttt{seq}\, c\, (\texttt{while}\, e\, \texttt{do}\, c)]\!]_n\, (l,h) & \text{if } [\![e]\!]^e\, l = \texttt{true} \\ \texttt{done}(l,h) & \text{if } [\![e]\!]^e\, l = \texttt{false} \\ \texttt{error}\, \texttt{other} & \text{otherwise} \end{cases} \\
[\![\texttt{load}\, x\, e]\!]_{n+1}\, (l,h) & = \begin{cases} \texttt{done}(l[x \mapsto v\,], h) & \text{if } [\![e]\!]^e\, l = p \text{ and } p \in \text{dom}\, h \\ \texttt{error}\, \texttt{invalidRead} & \text{if } [\![e]\!]^e\, l = p \text{ and } p \notin \text{dom}\, h \\ \texttt{error}\, \texttt{other} & \text{otherwise} \end{cases} \\
[\![\texttt{store}\, e_1\, e_2]\!]_{n+1}\, (l,h) & = \begin{cases} \texttt{done}(l, h[p \mapsto [\![e_2]\!]^e\, l\,]) & \text{if } [\![e_1]\!]^e\, l = p \text{ and} \\ & p \in \text{dom}\, h \\ \texttt{error}\, \texttt{invalidWrite} & \text{if } [\![e_1]\!]^e\, l = p \text{ and} \\ & p \notin \text{dom}\, h \\ \texttt{error}\, \texttt{other} & \text{otherwise} \end{cases} \\
[\![\texttt{alloc}\, x\, e]\!]_{n+1}\, (l,h) & = \begin{cases} \texttt{done}(l[x \mapsto (i,0)], & \text{if } [\![e]\!]^e\, l = k \text{ and} \\ \quad h[(i,j) \mapsto 0 \mid 0 \le j < k]) & i = \text{fresh}(\text{ids}(l,h)) \\ \texttt{error}\, \texttt{other} & \text{otherwise} \end{cases} \\
[\![\texttt{free}\, e]\!]_{n+1}\, (l,h) & = \begin{cases} \texttt{done}(l, h[(i,k) \mapsto \bot \mid k \in \mathbb{N}]) & \text{if } [\![e]\!]^e\, l = (i,0) \text{ and} \\ & \exists j.\ (i,j) \in \text{dom}\, h \\ \texttt{error}\, \texttt{invalidFree} & \text{if } [\![e]\!]^e\, l = (i,0) \text{ and} \\ & \forall j.\ (i,j) \notin \text{dom}\, h \\ \texttt{error}\, \texttt{other} & \text{otherwise} \end{cases}
\end{array}$$

Fig. 3. Evaluation of commands

The function $[\![-]\!] : \mathcal{C} \to \mathbb{N} \to \mathcal{S} \to \mathcal{R}$ defines big-step evaluation for commands. It evaluates a program c in a given state s, with a maximum of n execution steps, and produces a result. The result is either `done` s' with final state s', an error `error` err, which could be a memory-related error m or an `other` error, or a timeout result `notYet` if the execution limit n is reached before completion. Note that the number of steps n provided as fuel is a formalisation detail that is only necessary to conveniently formalise this semantics as a total function in Rocq.

The language enforces safety by raising errors that immediately halt execution, preventing undefined or unpredictable behaviour due to memory misuse. The command evaluation function is presented in Fig. 3. Unlike the original language semantics [1], our semantics distinguishes memory safety errors from other runtime errors. This distinction is necessary to define a separation logic validity triple that is sensitive to memory safety violations. Specifically, we extend the semantics to support the following memory safety errors: `invalidRead`, reading from an invalid memory location, `invalidWrite`, writing outside the bounds of a block or to otherwise invalid memory, `invalidFree`, freeing an invalid pointer.

The evaluation follows the standard imperative semantics for `skip`, `seq` c_1 c_2, $x := e$, `if` e `then` c_1 `else` c_2, and `while` e `do` c. The `load` x e heap lookup command first evaluates e to a pointer and then attempts to read from that location in the heap. If e does not evaluate to a pointer, evaluation results in `error other`. If the pointer is invalid, either because the block identifier is not allocated or the offset is out of bounds, the result is an `invalidRead` error. Similarly, the heap mutation command, `store` e_1 e_2, requires e_1 to evaluate to a valid pointer. If not, evaluation results in an `invalidWrite` error. The allocation command, `alloc` x e, evaluates e to a natural k and then produces a fresh block identifier that is not currently in use. A new block of size k is added to the heap, with all cells initialised to 0. This initialisation is important for ensuring that allocation does not leak information present in blocks that have been previously freed. The resulting state binds x to a pointer that refers to the first cell in the new block. Block identifiers in this language are immutable once assigned, making it impossible to fabricate a pointer to an allocated block. The heap deallocation command, `free` e, evaluates e to a pointer. If the pointer is valid, the result is a heap where all cells associated with the corresponding block identifier are removed. If the pointer is invalid, evaluation results in an `invalidFree` error.

To summarise, our semantics extends an existing imperative language with manual memory management [1] that was formalised in Rocq, by explicitly distinguishing memory safety violations from other errors. In the next section, we present a separation logic for reasoning about this language and prove its soundness with respect to the operational semantics described here.

4 Separation Logic

Now that we have defined the language and its operational semantics, we are ready to develop a separation logic for reasoning about programs in this language.

In standard separation logic [14], a triple (P, c, Q) is considered valid if whenever command c evaluates from a state satisfying precondition P, if evaluation terminates, it does so successfully in a state satisfying postcondition Q. In particular, this notion of *strong validity* $\{P\}\, c\, \{Q\}_{\mathtt{S}}$ holds exactly when $\forall s\, k.\, P\, s \to (\llbracket c \rrbracket_k\, s \neq \mathtt{error}\, err) \wedge (\forall s'.\, \llbracket c \rrbracket_k\, s = \mathtt{done}\, s' \to Q\, s')$. That is, for any state s satisfying P, the execution of the program c in s will not trigger an error and, if the execution terminates, it will do so in a state satisfying Q.

As previously observed in the literature [1], strong validity makes it difficult to distinguish reasoning about the validity of *heap-related behaviour*. For example, the triple $\{\mathtt{emp}\}\, \mathtt{free}\, \mathrm{true}\, \{\mathtt{emp}\}_{\mathtt{S}}$ (where $\mathtt{emp}$ denotes the predicate that holds on states with an empty heap) is invalid, even though it accurately describes the behaviour of the heap: when the program runs on an empty heap, we obtain an error. However, when the program stops, the heap remains empty.

To address this issue, a more permissive variant of separation logic allowing certain classes of errors was introduced [1], where *weak validity*, written $\{P\}\, \mathtt{c}\, \{Q\}_{\mathtt{W}}$, holds if evaluating c from a state satisfying P either diverges, triggers an error, or terminates in a state satisfying Q. Formally, $\{P\}\, \mathtt{c}\, \{Q\}_{\mathtt{W}}$ holds iff $\forall s\, k.\, P\, s \to (\forall s'.\, \llbracket c \rrbracket_k\, s = \mathtt{done}\, s' \to Q\ s')$. Note that, under weak validity, $\{\mathtt{emp}\}\, \mathtt{free}\, \mathrm{true}\, \{\mathtt{emp}\}_{\mathtt{S}}$ holds.

However, this relaxed notion of validity introduces two major drawbacks. First, it invalidates the soundness of the *frame rule*[1], which is an essential component of separation logic, enabling the extension of local reasoning to parts of the heap that the command has not modified. Second, it is insensitive to the particular error that has occurred. In this way, it undermines efforts to characterise memory-safe programs: for instance, $\{\mathtt{emp}\}\, \mathtt{load}\, \mathtt{x}\, \mathtt{e}\, \{\mathtt{emp}\}_{\mathtt{W}}$ is deemed valid, even though the command may raise a memory safety error.

To remedy this, we introduce a third notion of validity that explicitly *distinguishes memory safety violations* from other kinds of errors. A *memory-safe validity* triple, written $\{P\}\, \mathtt{c}\, \{Q\}_{\mathtt{M}}$, holds if evaluating c from a state satisfying P does not result in a memory safety error and if it terminates successfully, the resulting state satisfies Q. In particular, memory-safe validity $\{P\}\, \mathtt{c}\, \{Q\}_{\mathtt{M}}$ holds iff $\forall s\, k.\, P\, s \to (\llbracket c \rrbracket_k\, s \neq \mathtt{error}\, m) \wedge (\forall s'.\, \llbracket c \rrbracket_k\, s = \mathtt{done}\, s' \to Q\, s')$. All other behaviour, including divergence or non-memory-related errors, is permitted.

This definition achieves the desired balance: it rejects $\{\mathtt{emp}\}\, \mathtt{load}\, \mathtt{x}\, \mathtt{e}\, \{\mathtt{emp}\}_{\mathtt{M}}$ as invalid reads violate memory safety; on the other hand, since the evaluation results in a non-memory-related error, $\{\mathtt{emp}\}\, \mathtt{free}\, \mathrm{true}\, \{\mathtt{emp}\}_{\mathtt{S}}$ is a valid triple. An added benefit of this notion is that it turns out that it preserves the soundness of the frame rule, enabling modular reasoning about heap-manipulating programs using the standard frame rule. Since this separation logic triple is the one we focus on in this paper, we simply refer to it as $\{P\}\, \mathtt{c}\, \{Q\}$ in the rest of the paper.

Before introducing the assertion logic used to describe program predicates, we first present the basic notation that underpins our logic (see Fig. 4). We write set disjointness as $\#$; this holds when two sets have no elements in common. Heap

[1] A detailed explanation of this issue can be found at the end of this section.

Map notation		
$f_1 \cup f_2$	$\equiv \{(k,v) \mid (k,v) \in f_1 \land k \notin \text{dom}(f_2) \lor (k,v) \in f_2\}$	(map union)
$f \upharpoonright S$	$\equiv \{(k,v) \in f \mid k \in S\}$	(map restriction)
Local store, heap, and state notation		
$(l_1, h_1) \cup (l_2, h_2)$	$\equiv (l_1 \cup l_2,\ h_1 \cup h_2)$	(state union)
$h_1 \perp h_2$	$\equiv \text{dom}\,(h_1) \cap \text{dom}\,(h_2) = \emptyset$	(heap disjointness)
$\text{blocks}(l, h)$	$\equiv \{i \in \mathcal{I} \mid \exists n, (i,n) \in \text{dom}(h)\}$	(live block identifiers)
$\text{ids}(l, h)$	$\equiv \text{blocks}(l,h) \cup \{i \mid \exists x\, n.\ l(x) = (i,n)\} \cup \{i \mid \exists p\, n.\ h(p) = (i,n)\}$	(all identifiers)
$\text{vars}(l, h)$	$\equiv \text{dom}(l)$	(state vars)
$\text{vars}(c)$	$\equiv$ local variables of program c	(command vars)
Nominal notation		
$\pi \cdot (b, k)$	$\equiv (\pi(b), k)$	(permute pointer)

Fig. 4. Basic notation

disjointness, written $\perp$, holds when the domains of two heaps, treated as partial maps from addresses to values, are disjoint. The union operator $\cup$ is overloaded in our setting: For sets, it denotes the usual set-theoretic union. Map union, also $\cup$, is the union of the key-value pairs from the two maps, choosing the first map's values for pairs with the same key. For states, which consist of both a local store and a heap, union is defined pointwise by applying the map-union operator to both components. We write map restriction as $f \upharpoonright S$, which denotes the restriction of a map f to a set S: this is a new map containing only the entries in f whose keys belong to S. With this notation in place, we can now define the predicates of our separation logic.

Preconditions and postconditions are logical predicates over states. Separation logic predicates extend usual logical predicates with operators that enable local reasoning about the heap. It has the following additional predicates.

Empty heap `emp` holds when the heap is empty, that is, when $\text{dom}(h) = \emptyset$.
Singleton heap $(p \mapsto v)$ holds iff the heap contains exactly one binding $p \mapsto v$. That is, when $\text{dom}(h) = \{p\} \land h(p) = v$.
Separating conjunction $P * Q$ holds iff the heap splits into disjoint parts satisfying P and Q respectively (with a shared store).
That is, when $\exists h_1\, h_2.\ h_1 \perp h_2 \land h = h_1 \cup h_2 \land P(l, h_1) \land Q(l, h_2)$. Separating conjunction is associative, commutative, and has `emp` as unit:
$P * (Q * R) \leftrightarrow (P * Q) * R, \quad P * Q \leftrightarrow Q * P, \quad P * \texttt{emp} \leftrightarrow P.$

The separation logic rules are presented in Fig. 5. We define a separation logic rule for each command in our language. We also include the standard consequence rule and the central frame rule. We verified the soundness of these separation logic triples with respect to the operational semantics.

The rules for assignment, `skip`, `if`, `seq`, and `while` adhere to the standard Hoare logic conventions. More interesting are rules for primitive memory opera-

$$\boxed{\text{Separation Logic Memory-Safe Validity } \{P\}\, \mathtt{c}\, \{Q\}}$$

$$\{P\}\, \mathtt{skip}\, \{P\} \qquad\qquad \{P[e/x]\}\, \mathtt{x} := \mathtt{e}\, \{P\}$$

$$\{\lambda s.\ [\![e]\!]^e s = p\ \wedge (p \mapsto v)s\}\, \mathtt{load\,x\,e}\, \{\lambda s.\ (x = v)s \wedge (p \mapsto v)s\}$$

$$\{\lambda s.\ [\![e_1]\!]^e\, s = p\ \wedge (p \mapsto \_)s \wedge [\![e_2]\!]^e\, s = v\}\, \mathtt{store\,e_1\,e_2}\, \{p \mapsto v\}$$

$$\{\lambda s.\ [\![e]\!]^e s = k \wedge \mathtt{emp}\, s\}\, \mathtt{alloc\,x\,e}\, \{\lambda s.\ \exists i.\ [\![x]\!]^e s = (i, 0)\ \wedge [((i, j) \mapsto 0) \mid 0 \leq j < k](s)\}$$

$$\{\lambda s.\ [\![e]\!]^e s = (i, 0)\ \wedge ((i, 0) \mapsto \_)s\}\, \mathtt{free\,e}\, \{\mathtt{emp}\} \qquad \frac{\{P\}\, \mathtt{c_1}\, \{Q\} \qquad \{Q\}\, \mathtt{c_2}\, \{R\}}{\{P\}\, \mathtt{seq\,c_1\,c_2}\, \{R\}}$$

$$\frac{\{\lambda s.\ [\![e]\!]^e s \wedge Ps\}\, \mathtt{c_1}\, \{Q1\} \qquad \{\lambda s.\ \neg[\![e]\!]^e s \wedge Ps\}\, \mathtt{c_2}\, \{Q2\}}{\{P\}\, \mathtt{if\,e\,then\,c_1\,else\,c_2}\, \{\lambda s.\ Q1s \vee Q2s\}}$$

$$\frac{\{\lambda s.\ [\![e]\!]^e s \wedge Ps\}\, \mathtt{c}\, \{P\}}{\{P\}\, \mathtt{while\,e\,do\,c}\, \{\lambda s.\ \neg[\![e]\!]^e s \wedge Ps\}}$$

$$\frac{\forall s.\ P's \rightarrow Ps \qquad \{P\}\, \mathtt{c}\, \{Q\} \qquad \forall s.\ Qs \rightarrow Q's}{\{P'\}\, \mathtt{c}\, \{Q'\}} \qquad \frac{\{P\}\, \mathtt{c}\, \{Q\}}{\{P * R\}\, \mathtt{c}\, \{Q * R\}}\ \text{Frame}$$

Fig. 5. Separation logic inference rules. Note that Frame holds under the condition that the variables referred to in R are independent of the variables modified by c.

tions. We begin with the `load` rule. To `load` $x\, e$, since pointers cannot be dereferenced within an expression `e`, the expression `e` must evaluate to a pointer p referring to some value v. After loading from p, the variable x now stores v, and the heap at p remains unchanged. For the `store` rule, given a program `store` $e_1\, e_2$, the precondition requires that the expression $\mathtt{e_1}$ evaluates to a pointer p and the heap consists solely of this pointer. The value v resulting from evaluating $\mathtt{e_2}$ is then stored at the pointer location p in the output heap, and the postcondition reflects this update.

The allocation rule requires an empty heap. While the precondition for this rule is not necessary for the correctness of `alloc` $x\, e$; it helps strengthen the postcondition, having it refer to the size of the allocated memory block k. The postcondition ensures that x stores the new pointer and the heap holds k contiguous cells initialised to 0.

In the rule for `free`, the precondition states that the heap contains exactly one cell, the one to be freed. After execution, the heap is empty. Regarding structural rules, we include the rule of consequence, which allows strengthening the precondition and weakening the postcondition. This rule is justified purely by logical entailment and does not depend on the operational semantics of commands. Last but not least, we prove the soundness of separation logic's vital frame rule, which enables local reasoning. We will elaborate further on this rule below.

The Frame Rule. Separation logic features the *frame rule*, which offers a principled solution to the frame problem. It allows an independent assertion R to be appended to both the pre- and postconditions of a triple using a separating conjunction.

$$\frac{\{P\}\, c\, \{Q\}}{\{P * R\}\, c\, \{Q * R\}}\ \text{(Frame)}$$

where variables modified by c are independent of the validity of R

The purpose of the frame rule is to facilitate *local reasoning.* If a program c safely executes on a minimal state described by P and results in a state satisfying Q, then it also safely executes in any larger state and does not invalidate any predicates R that are true of the independent parts of the larger state. Hence, the frame rule guarantees that the execution of c is unaffected by and does not interfere with these unrelated parts of the state.

As discussed in the definition of weak validity triples, however, allowing all types of errors undermines the soundness of the frame rule. For example, consider the triple $\{\texttt{emp}\}\, \texttt{load}\, \texttt{x}\, \texttt{e}\, \{x = 0\}_{\texttt{W}}$ where $\forall s. [\![e]\!]^e\, s = y$ noted in prior work [1]. This triple satisfies weak validity only because all types of errors (including memory errors) are allowed. Framing such a triple with $R = y \mapsto 1$ yields the triple $\{y \mapsto 1\}\, \texttt{load}\, \texttt{x}\, \texttt{e}\, \{x = 0 \wedge y \mapsto 1\}_{\texttt{W}}$, which is invalid. Since the load from an uninitialised address would now succeed due to the presence of $(y \mapsto 1)$ in the combined state, but the postcondition is false. Thus, the program's faulty behaviour is masked by the extended heap, and the faults are not propagated, violating the principle of local reasoning. Prior work [1] addressed this by introducing the *isolating conjunction* $P \rhd Q$, defined as:

$$(P \rhd Q)(l, h) \equiv \text{ids}(l, h_1) \# \text{blocks}(l, h_2) \wedge h = h_1 \cup h_2 \wedge P(l, h_1) \wedge Q(l, h_2).$$

The isolating conjunction prevents repairing a dangling pointer through heap union, when appending an independent assertion R. In this way, it avoids turning an erroneous execution into a successful one when extending the state. In contrast, our approach avoids these restrictions. Since our memory-safe validity triple explicitly distinguishes memory safety violations from other errors, programs that suffer from such errors are excluded by design. This allows us to safely apply the standard frame rule without introducing the isolating conjunction.

This section provided an overview of the standard separation logic features. We showed how we adapted the definition of triple validity [1] to prevent memory errors. Additionally, to facilitate reasoning about program specifications, we presented suitable separation logic rules for our language and proved their soundness with respect to the operational semantics. The following section outlines how we express the frame conditions imposed by uniqueness types, as presented in Sect. 2, into our separation logic to verify that a safe program respects the uniqueness imposed safety invariants.

5 Uniqueness as Separation

As previously mentioned, we aim to formally verify whether an imperative program meets the memory safety invariants from a uniqueness type system. To reduce our formal verification effort, we express these invariants within a framework suitable for reasoning about programs that access and mutate memory.

In this section, we represent the frame conditions (Sect. 2) using separation logic through the triple FC Triple. Additionally, we present Theorems 2, 3 and 4 showing that FC Triple implies each of the frame conditions.

It has been shown that expressing and manually proving the frame conditions requires considerable effort [4]. To mitigate this challenge, we begin by representing the frame conditions as a single separation logic triple:

Definition 1 (FC Triple). *Given a program c with an input set of pointers w_i and an output set of pointers w_o, we express the frame conditions as the single triple:*

$$\left\{\mathop{\ast}_{\ell\in w_i}\exists v.\ \ell \mapsto v\right\}\ c\ \{\mathop{\ast}_{\ell\in w_o}\exists v.\ \ell \mapsto v\}_{\mathtt{M}} \qquad \text{(FC Triple)}$$

fc triple describes w_i and w_o as the input and output footprints of the command c. It requires that all locations in w_i are valid pointers in the initial state. Additionally, it asserts that all locations in the output footprint are valid pointers in the final state and guarantees the integrity of non-overlapping memory.

To better understand the theorems that assert that fc triple implies the frame conditions, it is useful to recall some basic notions of nominal set theory, which is a mathematical framework to formally reason about names, binding and alpha-equivalence.

Let $\pi : \mathcal{I} \to \mathcal{I}$ be a bijective function on the set of block identifiers $\mathcal{I}$. In nominal, the renaming operation $\pi \cdot (-)$ is a function on structures containing block identifiers, that renames every block identifier in that structure using the function π. For example, considering the state s, $\pi \cdot s$ has the effect of changing all block identifiers contained in the state using the function π.

The support of π is the set of names that π actually change, i.e., the smallest set of names such that outside of it, π behaves as the identity. Formally, this is represented as $\operatorname{supp}\pi = \{x \in \mathcal{I} \mid \pi(x) \neq x\}$. Lastly, when we state that π fixes a set A, we mean that for all elements in A, π maps them to themselves.

The proofs of our theorems rely crucially on the following two results. The first lemma asserts that given a set of pointers w, if we can split the heap into $|w|$ different subheaps, where the domain of each subheap consists of a single element of w, we can conclude that the domain of the heap is precisely the heap footprint w, and vice versa.

Lemma 1 (Big-Sep as Domain Equation)

$$(\mathop{\ast}_{\ell\in w}\exists v.\ \ell \mapsto v)\ (ls, h) \longleftrightarrow \operatorname{dom}(h) = w$$

The Frame OK theorem, as introduced by Azevedo de Amorim et al. [1], states that if a program terminates successfully, then we can extend its initial state without affecting its execution, up to some permutation that accounts for different choices of fresh names to guarantee no clashes between identifiers.

Theorem 1 (Frame OK). *Let c be a command, and s_1, s_1', and s_2 be states. Suppose that $[\![c]\!]_n\, s_1 = s_1'$, $\mathsf{vars}\,(c) \subseteq \mathsf{vars}\,(s_1)$, and $\mathsf{blocks}\,(s_1) \ \#\ \mathsf{blocks}\,(s_2)$. Then, there exists a permutation π such that $[\![c]\!]_n\,(s_1 \cup s_2) = \pi \cdot s_1' \cup s_2$ and $\mathsf{blocks}\,(\pi \cdot s_1') \ \#\ \mathsf{blocks}\,(s_2)$.*

Having reviewed some nominal set basic concepts and presented important previous results, let us now turn to the main theorems. The first theorem states that if FC Triple holds for a program c with an input set of pointers w_i, and an output set of pointers w_o, then we know that any pointer initially available that is not returned in the final set, must have been freed – since freeing a pointer is the only way to remove pointers from the domain of the heap.

Theorem 2 (FC Triple Implies Leak Freedom). *Let c be a program, w_i the set of input pointers, and w_o the set of output pointers. Let s be the input state, s' and s_1' output states, and π a permutation.*[2] *Additionally, let $s = (ls, h)$, $s' = (ls', h')$, and $s_1' = (ls_1', h_1')$.*

Then, we have the following theorem:

$$\begin{array}{l}
\textit{assuming} \\
\quad \mathsf{vars}\, c \subseteq \mathsf{vars}(s \restriction w_i) \\
\quad w_i \subseteq \mathrm{dom}(h) \\
\quad [\![c]\!]_n\,(s \restriction w_i) = \mathtt{done}\,(\pi_1 \cdot s_1') \\
\quad \mathsf{blocks}(h \restriction w_i) \ \#\ \mathsf{blocks}(h \restriction \overline{w_i}) \\
\quad [\![c]\!]_n\, s = \mathtt{done}\,(\pi \cdot s') \\
\quad \left\{ \ast_{\ell \in w_i} \exists v.\ \ell \mapsto v \right\}\ c\ \left\{ \ast_{\ell \in w_o} \exists v.\ \ell \mapsto v \right\}_{\mathtt{M}} \\
\textit{then} \\
\quad w_i \cap \mathrm{dom}(h') \subseteq w_o.
\end{array}$$

The first premise, $\mathsf{vars}\, c \subseteq \mathsf{vars}(s \restriction w_i)$, guarantees that all the variables needed to run c are already defined in the state $s \restriction w_i$. This implies that their values remain unchanged when we extend that initial state with $s \restriction \overline{w_i}$. The second premise, $w_i \subseteq \mathrm{dom}(h)$, ensures that we can indeed split the heap h, and consequently the state s, into two distinct parts: $h \restriction w_i$ and $h \restriction \overline{w_i}$.

The fourth premise prevents the splitting of allocated memory blocks. The fifth premise assumes that the command c executes successfully on the state s, allowing us to refer to the final heap in the theorem's conclusion. Permutations are needed to facilitate expressing properties on states independently of the new

[2] Note that, instead of exposing the nominal monads used in the Rocq proofs, we will just present proofs with permutations.

block identifiers potentially allocated by the command. Finally, the theorem's conclusion states that leak freedom (Sect. 2) holds on the final heap h'.

The proof of the Theorem 2 relies crucially on the fact that the result of $[\![c]\!]_n\,(s \restriction w_i)$ does not change when we extend the initial state with $h \restriction \overline{w_i}$, aside from some permutation that we pick to be π_1.

Additionally, the proof relies on the conditions $\mathrm{supp}(\pi_1) \# \mathrm{ids}(s \restriction \overline{w_i}) \cup w_o$ and $\mathrm{supp}(\pi) \# w_i$. This means that π_1 fixes $\mathrm{ids}(s \restriction \overline{w_i}) \cup w_o$ and π fixes w_i. Moreover, the proof uses the fact that $\mathrm{dom}(h \restriction w_i) = w_i$ (true by construction), and $\mathrm{dom}(\pi_1 \cdot h'_1) = w_o$ (deduced by Lemma 1).

The second theorem states that if FC Triple holds for a program c with an input set of pointers w_i, and an output set of pointers w_o, then we know that any pointer in the final set of pointers that is not in the initial footprint, must have been allocated – which implies that the new pointer is fresh with respect to the initial state.

Theorem 3 (FC Triple Implies Fresh Allocation).
Let c be a program, w_i the set of input pointers, and w_o the set of output pointers. Let s be the input state, s'_1 an output state, and π_1 a permutation. Additionally, let $s = (ls, h)$, and $s'_1 = (ls'_1, h'_1)$.

If

$$\mathrm{vars}\, c \subseteq \mathrm{vars}(s \restriction w_i)$$
$$w_i \subseteq \mathrm{dom}(h)$$
$$\mathrm{blocks}\,(h \restriction w_i) \# \mathrm{blocks}\,(h \restriction \overline{w_i})$$
$$[\![c]\!]_n\,(s \restriction w_i) = \mathtt{done}\,(\pi_1 \cdot s'_1)$$
$$\left\{\ast_{\ell \in w_i} \exists v.\ \ell \mapsto v\right\}\ c\ \left\{\ast_{\ell \in w_o} \exists v.\ \ell \mapsto v\right\}_{\mathtt{M}}$$

then

$$w_o \cap \mathrm{dom}(h) \subseteq w_i.$$

Note that, in this case, the Theorem 3 does not require the assumption $[\![c]\!]_n\, s = \mathtt{done}\,(\pi \cdot s')$. Instead, we derive this information throughout the proof from hypotheses three and four, along with the theorem Frame OK.

The proof of the theorem relies on the condition $\mathrm{supp}\,(\pi_1) \# \mathrm{ids}(s \restriction \overline{w_i}) \cup w_o$. This means that π_1 fixes $\mathrm{ids}(s \restriction \overline{w_i}) \cup w_o$. Additionally, the proof uses the fact that $\mathrm{dom}(h \restriction w_i) = w_i$ (which holds true by construction) and $\mathrm{dom}(\pi_1 \cdot h'_1) = w_o$ (deduced from Lemma 1). Moreover, the proof uses $\mathrm{dom}(\pi_1 \cdot h'_1) \perp \mathrm{dom}(h \restriction \overline{w_i})$ (inferred from Theorem 1) that leads to the fact that $w_o \# \mathrm{dom}(h \restriction \overline{w_i})$.

The third and final theorem, shows that if FC Triple holds for a program c with an input set of pointers w_i, and an output set of pointers w_o, we can conclude that the program does not modify any pointer outside of the specified input and output footprints.

Theorem 4 (FC Triple Implies Inertia).
Let c be a program, w_i the set of input pointers, and w_o the set of output pointers. Let s be the input state, s' and s'_1 output states, and π a permutation. Additionally, let $s = (ls, h)$, $s' = (ls', h')$, and $s'_1 = (ls'_1, h'_1)$.

Then, we have the following theorem:

$$\begin{array}{l}
\textit{assuming} \\
\quad \mathrm{vars}\, c \subseteq \mathrm{vars}(s \restriction w_i) \\
\quad w_i \subseteq \mathrm{dom}(h) \\
\quad \mathrm{blocks}\,(h \restriction w_i) \mathbin{\#} \mathrm{blocks}\,(h \restriction \overline{w_i}) \\
\quad [\![c]\!]_n\,(s \restriction w_i) = \mathtt{done}\,(\pi_1 \cdot s'_1) \\
\quad [\![c]\!]_n\, s = \mathtt{done}\,(\pi \cdot s') \\
\quad \left\{ \ast_{\ell \in w_i} \exists v.\ \ell \mapsto v \right\}\ c\ \left\{ \ast_{\ell \in w_o} \exists v.\ \ell \mapsto v \right\}_{\mathtt{M}} \\
\textit{then} \\
\quad \forall p.\ p \notin w_o \cup w_i \rightarrow h(p) = h'(p)
\end{array}$$

Similar to the proof for Theorem 2 and 3, this proof relies on π_1 fixing $\mathrm{ids}(s \restriction \overline{w_i}) \cup w_o$. Additionally, it uses the fact that $\mathrm{dom}(h \restriction w_i) = w_i$ (holds by construction) and $\mathrm{dom}(\pi_1 \cdot h'_1) = w_o$ (deduced from Lemma 1).

Based on Theorems 2, 3, and 4 we can conclude that to prove that a safe program complies with the memory safety invariants from uniqueness types, specifically the frame conditions, it is enough to show that the program satisfies FC TRIPLE . We now have a framework suitable for reasoning about the safety invariants from uniqueness types, namely the separation logic presented in Sect. 4, and the exact triple we need to discharge for a program intended for integration into the uniquenessly typed program.

6 Examples: Discharging Frame Conditions

For convenience, we define a validity triple on expressions $\{P\}\,\mathtt{e}\,\{\lambda v.\ Q\,v\}_e$. Given an expression e, a precondition on states P and a postcondition predicate on values Q, $\{P\}\,\mathtt{e}\,\{\lambda v.\ Q\,v\}_e$ holds iff $\forall (l,h).\, P\,(l,h) \rightarrow (\forall e[\![e]\!]^e\, l = v \rightarrow Q\,v)$.

Figure 6 has some examples of derived separation logic rules for single commands, maintaining the frame conditions separation logic triple under specific constraints on the input and output pointer sets that represent the heap footprint. These demonstrate that one can discharge the frame condition separation logic triples, which in turn imply that the uniqueness type invariants are satisfied (as per the theorems in Sect. 5). We verified these examples in Rocq. These derived rules can be directly used to verify larger programs when each command within the larger program respects the uniqueness constraints. In complex cases, when the entire program respects the separation logic frame conditions triple, but single commands within the program do not respect these conditions, such programs can still be verified manually. Even then, one can rely on the general separation logic rules to aid with the manual proofs.

$$\frac{w_i = w_o}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{skip}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{w_i = w_o}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{x} := \mathtt{e}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{w_i = w_o \qquad \{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{e}\ \{\lambda v.\ v \in \mathcal{P} \to \mathit{fst}\ v \in w_i\}_e}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{load\,x\,e}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{w_i = w_o \qquad \{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{e}_1\ \{\lambda v.\ v \in \mathcal{P} \to \mathit{fst}\ v \in w_i\}_e}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{store\,e_1\,e_2}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{e}\ \{\lambda (p,n).\ (p,n) \in \mathcal{P} \to (p \in w_i \wedge w_o = w_i \setminus \{p\} \wedge p \notin \mathit{fst}\,\grave{}\ w_o\}_e}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{free\,e}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{e}\ \{\lambda v.\ v \notin \mathcal{P} \to w_o = w_i\}_e}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{free\,e}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{c}_1\ \{\ast_{\ell\in w'}\exists v.\ \ell \mapsto v\} \qquad \{\ast_{\ell\in w'}\exists v.\ \ell \mapsto v\}\ \mathtt{c}_2\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{seq\,c_1\,c_2}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\}}$$

$$\frac{\begin{array}{c}\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{e}\ \{\lambda v.\ v \in \mathbb{B}\}_e \\ \{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{c}\ \{\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v\} \qquad \{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{c}'\ \{\ast_{\ell\in w_{o'}}\exists v.\ \ell \mapsto v\}\end{array}}{\{\ast_{\ell\in w_i}\exists v.\ \ell \mapsto v\}\ \mathtt{if\,e\,then\,c\,else\,c'}\ \{\lambda s.\ (\ast_{\ell\in w_o}\exists v.\ \ell \mapsto v) s \vee (\ast_{\ell\in w_{o'}}\exists v.\ \ell \mapsto v) s\}}$$

Fig. 6. Derived uniqueness separation logic triples. Note that $f\,\grave{}\,S$ is used to denote the image of a function f on a set S, and $S_1 \setminus S_2$ is used to denote set difference.

7 Related Work

The foundations of this work rest on three separate areas: separation logic, uniqueness and ownership types, and verified interoperability through foreign function interfaces.

Separation Logic. Separation logic [11,12,14] extends Hoare logic to enable local reasoning about heaps and pointers. Its key feature, the frame rule, ensures that the mutation of one part of the heap does not invalidate assertions about unmutated parts of the heap. Particularly relevant to the present work is the use of separation logic to formally define memory safety [1] as well as the RustBelt project [6,7].

Azevedo de Amorim et al. extend separation logic with a memory-safe variant of the frame rule accommodating errors [1]. While this work accounts for a weak form of validity that holds for programs that potentially return an error, it requires a stronger separation between the states resulting in a non-standard definition of separating conjunction. In contrast, we introduce a notion of validity that precludes memory errors while still allowing other types of errors and prove the soundness of the frame rule using the traditional notion of separating conjunction. Moreover, we verify the soundness of an entire inference system, facilitating more extensive reasoning.

The RustBelt project aims to ensure that Rust's core safety properties are preserved even for programs incorporating *unsafe* code, which can bypass Rust memory safety features. RustBelt provides a formal semantic model for a significant subset of Rust. This model provides the verification conditions necessary to guarantee that a program with unsafe code preserves Rust's type system invariants. RustBelt uses Iris, an advanced separation logic, to prove that specific unsafe code is sound, by associating it with logical invariants that mimic Rust's safety guarantees. Unlike RustBelt, the present research focuses on expressing the safety invariants within separation logic, rather than embedding a semantic interpretation of the type system into separation logic.

Uniqueness Types. Uniqueness types [17] track exclusive access to data, ensuring that unaliased data remains unaliased. We focus on the memory safety guarantees induced by uniqueness types, especially the frame invariants they establish. Clean [2], Idris [3], and Cogent [10] are some of the languages incorporating uniqueness systems. In particular, our research builds on Cogent's type system and its approach to memory safety.

FFI Verification. Significant research on FFIs aims to ensure that external code adheres to the safety requirements of the host language.

For example, Patterson et al. [13] present a framework for reasoning about cross-language correctness at the compiled level. The framework uses logical relations to model the semantics of the source language type system. This allows for identifying type equivalences across languages, facilitating safe data exchange between them and reasoning about the program's behaviour after compilation.

VeriFFI is a project that establishes a verified FFI between Rocq and C. It ensures that C functions comply with Rocq programs specifications. This is achieved by translating Rocq types and models into specifications compatible with the Verified Software Toolchain (VST). Then, developers can prove that the corresponding C implementations satisfy these specifications, guaranteeing type safety.

Cheung et al. manually verified that certain C components align with Cogent's type system invariants, including the frame conditions. They abstracted a monadic C embedding in Isabelle/HOL into a functional Cogent embedding within the same assistant. While their work successfully verified safety invariants of C code, it also showed the significant effort required, especially when using a framework not designed to facilitate pointer reasoning.

While these approaches to FFI verification are more general and often require comprehensive models of both the host and the guest languages, our approach focuses on expressing the invariants of a uniqueness type system within a version of separation logic suitable for reasoning on safe imperative languages.

8 Conclusion and Future Work

This paper describes the obligations that a programming language with uniqueness types, specifically Cogent, imposes on references and heaps to fully verify a program written in Cogent incorporating C code. We show how these frame obligations can be expressed in separation logic. Furthermore, we formally prove, using Rocq, that the proposed formulation in separation logic implies the frame obligations of the uniqueness-type system.

In the process of formally verifying programs with uniqueness types that feature a bypass mechanism, it is typical to assume the memory safety obligations on the external code. This practice leads to a verification that, while potentially encompassing safety, remains incomplete. Our formulation of the frame obligations derived from Cogent's uniqueness-type system into separation logic FC Triple, marks progress toward fully discharging these conditions.

We also aim to explore proof composition: given two programs c_1 and c_2 that satisfy the frame conditions expressed in separation logic, does it follow that the program $c_1; c_2$ also satisfy the frame conditions? We suspect that proof composition requires a weaker version of our FC Triple proposed.

Finally, we believe that this formulation of the frame conditions helps solidify the connection between uniqueness types and separation logic. Type systems and program logics are both tools for formal reasoning, and we run the risk of reinvention if we do not realise the connections between them.

References

1. Azevedo de Amorim, A., Hriţcu, C., Pierce, B.C.: The meaning of memory safety. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 79–105. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_4
2. Barendsen, E., Smetsers, S.: Conventional and uniqueness typing in graph rewrite systems. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 41–51. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57529-4_42
3. Brady, E.C.: Type-driven development of concurrent communicating systems. Comput. Sci. **18**(3) (2017). https://doi.org/10.7494/CSCI.2017.18.3.1413
4. Cheung, L., O'Connor, L., Rizkallah, C.: Overcoming restraint: composing verification of foreign functions with cogent. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2022, pp. 13–26. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503686
5. Herhut, S.B.S.S., Penczek, F., Grelck, C., Shinkarov, A., Viessmann, H.N.: Single assignment C tutorial (2025)
6. Jung, R.: Understanding and evolving the Rust programming language. Ph.D. thesis, Universität des Saarlandes (2020). https://doi.org/10.22028/D291-31946
7. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158154
8. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press (1969)

9. O'Connor, L., et al.: Refinement through restraint: bringing down the cost of verification (2016)
10. O'Connor, L., et al.: Cogent: uniqueness types and certifying compilation. J. Funct. Program. **31**, e25 (2021). https://doi.org/10.1017/S095679682100023X
11. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
12. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theore. Comput. Sci. **375**(1), 271–307 (2007). https://doi.org/10.1016/j.tcs.2006.12.035. https://www.sciencedirect.com/science/article/pii/S030439750600925X. festschrift for John C. Reynolds's 70th birthday
13. Patterson, D., Mushtak, N., Wagner, A., Ahmed, A.: Semantic soundness for language interoperability. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, pp. 609–624. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523703
14. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE (2002). https://doi.org/10.1109/LICS.2002.1029817
15. Rinus Plasmeijer, van Eekelen, M.: Clean version 2.2 language report. Language report (2011)
16. Scholz, S.B.: Single assignment C: efficient support for high-level array operations in a functional setting. J. Funct. Program. **13**(6), 1005–1059 (2003)
17. Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, R.: Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In: Schneider, H.J., Ehrig, H. (eds.) Graph Transformations in Computer Science. LNCS, vol. 776, pp. 358–379. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57787-4_23
18. Somogyi, Z., Henderson, F., Conway, T.C.: The execution algorithm of mercury, an efficient purely declarative logic programming language. J. Log. Program. **29**(1-3), 17–64 (1996). https://doi.org/10.1016/S0743-1066(96)00068-4